Linux 基礎とアプリケーション開発

実践ガイド

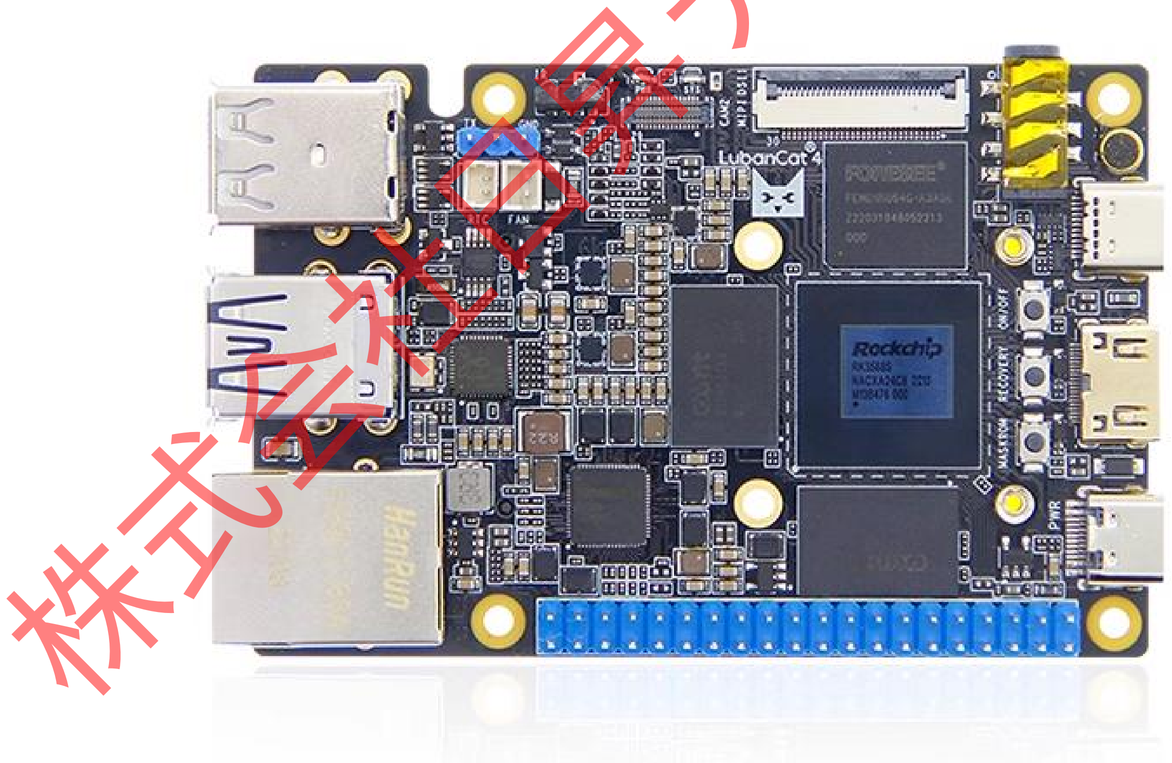
株式会社日昇テクノロジー

https://www.csun.co.jp

info@csun.co.jp

作成日 2024/06/07

株式会社日昇テクノロジー



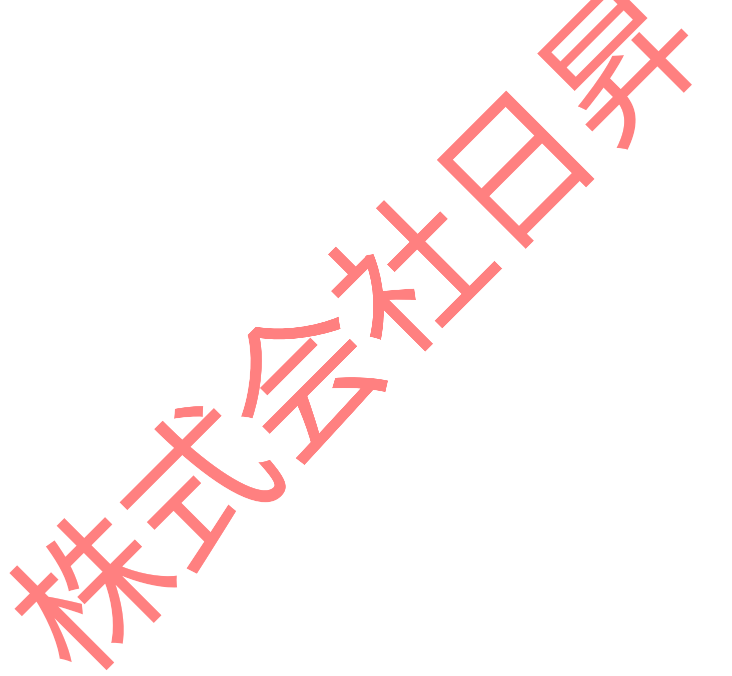
copyright@2024

- **修正履歴**

NO	バージョン	修正内容	修正日
1	Ver1.0	新規作成	2024/06/07

※ この文書の情報は、文書を改善するため、事前の通知なく変更されることがあります。最新版は弊社ホームページからご参照ください。「https://www.csun.co.jp」

※ (株)日昇テクノロジーの書面による許可のない複製は、いかなる形態においても厳重に禁じられています。


株式会社日昇テクノロジー

目　次

第 1 章 Linux 開発を学ぶ理由 33
1.1 LINUX の適用シーン 33
1.1.1 サーバー 33
1.1.2 組み込みデバイス 34
1.2 適応職業 35
1.2.1 サーバー運用 36
1.2.2 アプリケーションソフトウェアの開発 36
1.2.3 デバイスドライバ開発 36
1.3 プログラマーの宝庫 38
第 2 章 Linux 開発の学び方 39
2.1 組み込み Linux の学習ルート 39
2.1.1 どのユーザーグループに適しているか 39
2.1.2 教育理念 41
第 3 章 Linux システム入門 42
3.1 Linux システム入門 42
3.2 Linux カーネルとディストリビューション 42
3.2.1 Debian 系 44
3.2.2 Fedora 系 45
3.2.3 OpenSUSE 系 46

株式会社日昇テクノロジー

第 4 章 デスクトップ Linux を体験する 46

4.1 序文 46

4.1.1 HDMI インターフェーススクリーンの使用 46

4.1.2 MIPI スクリーンの使用 47

4.2 電源を入れる前の準備 48

4.2.1 外部デバイスの接続 48

4.2.2 電源供給 49

4.3 電源オン 49

4.3.1 ロックスクリーンインターフェース 50

4.3.2 デスクトップに入る 51

4.3.3 デスクトップの自動ログイン 52

4.4 ネットワーク接続 53

4.4.1 有線接続 53

4.4.2 Wi-Fi 接続 53

4.4.3 その他の接続 55

4.5 デスクトップの機能の簡単な紹介 55

第 5 章 ターミナル Linux（SSH）を体験する 62

5.1 前言 62

5.2 開始前の準備 63

5.2.1 ソフトウェアの準備 63

株式会社日昇テクノロジー

5.2.2 電源投入 63
5.3 ボードのスイッチオン 64
5.3.1 コンピュータネットワークの設定 64
5.4 ネットワーク接続 67
5.5 機能紹介 67
5.5.1 簡単なコマンドラインの使い方 67
5.6 ソフトウェアのアップデート 69
5.6.1 ソフトウェアのインストールとアンインストール 70
第 6 章 ターミナル Linux の体験（シリアル接続） 71
6.1 はじめに 71
6.2 電源投入前の準備 72
6.2.1 周辺機器の接続 72
6.2.2 ソフトウェアの準備 74
6.2.3 電源投入 76
6.3 デバイスの電源を入れる 77
6.4 ネットワーク接続 78
6.4.1 ネットワークケーブルの接続 78
6.4.2 Wifi 接続 79
6.5 機能の紹介 81
6.5.1 シンプルなコマンドラインを使う 81

株式会社日昇テクノロジー

6.5.2 ソフトウェアの更新 82
6.5.3 ソフトウェアのインストールとアンインストール 83
第 7 章 Linux コマンドライン 85
7.1 ターミナルの開き方 87
7.1.1 シリアル接続 87
7.1.2 SSH 接続 87
7.1.3 デスクトップからターミナルを開く 88
7.2 コマンドプロンプト 88
7.3 コマンドライン体験 89
7.4 コマンドの形式とヘルプ 91
7.4.1 コマンドの形式 91
7.4.2 コマンドのヘルプ 91
7.4.3 オートコンプリート 92
7.4.4 コマンドの終了とキャンセル 94
7.4.5 コマンドとは実際には何か 95
7.5 一般的なコマンド 97
7.5.1 pwd コマンド 97
7.5.2 cd コマンド 97
7.5.3 mkdir コマンド 99
7.5.4 touch コマンド 100

株式会社日昇テクノロジー

7.5.5 ls コマンド 100
7.5.6 cat コマンド 102
7.5.7 echo コマンド 102
7.5.8 ファイルへの出力リダイレクト 103
7.5.9 rm コマンド 104
7.5.10 su コマンド 107
7.5.11 sudo コマンド 107
7.5.12 clear コマンド 109
7.5.13 reboot/poweroff コマンド 109
第 8 章 ユーザーとファイル 110
8.1 ユーザーとユーザーグループ 110
8.2 ファイル 111
8.3 chmod コマンド 114
第 9 章 Linux ファイルディレクトリ 117
9.1 ホームディレクトリ 117
9.2 ルートディレクトリ 118
9.2.1 ルートディレクトリ構造 121
9.2.2 /bin ディレクトリ 123
9.2.3 /sbin ディレクトリ 123
9.2.4 /etc ディレクトリ 124

株式会社日昇テクノロジー

9.2.5 /root ディレクトリ 124
9.2.6 /home ディレクトリ 124
9.2.7 /lib ディレクトリ 124
9.2.8 /dev ディレクトリ 125
9.2.9 /proc ディレクトリ 125
9.2.10 /sys ディレクトリ 125
9.2.11 /tmp ディレクトリ 125
9.2.12 /boot ディレクトリ 125
9.2.13 /mnt ディレクトリ 126
9.2.14 /media ディレクトリ 126
9.2.15 /usr ディレクトリ 126
第 10 章 テキストエディタ 127
10.1 Vi/Vim エディタ 127
10.1.1 Vim の使用デモ 127
10.1.2 Vim3 つの作業モード 130
10.1.3 挿入モード 131
10.1.4 ノーマルモード 132
10.1.5 コマンドラインモード 133
10.1.6 Vi/Vim で簡単な sh スクリプトを作成 133
第 11 章 この章のコードと記事の取得 134

株式会社日昇テクノロジー

11.1 Git とは 135
11.2 Git の特徴 135
11.3 Git と GitHu 135
11.4 Git を使用してプロジェクト資料をダウンロードする 136
11.4.1 Linux システム 136
11.4.2 Windows システム 137
第 12 章 プログラムのコンパイル 139
12.1 GCC コンパイルツールチェーン 140
12.1.1 GCC コンパイラ 141
12.1.2 Binutils ツールセット 142
12.1.3 glibc ライブラリ 143
12.2 HelloWorld 144
12.2.1 作業ディレクトリの作成 144
12.2.2 コードファイルの作成 145
12.2.3 コンパイルと実行 146
12.3 GCC コンパイルプロセス 147
12.3.1 基本構文 147
12.3.2 プリプロセス段階 150
12.3.3 コンパイル段階 153
12.3.4 アセンブル段階 155

株式会社日昇テクノロジー

12.3.5 リンク段階 157

12.4 HelloWorld アドバンス版-1 159

12.4.1 コンパイルして実行 161

12.5 HelloWorld の進化版-2 162

12.5.1 コンパイルと実行 166

第 13 章 Linux システム下のプログラム 167

13.1 マイクロコントローラでの Hello World 167

13.2 Linux システムでの Hello World 168

第 14 章 vscode での便利なデバッグ開発 172

14.1 環境設定 172

14.1.1 vscode の使用 172

14.1.2 LubanCat の IP アドレスの設定確認 172

14.1.3 vscode でボードに接続 173

14.1.4 パスワードなしでログイン 177

14.1.5 ボードへの接続 179

第 15 章 Makefile の紹介 181

15.1 Makefile とは何か 181

15.2 Makefile 概要 182

第 16 章 Makefile を使ったコンパイル制御 185

16.1 Makefile の小実験 186

株式会社日昇テクノロジー

16.2 プログラムの Makefile を使用したコンパイル 190

16.2.1 複数のファイルを GCC でコンパイルする 190

16.2.2 Makefile を使用したコンパイル 192

16.3 ターゲットと依存関係 195

16.4 偽目標 196

16.5 デフォルトルール 199

16.6 変数の使用 201

16.6.1 基本構文 201

16.6.2 デフォルトルールの改造 202

16.6.3 リンクルールの改造 204

16.6.4 その他の自動変数 206

16.7 関数の使用 206

16.7.1 関数の形式と例 207

16.7.2 多層構造プロジェクトの Makefile 209

16.8 ブランチの使用 213

第 17 章 ファイル操作とシステムコール 218

17.1 ストレージデバイスファイルシステム 219

17.2 仮想ファイルシステム 222

17.2.1 procfs ファイルシステム 222

17.2.2 sysfs ファイルシステム 226

株式会社日昇テクノロジー

17.2.3 devfs ファイルシステム 228
17.3 仮想ファイルシステム 228
17.4 Linux システムコール 230
17.5 ファイル操作（C 標準ライブラリ） 231
17.5.1 一般的なファイル操作（C 標準ライブラリ） 232
17.5.2 実験コード分析 235
17.5.3 Makefile 説明 237
17.5.4 コンパイルとテスト 240
17.6 ファイル操作（システムコール） 240
17.6.1 open 関数 241
17.6.2 read 関数 242
17.6.3 write 関数 243
17.6.4 close 関数 243
17.6.5 lseek 関数 243
17.7 実験コード分析 244
17.7.1 実行フロー 246
17.7.2 ヘッダーファイルのディレクトリ 247
17.7.3 一般的なヘッダーファイル 248
17.8 コンパイルとテスト 249
17.9 どちらを選ぶべきか 250

株式会社日昇テクノロジー

第 18 章 デバッグツール......251

18.1 gdb......251

18.2 strace......253

第 19 章 GPIO サブシステム......258

19.1 紹介......258

19.1.1 GPIO デバイスディレクトリ......259

19.1.2 GPIO デバイス属性......261

19.2 ピン番号の変換......262

19.3 GPIO sysfs インターフェース制御......263

19.3.1 コマンドライン......263

19.3.2 プログラム作成......264

19.3.3 コンパイル＆実行......271

19.4 libgpiod を使った IO 制御......272

19.4.1 コマンドライン制御......272

19.4.2 libgpiod プログラミング......273

19.5 system 関数を使ったプログラミング......280

第 20 章 input サブシステム......281

20.1 input サブシステム......281

20.2 input イベントディレクトリ......281

20.2.1 evtest ツールを使ったテスト......281

株式会社日昇テクノロジー

20.2.2 input イベント構造 283

20.2.3 input イベントデバイス名 286

20.3 ボードキー検出実験 287

20.3.1 実験コード分析 287

20.3.2 コンパイル 292

20.3.3 実行 292

第 21 章 シリアル通信 293

21.1 シリアルピンの関係 293

21.2 シリアルインターフェイスの有効化 294

21.3 シリアルデバイスの確認 294

21.4 シリアル通信実験（Shell） 295

21.4.1 シリアルの接続 295

21.4.2 シリアル 3 の通信パラメータの照会 295

21.4.3 シリアルのボーレートの変更 296

21.4.4 エコーの無効化 296

21.4.5 Windows ホストとの通信 296

21.5 シリアル通信実験（システムコール） 303

21.5.1 実験 304

21.5.2 コード分析 309

21.6 ioctl システムコール 321

株式会社日昇テクノロジー

21.6.1 ioctl の原型 322
21.6.2 tcgetattr と tcsetattr の代わりに ioctl を使用する 323
21.6.3 コンパイル 329
21.6.4 実行 329
21.7 glibc のソースコードを見る 331
第 22 章 I2C 通信 334
22.1 I2C 通信プロトコルの概要 334
22.1.1 I2C の物理層 334
22.1.2 プロトコル層 336
22.2 本ボードの I2C ピン 341
22.2.1 IIC 通信インターフェースの有効化 342
22.3 IIC デバイスの確認 343
22.4 デバイスの接続 343
22.5 IIC 第三者ツール - i2c-tools 344
22.5.1 i2cdetect その他のコマンド 344
22.6 ジャイロスコープセンサーデータの読み取り実験 347
22.6.1 実験説明 347
22.6.2 ioctl 関数 348
22.6.3 アプリケーションの作成 349
22.6.4 コード分析 357

株式会社日昇テクノロジー

22.7 OLED ディスプレイ実験 359
22.7.1 ハードウェア接続 360
22.7.2 コンパイル&実行 360
22.7.3 コード分析 361
第 23 章 SPI 通信 370
23.1 SPI 通信プロトコルの紹介 370
23.1.1 SPI の物理層 370
23.1.2 プロトコル層 372
23.1.3 拡張 SPI プロトコル（シングル/デュアル/クアッド/オクタル SPI） 376
23.2 SPI 関連データ構造と ioctl 関数 377
23.3 LubanCat ボードの SPI インターフェース 379
23.3.1 SPI 通信インターフェースの有効化 380
23.3.2 SPI デバイスの確認 381
23.4 SPI ループバック通信テスト実験 382
23.4.1 ハードウェア接続： 382
23.4.2 プログラムの作成 382
23.4.3 コンパイル＆実行 385
23.4.4 コード分析 386
23.5 SPI_OLED ディスプレイ実験 395
23.5.1 ハードウェア説明： 396

株式会社日昇テクノロジー

23.5.2 コード解析 397
第 24 章 PWM（パルス幅変調） 405
24.1 パルス幅変調 405
24.1.1 PWM とは 405
24.1.2 PWM の周波数 405
24.1.3 PWM の周期 405
24.1.4 占有率 406
24.1.5 PWM の原理 406
24.2 PWM ピン 407
24.2.1 PWM インターフェース機能の有効化 408
24.3 PWM デバイスの確認 408
24.4 PWM 制御方法(shell) 409
24.5 PWM 制御方法（システムコール） 410
第 25 章 カメラ 414
25.1 カメラ設定 415
25.1.1. シングルカメラ 415
25.1.2. デュアルカメラ 417
25.1.3. トリプルカメラ 418
25.2 カメラ情報を表示する 420
25.2.1 カメラデバイスを一覧表示する 420

株式会社日昇テクノロジー

25.2.2 カメラのフォーマットと解像度を確認する 420
25.2.3 カメラがサポートする設定パラメータを確認する 421
25.3 ビデオ 423
25.4 写真 423
25.5 デスクトップでの撮影 423
25.5.1 使用方法 423
第 26 章 音声 429
26.1 サウンドカードの設定 429
26.1.1 デバイス設定とツールのインストール 429
26.1.2 サウンドカードの音量設定 430
26.2 デスクトップでの音声再生 432
26.2.1 音楽の再生 432
26.2.2 録音 435
26.3 コマンドラインでの音声再生 436
26.3.1 WAV 形式の音楽の再生 436
26.3.2 録音 437
26.3.3 複数形式の再生 437
第 27 章 スクリーン 443
27.1 回転タッチスクリーン 443
27.2 タッチスクリーンのスクリーンへのバインド 444

株式会社日昇テクノロジー

27.3 スクリーンの回転方向 445
27.3.1 コマンドライン設定 445
27.3.2 デスクトップ設定 446
27.4 スクリーンインターフェース 447
27.4.1 mipi-dsi インターフェース 447
27.4.2 HDMI インターフェース 448
27.5 スクリーン切り替え 448
27.5.1 MIPI スクリーン 448
27.5.2 HDMI 449
27.5.3 複数スクリーン異なる表示 449
27.6 デスクトップログインを無効にし、スクリーンをオンに保つ 450
第 28 章 スクリーン表示（フレームバッファ） 450
28.1 LubanCat-RK ボード 450
28.2 フレームバッファの紹介 450
28.3 フレームバッファアプリケーション 452
28.3.1 コード分析 457
28.4 ダブルバッファリング設計 464
28.4.1 前書き 464
28.4.2 ダブルバッファリング 464
28.4.3 実装思路 465

株式会社日昇テクノロジー

28.5 参考資料 466

第 29 章 画面表示(DRM) の紹介 466

29.1 前置き 466

29.2 DRM の紹介 467

29.3 DRM 表示 468

29.3.1 DRM と framebuffer の違い 468

29.3.2 DRM 表示システムの分析 469

29.3.3 LubanCat-RKdrm 分析 472

第 30 章 DRM アプリケーションプログラミング - レガシーインターフェース 479

30.1 最も簡単な DRM (drm-single) 479

30.1.1 詳細コード分析 488

30.1.2 コンパイル 500

30.1.3 実行 500

30.2 ダブルバッファ DRM（drm-double.c） 500

30.2.1 単一フレームバッファ ダブルバッファ 501

30.2.2 コンパイル 501

30.2.3 実行 502

30.2.4 複数フレームバッファ ダブルバッファ 505

30.2.5 コンパイル 506

30.2.6 実行 506

株式会社日昇テクノロジー

30.2.7 まとめ 510

30.3 ページフリップ 510

30.3.1 ページフリップの実験 511

30.3.2 コンパイル 511

30.3.3 実行 512

30.3.4 プログラム分析 512

30.4 drm-planes 515

30.4.1 drm-planes 実験 516

30.4.2 コンパイル 516

30.4.3 実行 516

30.4.4 プログラム分析 518

30.5 レガシーインターフェース関数 521

30.6 まとめ 521

30.7 デュアルスクリーン異表示 522

第 31 章 DRM アプリケーションプログラミング - アトミックインターフェース 524

31.1 Property（プロパティ） 525

31.2 DRM アプリケーションプログラミング（drm-atomic-crtc） 527

31.2.1 コンパイル 537

31.2.2 実行 538

31.2.3 実験分析 539

株式会社日昇テクノロジー

31.2.4 まとめ 547

31.3 DRM アプリケーションプログラミング（drm-atomic-planes） 548

31.3.1 コンパイル 548

31.3.2 実行 549

31.3.3 実験分析 550

31.3.4 まとめ 557

31.4 DRM アプリケーションプログラミング（drm-atomic-page-flip） 558

31.4.1 コード分析 558

31.4.2 コンパイル 563

31.4.3 実行 564

31.5 Makefile 管理 564

31.6 スクリーン二つそれぞれ表示 568

第 32 章 文字表示 570

32.1 ASCII 文字の表示（8x16） 571

32.1.1 実装方法 571

32.1.2 実験 575

32.1.3 プログラム分析 577

32.2 漢字の表示（8x16） 580

32.2.1 実装方法 580

32.2.2 実験 580

株式会社日昇テクノロジー

32.2.3 プログラム分析 582
32.3 ベクターフォント(FreeType) 586
32.3.1 実装方法 586
32.3.2 単一のベクターフォントの表示 588
32.3.3 Freetype で文字列を表示 605
32.3.4 freetype の上級 620
第 33 章 画像表示 620
33.1 BMP 621
33.1.1 BMP ファイルの取得 621
33.1.2 ファイルフォーマット解析 621
33.1.3 コードの位置 626
33.1.4 コード構造 627
33.1.5 コンパイル & 実行 628
33.1.6 コード分析 629
33.2 JPEG 639
33.2.1 前言 639
33.2.2 コード位置 640
33.2.3 コード構造 640
33.2.4 コンパイル&実行 642
33.2.5 コード分析 642

株式会社日昇テクノロジー

33.2.6 まとめ 650

33.3 PNG 650

33.3.1 前言 650

33.3.2 コード位置 651

33.3.3 コード構造 651

33.3.4 コンパイル＆実行 652

33.3.5 コード分析 653

33.3.6 まとめ 660

第 34 章プロセス 660

34.1 簡単にプロセスを理解する 661

34.2 プロセスの確認 662

34.2.1 プロセス ID 662

34.2.2 親プロセス ID 663

34.2.3 親プロセスと子プロセス 664

34.3 プログラムとプロセス 665

34.3.1 プログラムの概念 665

34.3.2 プロセスの概念 666

34.3.3 プログラムがプロセスになる過程 667

34.3.4 まとめ 668

34.4 プロセス状態 669

株式会社日昇テクノロジー

34.5 プロセス状態の変化 670
34.6 新しいプロセスの起動 674
34.6.1 `system()`関数によるプロセス実験 674
34.6.2 fork()によるプロセス実験 679
34.6.3 exec 系列関数によるプロセス実験 686
34.7 プロセスの終了 690
34.8 プロセスの待機 694
34.8.1 wait()関数 694
34.8.2 waitpid() 699
第 35 章 信号 700
35.1 信号の基本概念 700
35.1.1 概要 700
35.1.2 システムがサポートする信号 701
35.1.3 "非リアルタイム信号とリアルタイム信号" 702
35.2 信号の処理 703
35.2.1 実験分析 705
35.3 信号を捕捉するための関連 API 関数 707
35.3.1 signal() 707
35.3.2 sigaction() 712
35.4 信号関連 API 関数 719

株式会社日昇テクノロジー

35.4.1 kill() 720

35.4.2 raise() 723

35.4.3 alarm() 727

第 36 章 パイプについて 733

36.1 パイプの基本概念 733

36.2 パイプの分類 735

36.2.1 匿名パイプ（PIPE） 736

36.2.2 命名パイプ（FIFO） 737

36.3 pipe()関数 738

36.3.1 実験分析 740

36.4 fifo()関数 746

36.4.1 実験分析 749

第 37 章 メッセージキュー 757

37.1 メッセージキューの基本概念 758

37.2 メッセージキューとシグナル、パイプの比較 759

37.3 メッセージキュー関数説明 759

37.3.1 msgget() 関数 760

37.4 メッセージの送受信 762

37.4.1 msgsnd() 送信関数 762

37.4.2 msgrcv() 受信関数 764

株式会社日昇テクノロジー

37.4.3 msgctl() メッセージキュー操作 765
37.5 メッセージキュー例 766
37.5.1 送信プロセス 767
37.5.2 受信プロセス 770
37.5.3 コンパイル及びテスト 774
第 38 章 System V IPC セマフォ 776
38.1 プロセスセマフォの基本概念 776
38.2 セマフォの動作原理 777
38.2.1 セマフォの作成または取得 778
38.3 semget 作成/取得関数 778
38.4 セマフォ操作 779
38.4.1 semop() PV 操作関数 779
38.4.2 semctl() 属性関数 782
38.5 セマフォ例 783
38.5.1 実験操作 790
第 39 章 共有メモリ 791
39.1 共有メモリの基本概念 791
39.2 shmget() 共有メモリ作成関数 793
39.3 shmat() マッピング関数 794
39.4 shmdt() 関数によるマッピング解除 796

株式会社日昇テクノロジー

39.5 shmctl() 関数による属性の取得または設定 ... 796
39.6 使用例 ... 798
39.6.1 共有メモリ書き込みプロセス ... 799
39.6.2 共有メモリ読み取りプロセス ... 803
39.6.3 実験操作 ... 807
第 40 章 スレッド ... 809
40.1 スレッドとプロセス ... 809
40.2 スレッドの作成 ... 810
40.2.1 pthread_create()によるスレッドの作成 ... 811
40.3 スレッド属性 ... 812
40.3.1 スレッドオブジェクト属性の初期化 ... 814
40.3.2 スレッド属性オブジェクトの破棄 ... 814
40.3.3 スレッドの分離状態 ... 815
40.3.4 スレッドのスケジューリングポリシー ... 816
40.3.5 スレッドの優先順位 ... 818
40.3.6 スレッドスタック ... 819
40.4 スレッドの終了 ... 819
40.5 スレッドの実験 ... 821
40.5.1 実験操作 ... 824
第 41 章 POSIX セマフォ ... 825

株式会社日昇テクノロジー

41.1 POSIX セマフォの基本概念......825

41.2 POSIX 名前付きセマフォ......828

41.3 POSIX 名前なしセマフォ......830

41.4 POSIX セマフォの使用例......831

41.4.1 名前付きセマフォ......831

41.4.2 無名セマフォの例：......837

第 42 章 POSIX ミューテックス......844

42.1 ミューテックスの基本概念......844

42.2 ミューテックスの初期化......847

42.2.1 静的初期化......847

42.2.2 動的初期化......848

42.3 ミューテックスの取得と解放......849

42.4 ミューテックスの破棄......850

42.5 ミューテックスの実験......850

42.5.1 実験操作......855

第 43 章 ネットワークプログラミング......857

43.1 ネットワーク関連知識の簡単な紹介......857

43.1.1 ネットワークプロトコルの階層モデル......860

43.1.2 プロトコル層間のデータのカプセル化とデカプセル化......862

43.2 IP プロトコル......863

株式会社日昇テクノロジー

43.2.1 IP アドレスの概要 864
43.2.2 IP アドレスのアドレス割り当て 865
43.2.3 特別な IP アドレス 867
43.3 UDP プロトコル 870
43.4 TCP プロトコル 871
43.4.1 TCP プロトコルの概要 871
43.4.2 TCP の特徴 872
43.5 ポート番号の概念 875
43.6 TCP セグメントの構造 876
43.7 TCP 接続の確立 879
43.8 TCP 接続の終了 882
43.9 TCP の状態 884
第 44 章 ソケット 888
44.1 ソケットの概要 889
44.2 socket() 890
44.3 bind() 892
44.4 connect() 895
44.5 listen() 895
44.6 accept() 896
44.7 read() 898

株式会社日昇テクノロジー

44.8 recv() 899

44.9 write() 901

44.10 send() 903

44.11 sendto 904

44.12 close() 904

44.13 ioctlsocket() 904

44.14 getsockopt()、setsockopt() 906

44.15 TCP クライアント実験 907

44.16 TCP サーバーの実験 910

44.17 実験現象 913

第 45 章 select、poll、epoll の違いについて深く理解する 915

45.1 序章 915

45.2 同期 I/O 915

45.3 非同期 I/O 916

45.4 ブロッキング I/O 916

45.5 ノンブロッキング I/O 918

45.6 マルチプレクシング I/O 918

45.6.1 select 920

45.6.2 poll 924

45.6.3 epoll 924

株式会社日昇テクノロジー

株式会社日昇テクノロジー

第 1 章 Linux 開発を学ぶ理由

1.1 LINUX の適用シーン

一般ユーザーは、PC やスマートフォンに頻繁に触れるため、Windows、iOS、Android OS は馴染みがありますが、Linux OS についてはあまり知らないことが多いです。その主な理由は、Linux の適用シーンが主にサーバーや組み込みデバイスに限られ、消費者向けのデスクトップシステム分野で優位を占めていないからです。しかし、ある意味で、Android システムも Linux の一種と言えます。なぜなら、それは Linux カーネルを基に開発されており、Linux のオープンソースライセンスを避けているため、Linux コミュニティに受け入れられていないものの、実質的には Linux に基づいているからです。Linux の適用シーンは、その特徴によって決定されます。主な特徴としては、オープンソース、安全性、安定性、強力なネットワーク機能、さまざまなプラットフォームのプロセッサをサポートすることが挙げられます。

1.1.1 サーバー

サーバーとは、計算サービスを提供する装置のことを指し、ウェブサーバー、ファイルサーバー、データベースサーバー、メールサーバー、ドメインネームサーバー、プロキシサーバーなどがあります。インターネット上で行うすべての操作の裏には、サーバーとのやり取りがあります。例えば、ウェブページの閲覧、オンラインショッピング、チャット、ネットワークゲームの実行、オンラインビデオの視聴など、インターネットに接続する必要があるアプリケーションはすべて、サーバーが支えとなっています。

サーバーの実体は様様で、小さなシングルボードコンピューターから、大規模なコンピュータクラスターまであります。実際、あなたが使用している個人用コンピューターに、対応するサービスソフトウェアをインストールし、他のコンピューターにネットワーク経由でリソースを提供することで、それもサーバーに変身します。サーバーの本質もコンピューターであり、通常はモニターがなく、マウスやキーボードも必要ありません。

ビジネス用では、サーバーで実行されるオペレーティングシステムには主に Linux、Windows、UNIX

株式会社日昇テクノロジー

があり、その中でも Linux はオープンソースで、使用コストが低く、安全で安定しているため、サーバー用のシステムとして大部分を占めています。

興味のある読者は、Netcraft のウェブサイト http://www.netcraft.com を訪れることで、他のウェブサイトがどのオペレーティングシステム上で動作しているかを調べることができます。詳細は以下の図を参照してください。

What's that site running?

Find out what technologies are powering any website:

www.baidu.com

Hosting History

Netblock owner	IP address	OS	Web server	Last seen Refresh
Baidu USA LLC 20883 Stevens Creek BLVD Cupertino CA US 95014	104.193.88.123	Linux	bfe/1.0.8.18	31-Jul-2019
Baidu USA LLC 20883 Stevens Creek BLVD Cupertino CA US 95014	104.193.88.77	Linux	bfe/1.0.8.18	28-Jul-2019
Baidu USA LLC 20883 Stevens Creek BLVD Cupertino CA US 95014	104.193.88.123	Linux	bfe/1.0.8.18	27-Jul-2019
Baidu USA LLC 20883 Stevens Creek BLVD Cupertino CA US 95014	104.193.88.77	Linux	bfe/1.0.8.18	26-Jul-2019
Baidu USA LLC 20883 Stevens Creek BLVD Cupertino CA US 95014	104.193.88.123	Linux	bfe/1.0.8.18	24-Jul-2019

多くの有名なウェブサイトは Linux システム上で運用されており、マイクロソフトの公式サイト（www.microsoft.com）も例外ではありません。特に、スーパーコンピューターは特別なサーバーであり、現在世界の TOP500 にランキングされているコンピューターの運用 os はすべて Linux です。

1.1.2 組み込みデバイス

組み込みデバイスは Linux os のもう一つの主戦場であり、これも本文章で主に説明されている Linux のアプリケーション方向です。組み込みデバイスの定義は比較的あいまいですが、基本的に PC、クラスターサーバー、スーパーコンピューターを除外した後、CPU（マイクロコントローラーなどを含む）を搭載し、設定されたプログラムに従って動作する電子デバイスのほとんどが組み込みデバイスに分類されます。

組み込みデバイスの種類は非常に豊富で、日常生活で使用される携帯電話、スマートウォッチ、各種家電、おもちゃ、ルーター、車用電子システム、工業応用では電力システムの監視、環境モニタリング、工

株式会社日昇テクノロジー

業用ロボット、スマート宅配ボックス、ハンドヘルド POS 端末、地下鉄の改札システム、駐車場管理、衛星、月面車などが含まれます。

組み込みデバイスの種類から、その応用シーンが断片化していることが分かります。これらの内部の電子システムは一般に、デバイスの機能に特化した制御を行います。一部の組み込みデバイスは os を使用せず、一部は freeRTOS などのリアルタイム os を使用し、また別の高性能な組み込みデバイスは Linux os を使用します。Linux os を使用する組み込みデバイスは、通常、Linux os の以下の特性で評価しています：

- 組み込みデバイスで使用されるプロセッサーは多種多様であり、Linux システムは x86、ARM、PowerPC、MIPS など、異なるプラットフォームのプロセッサー上で動作するサポートを提供します。
- コードがオープンソースであり、カスタマイズ可能であるため、特定のシナリオに合わせてカスタマイズするのに非常に適しています。カスタマイズされた Linux カーネルとファイルシステムを合わせることで、50MB 以内に収めることが可能であり、これによりハードウェアリソースとコストを節約できます。
- Python、Java、C++などの各種プログラミング言語や、Opencv、TensorFlow などのライブラリやフレームワークに対するサポートが良好です。freeRTOS などのリアルタイム os を使用する場合、直接的なサポートは難しいことが多いです。
- アプリケーションが豊富で、音楽プレーヤーやデータベースなど、既製のアプリケーションを直接使用できます。
- ネットワーク機能が強力で、インターネット接続が必要なアプリケーションの開発が非常に便利です。

1.2 適応職業

Linux の応用シーンに基づいて、Linux 技術者は主にサーバー運用、アプリケーションソフトウェアの開発、およびデバイスドライバーの開発の方向に分けています

株式会社日昇テクノロジー

1.2.1 サーバー運用

サーバー運用エンジニアは、サーバーアプリケーションがより効率的に、安定して、安全にサーバー上で動作するように実行環境を構築することを主に担当しています。例えば、オペレーティングシステムのインストール、コード実行環境のデプロイ、脆弱性の修正、サーバーの負荷監視、ログ分析などが含まれます。

1.2.2 アプリケーションソフトウェアの開発

Linux アプリケーションソフトウェアの開発とは、Linux システム上で動作するアプリケーションプログラムを開発することです。具体的には、アプリケーションはサーバー、デスクトップ、組み込みの方向に分けられます。サーバー方向では、ウェブサイトのバックエンドやデータベースシステムなどがあります。サーバー方向のアプリケーション開発は通常、オペレーティングシステムとはあまり関係がありません。デスクトップ開発は、Windows システム下での QQ、WPS、シリアルポートデバッグアシスタントなどのアプリケーション開発と同様です。組み込み方向のソフトウェアは、特定のデバイスのアプリケーションシナリオに合わせた開発が主で、掃除ロボットやルーターの制御プログラムなどが含まれます。

1.2.3 デバイスドライバ開発

デバイスドライバーは、その本質もソフトウェアプログラムですが、ハードウェアとオペレーティングシステムの間の階層に位置し、ハードウェアとオペレーティングシステムの間の橋渡しをするものです。詳細は以下の図を参照してください。

株式会社日昇テクノロジー

アプリケーションプログラム
HTTP,FTP,TFTP
SMTP,Telnet 等
ライブラリ
ユーザーランド
システムコール
プロセス管理
タイマ管理
メモリ管理
ファイル
システム
デバイス制御
(USB,SD等)
ネットワーク
プロトコル
スタック
(TCP,UDP,
IP,PPP 等)
Linuxカーネル
同時並行処理
仮想メモリ
ファイルと
ディレクトリ
TTYとデバイス
ネットワーク通信
アーキテクチャ
依存のコード
メモリ
マネージャ
ブロック型
デバイス
ドライバ
キャラクタ型
デバイス
ドライバ
ネットワーク
サブシステム
ネットワーク
ドライバ
ハードウェア
CPU
メモリ
ディスクやCD
コンソール、
シリアル通信等
ネットワーク
インターフェース

オペレーティングシステムを使用しない、または小型リアルタイムオペレーティングシステム（例えばfreeRTOS）を使用するデバイスにとって、ハードウェア関連のドライバーとアプリケーションはしばしば混在しているため、ドライバーとアプリケーションの厳密な区分が行われないことが多く、これがハードウェアプラットフォームを変更した時のアプリケーションの移植を困難にしています。Linux システムを使用したデバイスでは、ハードウェアはオペレーティングシステムによって管理されます。Linux システムの重要な設計哲学の一つは「すべてはファイルである」というものです。ハードウェアデバイスもシステムにとってはファイルの一つであり、そのためシステムは open、write、read、close などの統一されたファイル操作インターフェースを上層のアプリケーションに提供します。アプリケーションはこれらのインターフェースを利用してデバイスファイルにアクセスし、ハードウェアデバイスの初期化、書き込み、

株式会社日昇テクノロジー

読み取り、および閉じるなどの操作を実行することができます。

システムがアプリケーションからのこれらのアクセス要求を受け取ると、それに応じて具体的なデバイスの操作関数を下位で検索し、該当する関数を呼び出して要求を処理します。そして、これら異なるデバイスに対する具体的な操作関数こそがデバイスドライバーです。したがって、デバイスドライバーとは、ハードウェアとオペレーティングシステムをつなぐプログラムであり、Linux システムは統一された形式を提供し、デバイスドライバーの開発エンジニアは、具体的なデバイスに対してシステムの形式要求に従って対応するデバイス操作関数を実装します。

この本は主に組み込みデバイスのドライバー開発に関する説明を主目的としており、アプリケーションソフトウェア開発を目指すエンジニアにとっても、この本の内容を通じて Linux の下層に関連する知識を習得することは、アプリケーション開発にも大いに役立ちます。

1.3 プログラマーの宝庫

Linux は公開された宝庫と言えます。それはオープンソースの世界の代リスト作であり、そのカーネルのソースコードは完全に公開されています。あなたがそれに没頭する意欲があれば、必ず豊富な収穫が得られます。Linux 開発を学ぶことで、コマンドラインの使用に慣れることができます；オペレーティングシステムの原理を理解することができます；リンクリスト、赤黒木、ハッシュテーブルなどの一般的なデータ構造を学ぶことができます；オープンソースコミュニティの文化を理解し、それぞれの優れたコード実装を学ぶことができます、例えば一般的な C 言語ライブラリの libc など；Git バージョン管理システム、GCC コンパイラなどの様々な一般的な開発ツールにも慣れることができます。

これは、伝統的な MCU 開発や Windows 開発に従事している時にこれらを学ぶことができないというわけではありませんが、これらの内容はしばしば包装されていて、直接に根本に追いかけることができないか、隠されています。しかし、Linux 開発を行う時、関連する内容は直接ユーザーに公開されています。十分に深く掘り下げれば、常に驚きが待っています。例えば、Keil や IAR で開発している時に、C 言語ライブラリの printf 関数を呼び出した後に具体的に何が実行されるのか、コンパイルボタンをクリックし

株式会社日昇テクノロジー

た時に統合ツールがために何を処理してくれるのか、これらについてはなかなか理解することが難しいです。

長期間 MCU 開発に携わってきたプログラマーにとって、Linux 開発を学ぶことは、「原来コードの世界はずっとここで私を待っていたんだ！」と感嘆させるでしょう。

第 2 章 Linux 開発の学び方

2.1 組み込み Linux の学習ルート

ラズベリーパイの開発方式に基づいて、その一連の先進的な設計理念とソフトウェアアーキテクチャを参考にして、電子工作愛好家に強力で使いやすい DIY プラットフォームを提供することを目指しています。これにより、さまざまな斬新なアイデアを容易に実現できます。同時に、すべてのソフトウェアとチップマニュアルを完全にオープンソース化し、組み込み Linux 業界でキャリアを積みたい人が、さまざまなソフトウェアアーキテクチャやドライバプログラムを深く学ぶための障害を取り除きます。

2.1.1 どのユーザーグループに適しているか

組み込み Linux 学習者は大きく 2 つのカテゴリーに分けられます。一つ目は進級ユーザーで、大量の MCU 開発経験を持つ開発者が、より難易度高い仕事で給与の高い MPU 開発へ進むことを望んでいる人たちです。もう一つは学生ユーザーで、組み込み開発に初めて触れる大学生のグループが主です。

前者にとっては、仕事のプレッシャーと昇進・給与アップのニーズから、通常はコース学習の深さにより関心を持ち、学んだ知識を活用してスムーズに転職・昇進・給与アップができるかどうかに注目します。しかし後者にとっては、業界経験がなく、仕事のプレッシャーもないため、コースの面白さや機能性にもっと重点を置き、このコースを通じて何を作る能力があるのかを知りたがっています。

本チュートリアルは、両方のユーザーのニーズを考慮しています。まず、入門の障壁をできるだけ低くすること、これは主に 2 つの面からです。一方では、開発環境の構築を簡素化し、可能な限りさまざまなソフトウェアのバージョン問題やさまざまなコンピューターシステムの互換性問題を避けます。もう一方

株式会社日昇テクノロジー

では、組み込み Linux の高頻度スキルを先に学びます。高頻度スキルとは、実際の開発作業で頻繁に使用されるスキルのことです。次に、これらの高頻度スキルを使って、クールで興味深い製品を開発し、組み込み Linux のさまざまな業界をさらに理解し、組み込み Linux が実際の人間の生活に与える影響を体験します。最後に、組み込み Linux について全体的な理解を持った後、個人の興味や実際の仕事のニーズに応じて、組み込み Linux のある分野に深く研究を選択します。

通常、国内の伝統的な組み込み Linux 学習ルートは以下の通りです:

1. Linux の基本操作+C 言語の進級
2. ARM ボード開発
3. Linux システム移植（u-boot 移植、kernel 移植、ルートファイルシステムの構築）
4. Linux ドライバ開発
5. Linux アプリケーションプログラミング
6. プロジェクト実戦

上記の学習ルートに沿って学習すると、一方で、膨大な学習エネルギーを要することになり、たとえ一日中学習しても、通常は 4 ヶ月以上の時間が必要です。これは、まだ職業方向を決定していない学生ユーザーにとって、組み込み Linux を始めるためのハードルが非常に高いことを意味します。

一方、進級ユーザーの実際の仕事のニーズに合わないことがあります。なぜなら、ほとんどのコースが設計された時に、実際の組み込み Linux の仕事であまり使われない低頻度スキル（例えば ARM 開発など）で溢れているからです。そして、実際の仕事でシステムパフォーマンスの問題に直面したとき、実際に問題を解決する能力がないことが多いです。なぜなら、学んだ Linux システム移植は、単にチップ会社のチームが作成したシステムイメージを使って、いくつかのファイルをコピー＆ペーストし、いくつかの設定を変更して、それをシステムに再コンパイルしてインストールすることに過ぎず、実際には高度なスキルをあまり学んでいないからです。

株式会社日昇テクノロジー

2.1.2 教育理念

LubanCat は、Linux や Android を搭載したカード型コンピュータのブランドとして推し出したものです。こののカード型コンピュータは、豊富なハードウェア製品ラインを持ち、オペレーティングシステムの適応度が高く、オープンソースの教材資料が多数あり、アプリケーションの開発が簡単です。

優れた性能と教育、商業アプリケーション、工業制御などの分野をカバーする多様な製品ラインを活用して、広範なアプリケーションシナリオを実現しています：

- カード型コンピュータ：オフィス、プログラミング開発、家庭エンターテイメント、プログラミング教育など
- Linux サーバー：プライベートクラウド、ソフトルータ、NAS、個人 WEB サーバーなど
- 家庭用スマートハブ：TV ボックス、スマートホームコントロール、センサーデータ分析、セキュリティ監視など
- 産業化：電子広告板、自動販売機、ロボット、ドローンなど
- 組み込み開発ボード：組み込みプロジェクトの検証と開発を加速

LubanCat コンピュータは、ハードウェアからシステム、教材、アプリケーションに至るまで、豊富な資料とバージョンを提供しており、汎用性が高いです：

- ハードウェア：異なる性能のメインコントローラー、外部インターフェース、ストレージ容量、ボードサイズを具備
- システム：Ubuntu、Debian、Android などのシステムをサポート
- 教材：複数の教材セットを提供し、純粋なアプリケーション層のユーザーやシステム開発のユーザーをカバー。例えば Python、Qt、Android アプリ開発、Linux システムの使用とカーネル、ドライバー、イメージ作成
- アプリケーション：C/Python を使用した様々なハードウェアの制御、ROS ロボットシステムに基づ

株式会社日昇テクノロジー

く、製品マニュアル、システムソースコード、回路図ライブラリ、各種高品質 Linux 開発チュートリアルなどを含むがこれに限らない、充実したオープンソース資料を提供します。業界初心者の組み込みビギナーであっても、チュートリアルに従って開発を完了することができ、経験豊富な組み込みのベテランにとっては、製品の二次開発プロセスを加速させることができます。

第 3 章 Linux システム入門

本章では、Linux システムの基本的な使用法を主に説明します。これには、システムのいくつかの基本的な概念、インストール、および日常的なアプリケーションが含まれます。

3.1 Linux システム入門

Linux システムは、1991 年に Linus Torvalds（リーナス・トーバルズ）がニュースグループで発リストしたカーネルから発展しました。リリース初期から無料かつ自由に配布され、多ユーザー、多タスク、マルチスレッドをサポートし、POSIX 標準に準拠しているため、当時の主流システムである Unix のいくつかのツールソフトウェアを実行することができ、多くのユーザーと開発者を引き付け、徐々に発展してきました。Linux システムの発展史については、作者自身の自伝『Just For Fun』の読書をお勧めします。

3.2 Linux カーネルとディストリビューション

Linux システムと言うとき、その意味は通常、Linux カーネルを使用したオペレーティングシステムを指します。Linux カーネルは、ハードウェアの制御、ファイルシステムの管理、プロセス管理、ネットワーク通信などを担当していますが、それ自体ではユーザーに必要なツールやアプリケーションソフトウェアを提供していません。Linux カーネルはオープンソースであるため、個人や企業はそのルールの下で、Linux カーネルにさまざまなシステム管理ソフトウェアやアプリケーションツールを組み合わせて、完全に使用可能なオペレーティングシステムを構築しています。このようなシステムを Linux ディストリビューション（distribution）と呼びます。完全な Linux システムは、まるで自動車のようで、Linux カーネルが最も重要なエンジンを構成し、異なるディストリビューションは同じエンジンを使用する異なる車種に

株式会社日昇テクノロジー

相当します。

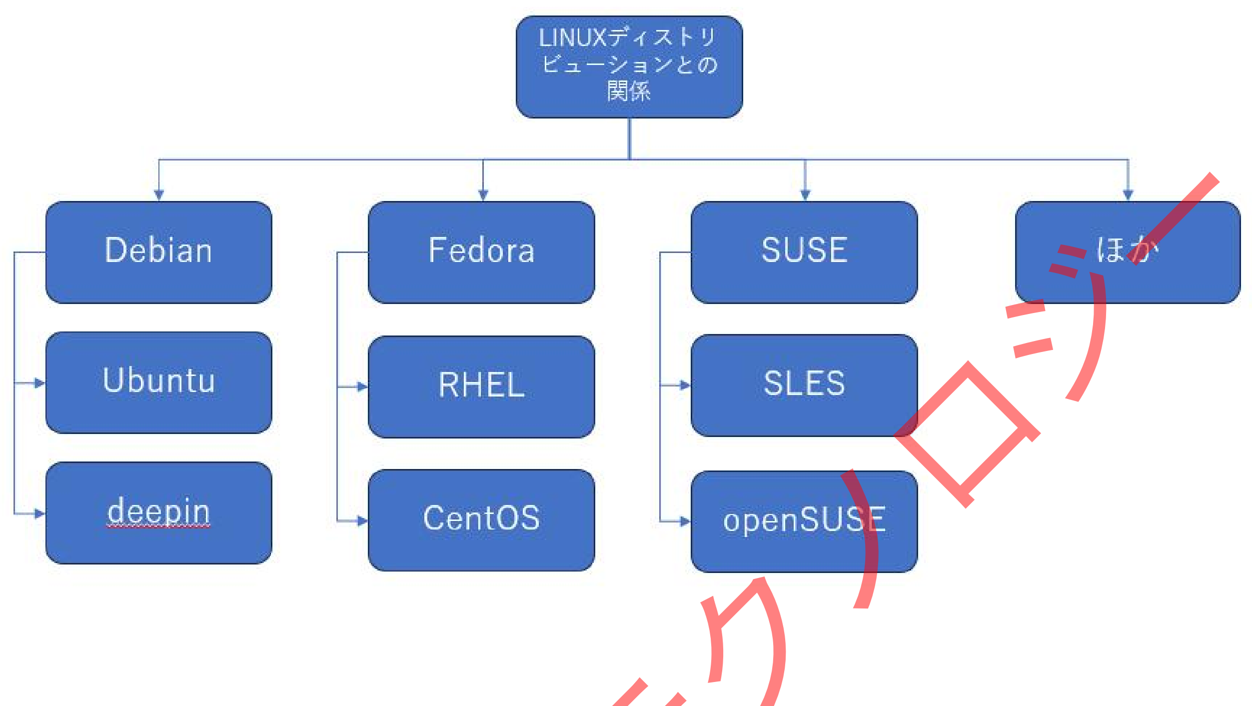
LINUXのシステム構成
LINUX用カーネル
プロセス管理
メモリ管理
ネットワーク通信
ファイルシステム
デバイス管理
GUNツール、クラスライブラリ
デスクトップ環境
パッケージ・マネジメント・システ
GCCコンパイラ
様々なアプリケーション

Linux システムの構成

人々がディストリビューションを作成する際、通常は特定の目的のためであり、構築の哲学は異なり、重点も異なるため、Linux ディストリビューションは多種多様です。その中でも Debian、Suse、Fedora 系のディストリビューションが最も一般的です。基礎的なユーザーにとっては、特定のソフトウェアをインストールする際に、異なる Linux 系統に対して異なるインストール命令が提供されることに混乱することがよくあります。これは実際には、異なる Linux 系統が異なるパッケージ管理ソフトウェア（異なるソフトウェアストアを使用すると理解できる）を使用しているためですが、これは大した問題ではありませ

株式会社日昇テクノロジー

ん。一般的な Linux ディストリビューションの関係を示す図が以下にあります。



一般的な Linux ディストリビューションの関係図

Linux に初めて触れるユーザーには、これらがすべて Linux システムであり、現在開発ホストには Ubuntu を使用していることを理解しておくだけで十分です。開発ボード上で動作する Linux システムに関しては、これを特定のボードに対してカスタマイズされた Linux ディストリビューションと理解することができます。

3.2.1 Debian 系

Debian 系には、Debian、Ubuntu などのディストリビューションが主に含まれます。

Debian の特徴は、膨大なソフトウェアのサポートと apt-get パッケージ管理システムにあり、さらにサポートするハードウェアプラットフォームが非常に幅広く、x86、AMD、MIPS、ARM、POWERPC などが含まれます。

Ubuntu は Debian から派生しており、その使いやすさから、最も成功した Linux デスクトップ版と言えるでしょう。その成功はサーバーバージョンにも広がり始めており、現在は物联網などの小型デバイス

株式会社日昇テクノロジー

向けに Ubuntu Core バージョンもリリースされており、大きな発展が期待されます。本書では、開発ホストのシステム環境として Ubuntu を使用します。

Debian と Ubuntu の公式ウェブサイトには非常に豊富な使用チュートリアルがあり、中文のサポートも良好で、初心者には特に Debian の内容を多く閲覧することをお勧めします。

Debian 公式ウェブサイト：https://www.debian.org

Ubuntu 公式ウェブサイト：https://ubuntu.com

3.2.2 Fedora 系

Fedora 系には、Fedora、Red Hat Linux、Red Hat Enterprise Linux（RHEL）、CentOS などのディストリビューションが含まれます。

Red Hat Linux は、Red Hat（レッドハット）社によってリリースされた個人向けの Linux バージョンで、現在は開発が停止されています。代わりに、主にサーバー領域で使用される Red Hat Enterprise Linux（RHEL）、すなわちレッドハット企業版 Linux の開発に注力しています。RHEL を使用する利点は、安全で安定した技術サポートと企業レベルの保証を受けることができる点で、これは多くのサーバーアプリケーションシーンの核心的な要求です。レッドハット社は、これらのサービスを提供することで、最も利益を上げる Linux 企業となりました。現在レッドハット社は IBM に買収されています。

Fedora ディストリビューションは、RHEL を基にコミュニティによって構築され、レッドハット社のスポンサーを受けています。それは最新の技術を大胆に採用し検証し、一部の検証済み技術は RHEL に追加されます。そのため、Fedora と RHEL は互恵的な関係にあります。別の観点から見ると、Fedora は RHEL のテストバージョンとも考えられています。

CentOS の正式名称は Community Enterprise Operating System、つまりコミュニティ企業向けオペレーティングシステムです。それはレッドハットが開源プロトコルに従って公開した RHEL のソースコードを基に構築されています。CentOS と RHEL の違いは、CentOS が無料の長期アップグレードと更新サービ

株式会社日昇テクノロジー

スを提供し、レッドハットの技術サポートを求めずに RHEL を使用することに相当する点です。

3.2.3 OpenSUSE 系

SUSE 系には、SUSE、SUSE Linux Enterprise Server（SLES）、openSUSE などのディストリビューションが含まれており、その関係は Fedora、Red Hat Enterprise Linux（RHEL）、CentOS の関係に似ています。RedHat が x86 アーキテクチャのコンピュータに特化しているのに対して、SUSE は誕生当初からメインフレームをターゲットにしていたため、大型サーバー領域で一定の地位を占めています。

第 4 章 デスクトップ Linux を体験する

4.1 序文

この章の内容を体験したい場合、以下の材料を準備する必要があります：

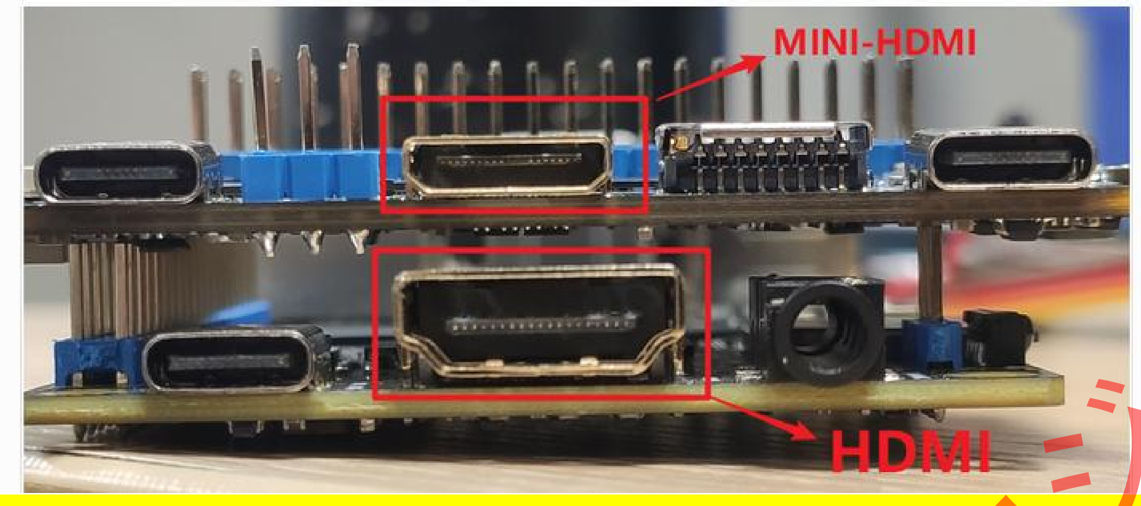
4.1.1 HDMI インターフェーススクリーンの使用

1. HDMI インターフェースを持つ画面（または、VGA から HDMI へ、DP から HDMI への変換ケーブルを使用可能）

2. ボードに対応する HDMI ケーブル、一部のボードは MINI-HDMI を使用し、他は標準の HDMI を使用するため、ケーブルのインターフェースに応じて選択する必要があります。

HDMI 種類	ボード型番
MINI-HDMI	LubanCat- 4
標準 HDMI	

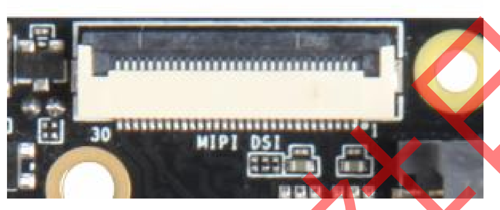
株式会社日昇テクノロジー



（注意：HDMI と MINI-HDMI のインターフェースのサイズと形状は大きく異なるため、直接接続することはできません。HDMI と MINI-HDMI を接続して使用する場合は、HDMI から MINI-HDMI への変換アダプタを購入する必要があります。）

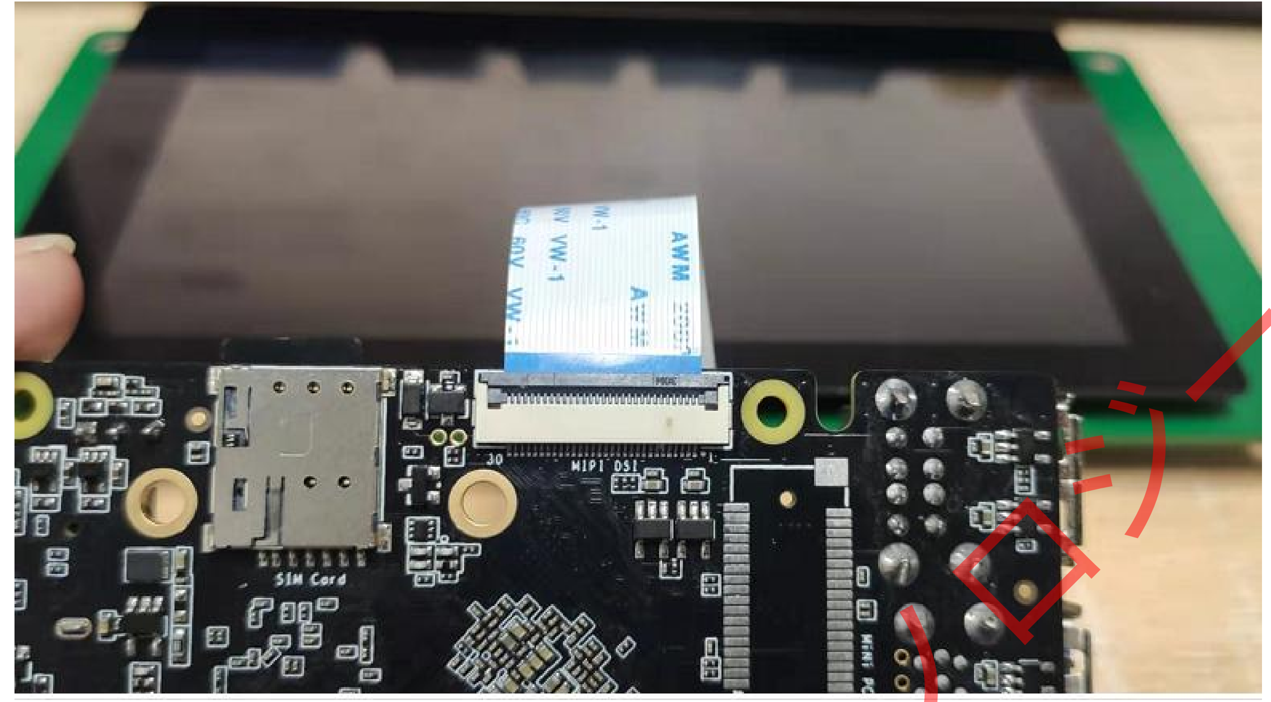
4.1.2 MIPI スクリーンの使用

MIPI インターフェースは 30 ピンの fpc ソケットを使用しています。下図のように



mipi スクリーンとの接続は下図のようになります：

株式会社日昇テクノロジー



デフォルトでは HDMI が開いており、MIPI はデフォルトで閉じています。正常に表示するにはスクリーンの切り替えが必要です。切り替えについては、クイックスタートガイドのスクリーン切り替えセクションを参照してください。

注意：mipi-dsi インターフェースはホットプラグをサポートしていません。電源が入っている状態でスクリーンを取り付けたり取り外したりしないでください。電源が入っている状態での取り付けや取り外しは、ボードがショートする可能性があります。軽度な場合はボードが保護シャットダウンを行い、重度な場合はチップのインターフェースが壊れたりチップが焼けたりする可能性があります。

4.2 電源を入れる前の準備

4.2.1 外部デバイスの接続

- 電源を入れる前に必ず接続する必要がある：SD カード（SD カードを使ってログインする場合）

- ホットプラグ可能：USB マウス、USB キーボード、HDMI スクリーン

一部のデバイスはホットプラグに対応していますが、電源を入れる前に先に接続することをお勧めします。

株式会社日昇テクノロジー

4.2.2 電源供給

ボードのほとんどは 5V Type-C 供給をサポートしていますが、一部は DC12V または DC5V 供給を使用しています。したがって、ボードのモデルに応じて適切な電源を選択する必要があります。

リスト 2: ボード電源

電力供給カテゴリー	ボード
Type-C(5V@4A)	LubanCat-4
DC5V(5V@4A)	LubanCat-4

ボードに対応する電源を用意した後、正しいインターフェースで接続を選択する必要があります。

- 一部のボードには 2 つの Type-C ポートがあります。Type-C ポートの近くのシルク印刷を確認する必要があります。もし「pwr」または「OTG」と書かれている場合、そのインターフェースを使用して電源を入れることができます。

- 電源を接続したら、電源を入れることができます。

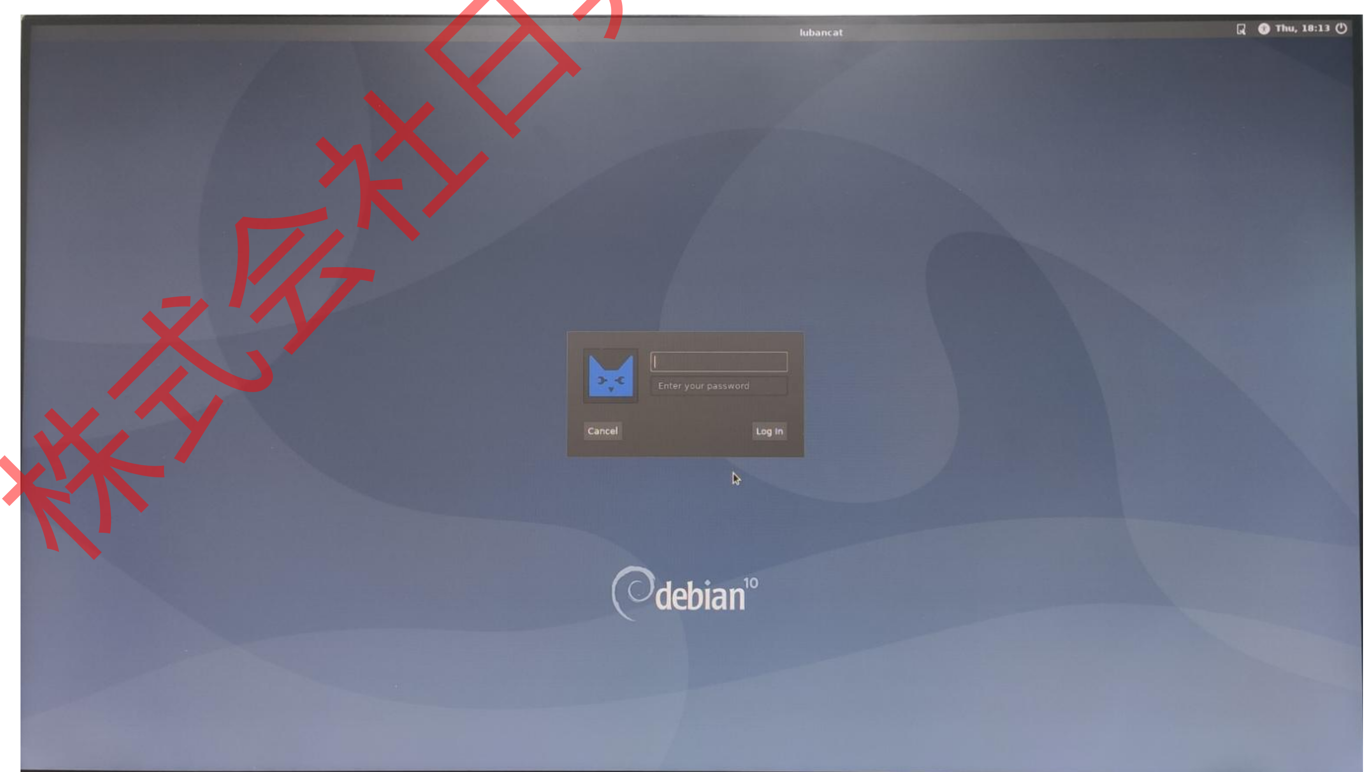
4.3 電源オン

電源を入れると、画面の真ん中に LubanCat のロゴが表示されます。しばらく待っていると、マウスカーソルとロックスクリーンの全体的なインターフェースがロードされます。

4.3.1 ロックスクリーンインターフェース

ロックスクリーンインターフェースでは以下のようなものが見られます。

Debian ロックスクリーンインターフェース



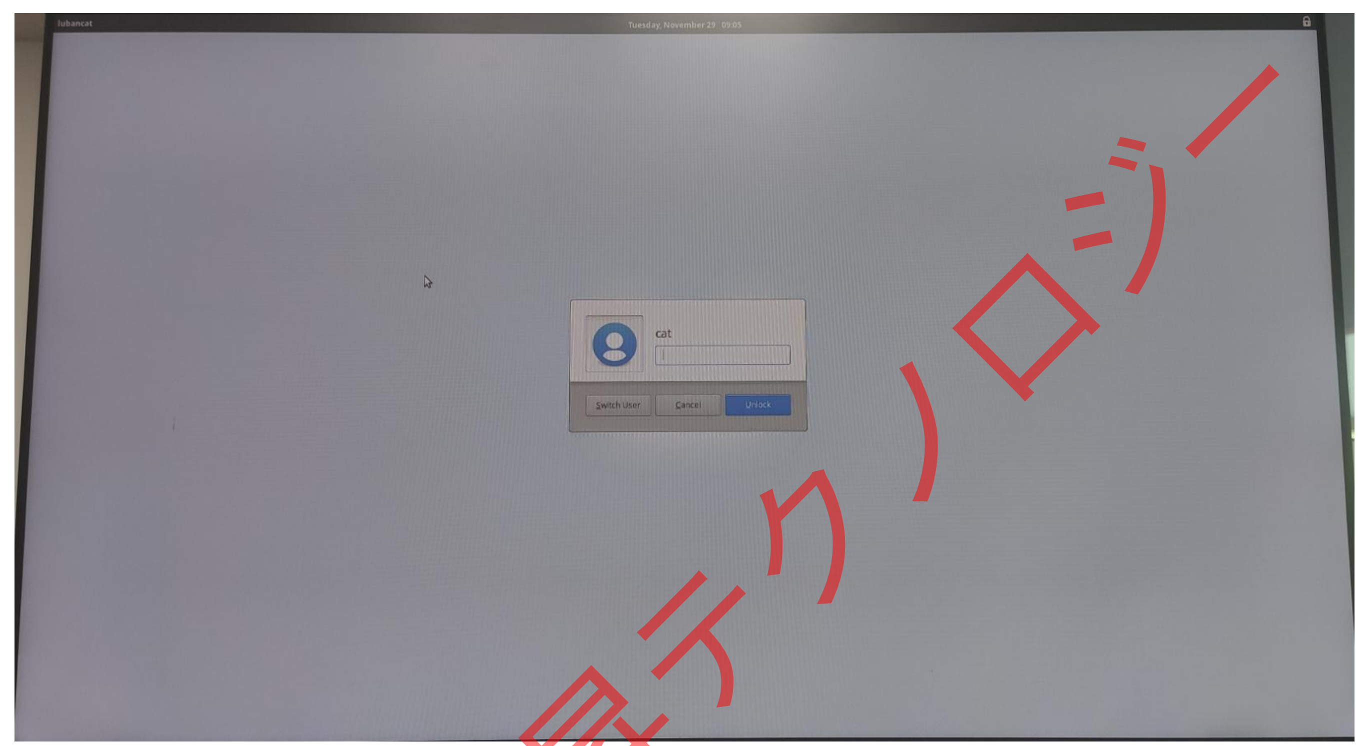
- ここでキーボードを使ってユーザー名「cat」を入力する必要があります。

株式会社日昇テクノロジー

- ここでキーボードを使ってパスワード「temppwd」を入力する必要があります。

- その後、エンターキーを押してデスクトップに入ります。

Ubuntu ロックスクリーンインターフェース



- ここでキーボードを使ってパスワード「temppwd」を入力する必要があります。

- その後、エンターキーを押してデスクトップに入ります。

4.3.2 デスクトップに入る

デスクトップに入った後、デスクトップの全体的なレイアウトが Windows のデスクトップと似ていることがわかりますが、違う点はステータスバーがデスクトップの上部にあることです。

Debian メインインターフェース

クイックツールバー

株式会社日昇テクノロジー

システムマネージャ
ワークスペース
ブルートゥース
ネットワークマネージャ
デバイス制御
クイックツールバ

Debian メインインターフェース

システムマネージャー
通
電源管
ネットワークマネージャ
画面の明るさ調整
サウンドサイズ

株式会社日昇テクノロジー

4.3.3 デスクトップの自動ログイン

ファイル /etc/lightdm/lightdm.conf を開くか新規作成し、以下の内容を修正または追加します。

```
[Seat:*]

autologin-guest=false

autologin-user=cat

autologin-user-timeout=0
```

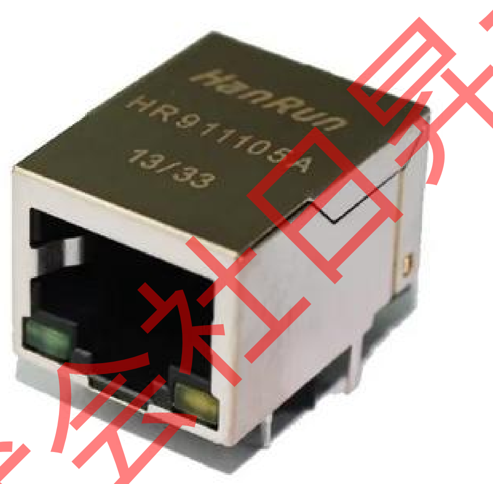
- autologin-user：自動ログインするアカウント名

- autologin-user-timeout：自動ログインまでの時間、単位は秒

4.4 ネットワーク接続

4.4.1 有線接続

有線接続を使用する場合は、以下のインターフェイスを選択し、ケーブルの形状に合わせて該当するインターフェースに接続する必要があります。



注意：一般的に、ルーターやスイッチから引き出されたネットワークケーブルを直接接続するだけでネットワークに接続できます。しかし、学校のネットワークなどの場合は、ブラウザを開いてログイン認証を行う必要があります。

4.4.2 Wi-Fi 接続

Ubuntu と Debian システムのネットワーク設定方法はほぼ同じなので、ここでは Debian システムの

株式会社日昇テクノロジー

Wi-Fi 設定を示します。

- 第一歩：右上隅のネットワーク管理をクリックします。

- 第二歩：接続したい Wi-Fi を選択します。

- 第三歩：Wi-Fi のパスワードを入力します。

- 第四歩：接続をクリックします。

株式会社日昇テクノロジー

• 以下は接続が成功した画像です。

4.4.3 その他の接続

ネットワークへの接続方法はさまざまです。詳細な方法については、以下のセクションをご覧ください。

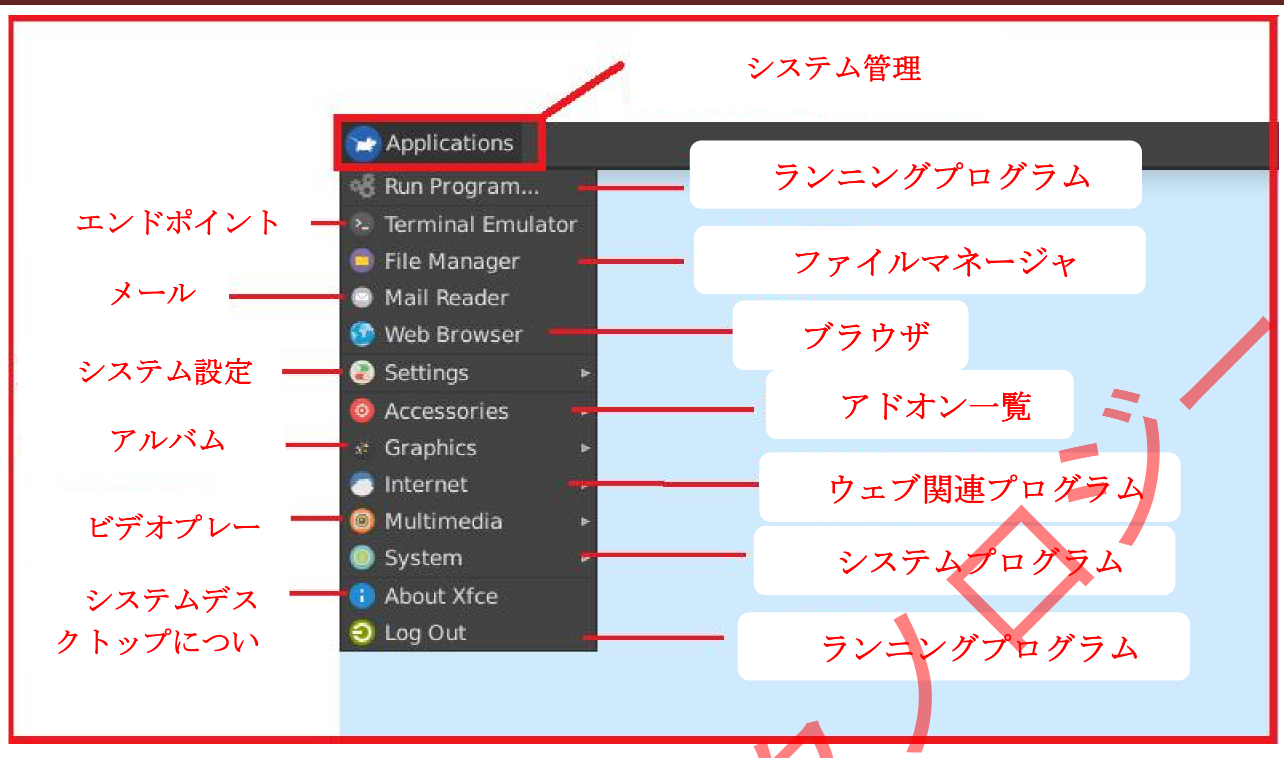
"ネットワーク接続および静的 ip 設定"

4.5 デスクトップの機能の簡単な紹介

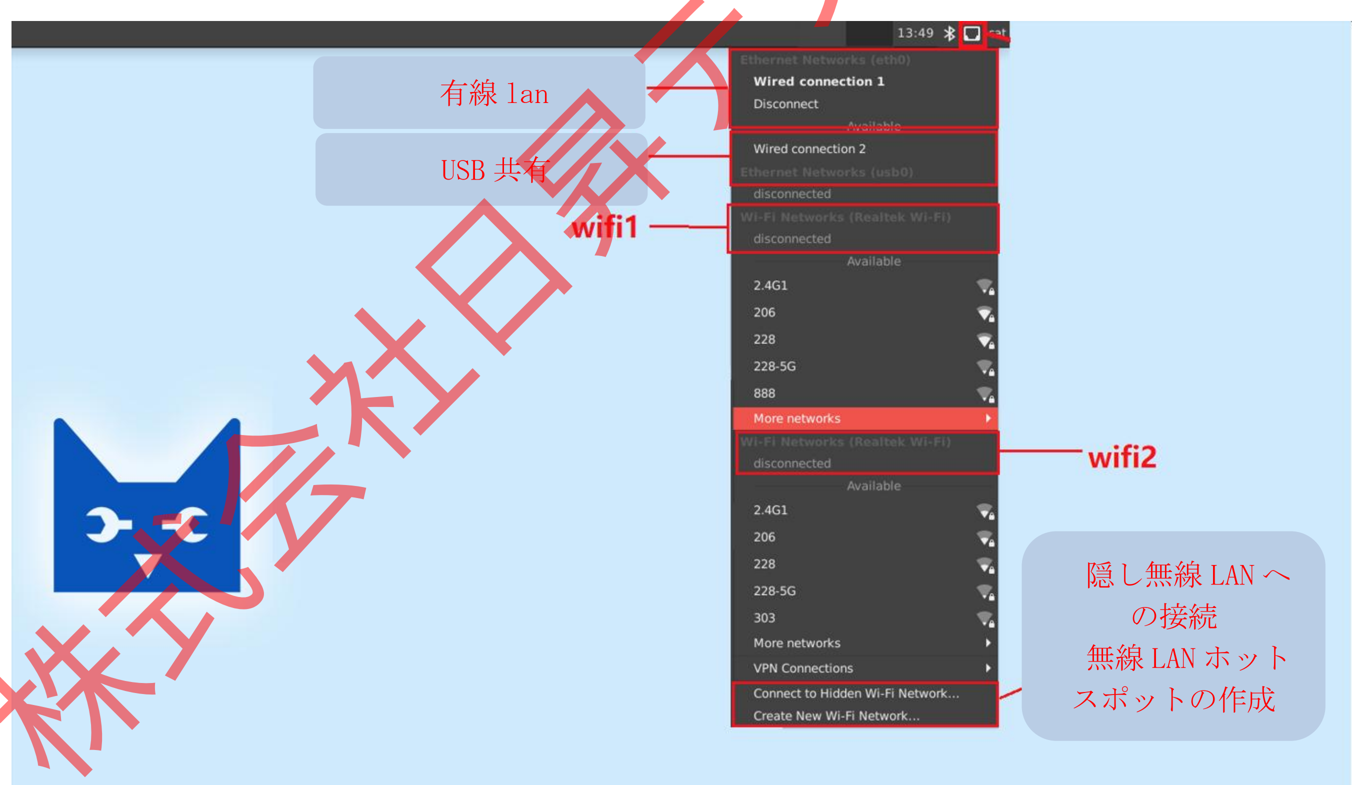
Debian インターフェース

• システムマネージャー

株式会社日昇テクノロジー



• ネットワークマネージャー



• 壁紙の変更

株式会社日昇テクノロジー

右クリックでメニューを表示

Applications を選択

Setting を選択

Desktop を選択

• デスクトップ設定に入る"

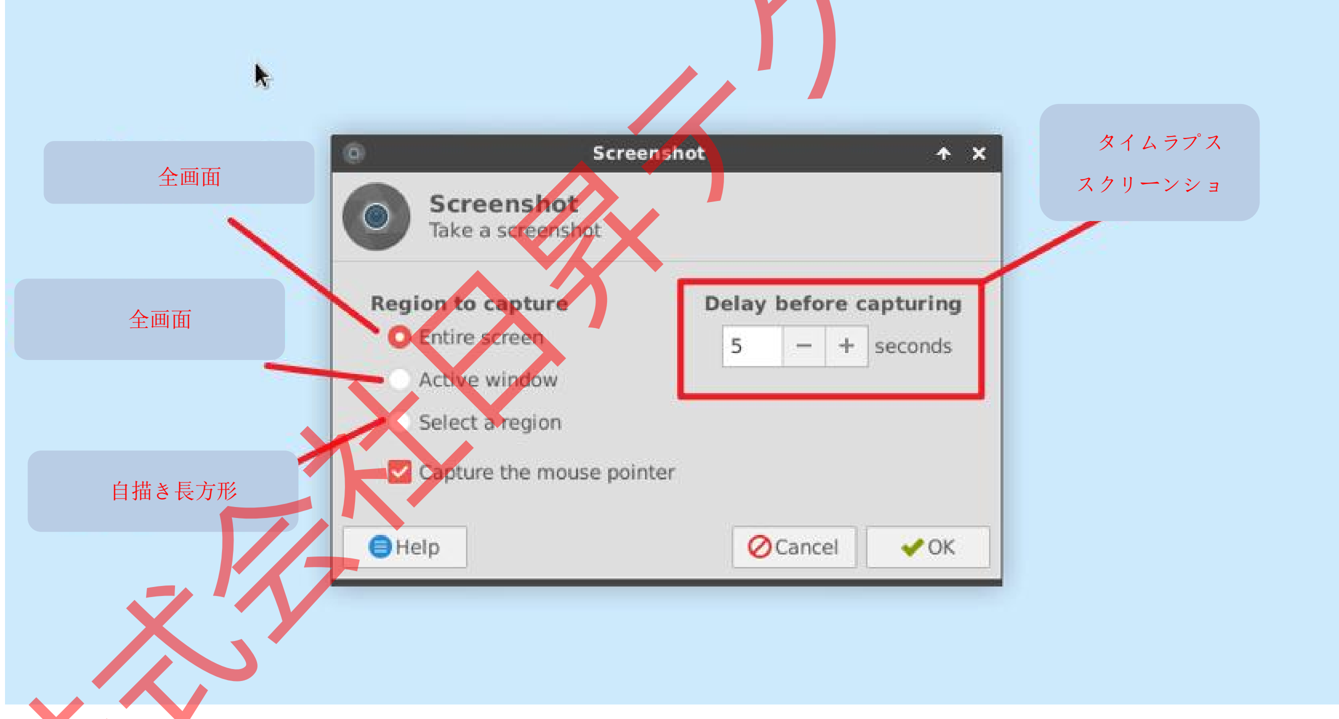
背景画像を追加するフォルダ

壁紙の画像を選択

・ スクリーンショット

株式会社日昇テクノロジー

自分自身のスクリーンショット規則を設定できます



Ubuntu インターフェース

システムマネージャー

株式会社日昇テクノロジー

システム・マネージャーット
クリックして入る

お気に入り

最近の使用

すべてのソフトウェア

アドオンソフトウェア

マルチメディア・アプリケーショ

設定

システム設定

Favorites
Recently Used
All Applications
Accessories
Multimedia
Settings
System

Web Browser
Mail Reader
File Manager
Terminal Emulator
Help

cat

ソフトウェア検索

ネットワークマネージャー

有線 lan0

有線 lan1

USB 共有

Wifi1

Wifi2

隠し無線 LAN への接続
無線 LAN ホットスポットの作成

ネットワークを有効にする
wifi を有効にする

接続情報の表示
接続情報を編集する

Ethernet Network (eth0)
Wired connection 1
Disconnect
Ethernet Network (eth1)
disconnected
Ethernet Network (usb0)
device not managed
disconnected
206
228
228-5G
A1
aEBF_Guest
More networks
disconnected
206
228
228-5G
A1
aEBF_Guest
More networks
Connect to Hidden Wi-Fi Network...
Create New Wi-Fi Network...
VPN Connections
Enable Networking
Enable Wi-Fi
Connection Information
Edit Connections...

背景画像の変更

株式会社日昇テクノロジー

デスクトップ設定に入る

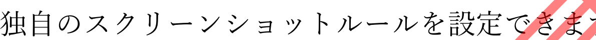
スクリーンショットを撮る

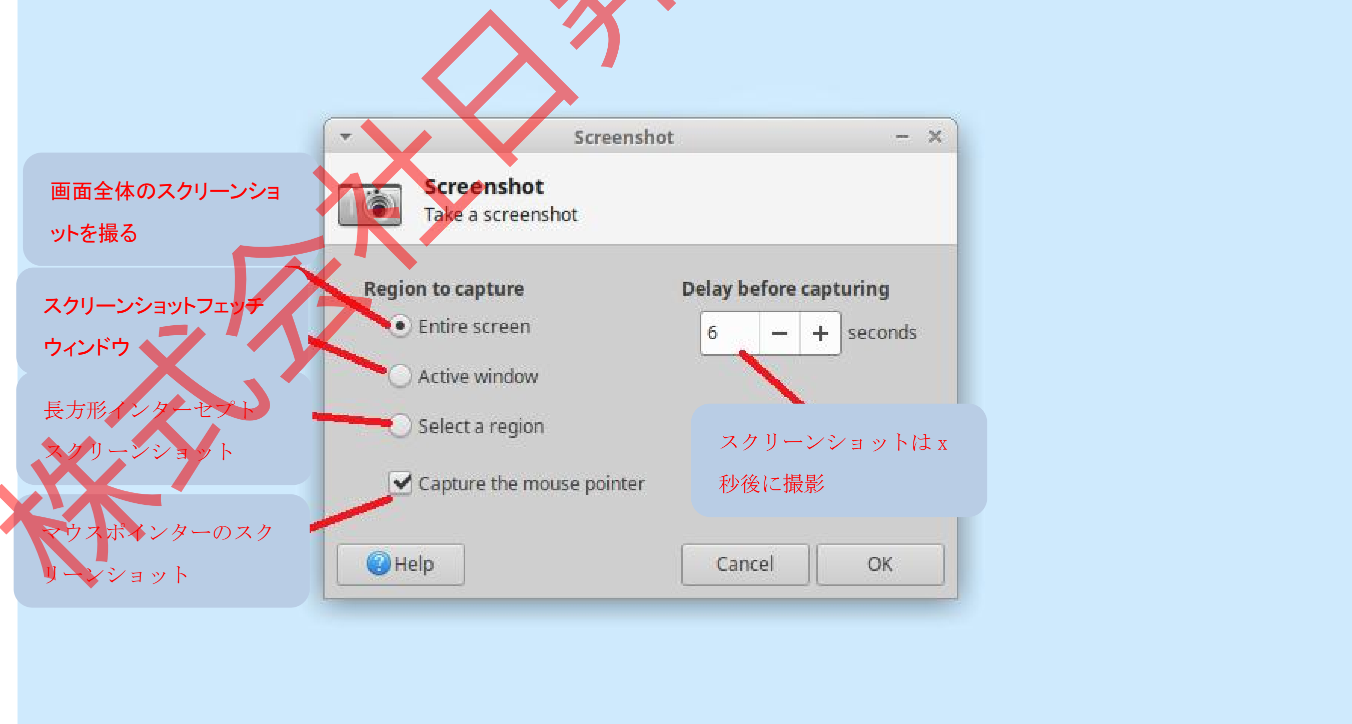
キーボードの PrtSC を押す

株式会社日昇テクノロジー

1. Accessories をクリック

2. Screenshot をクリック





株式会社日昇テクノロジー

第 5 章 ターミナル Linux（SSH）を体験する

注意：この章の内容は、ローカルネットワーク内に LubanCat ボードが 1 つしかない場合にのみ適用されます。複数のボードがある場合、この方法は適用されません。

注意：コンピューターの USB 電源を接続してしばらくすると、緑色の LED が点滅し続ける場合、電力供給が十分であることを示します。電力供給が不足している場合、ボードが繰り返し再起動する可能性があります（緑色の LED が非常に暗いか、2 秒以内に連続して点滅しない場合があります）。

5.1 前言

この章の内容を体験したい場合、以下の条件が必要です。

1. イメージファイル：eMMC を搭載したボードは工場出荷前にイメージが書き込まれています。SD カードを使用するか、デスクトップイメージが含まれていない場合は、イメージを書き込む必要があります。イメージの書き込み方法については、「イメージの書き込み」セクションを参照してください。

2. 通信ソフトウェア：MobaXterm の使用をお勧めします。

3. コンピューターが RNDIS 機能をサポートしている必要があります（Windows 10 はデフォルトでサポートしていますが、他のオペレーティングシステムでは RNDIS ドライバーのインストールが必要かもしれません）。

株式会社日昇テクノロジー

4. 通信機能を備えた Type-C ケーブル

5. コンピューターの電源供給ポートが 5V @ 1A 以上である必要があり、コンピューターのマザーボードに直接接続することをお勧めします。HUB のポートは電力供給が不足する可能性があります。

条件を満たしていない場合、この章をスキップできます。

注意：コンピューターの電源供給能力が不足している場合、ボードが繰り返し再起動する可能性があり、システムにアクセスできなくなる可能性があります。再起動が継続する場合は、外部デバイスの使用を減らすことができます。

正常な起動を確認する方法：

1. 緑色の LED がハートビートのように点滅します - 正常

2. 緑色の LED が非常に暗いか、2 秒以内に連続して点滅しない場合 - 再起動

5.2 開始前の準備

5.2.1 ソフトウェアの準備

MobaXterm のインストールと使用方法

• ダウンロードリンク：https://mobaxterm.mobatek.net/download.html

MobaXterm 端末ソフトウェアの詳細な使用方法は、「Linux 基本とアプリケーション開発実践ガイド - i.MX6ULL をベースにして」に記載されています。

i.MX6ULL」を参照してください。 パラメータ設定の違いを除けば、使い方は同じです。

5.2.2 電源投入

LubanCat-4 または LubanCat-OTG を使用している場合は、Type-C ケーブルを使用して、ボード上の pwr または OTG と書かれた Type-C コネクタに接続します。

LubanCat-4 または LubanCat-4N を使用している場合は、DC コネクタを使用して電源を供給することができます。

株式会社日昇テクノロジー

リスト 1: ボード給電

Type-C(5V@4A)	LubanCat-4
DC5V(5V@4A)	LubanCat-4

準備ができたら、ボードの電源を入れます（USB または DC オスコネクタを差し込みます）。

5.3 ボードのスイッチオン

注意: このセクションでは、ボードを使用する前にボードの電源を正常に入れる必要があります。

- バーンイン後、初めてボードを起動するときは、ボードのステータス・ランプが点灯するまで約 1 分間待つ必要があります。

- 初回起動時でない場合は、20 秒待つだけでよい。

1. 緑のランプが心臓の鼓動のように点滅している。
2. 緑のランプが非常に暗いか、2 秒以内に点滅し続けない場合 → 再起動している可能性があります（お使いのコンピュータの USB ポートが大電流供給に適していない可能性があります。）

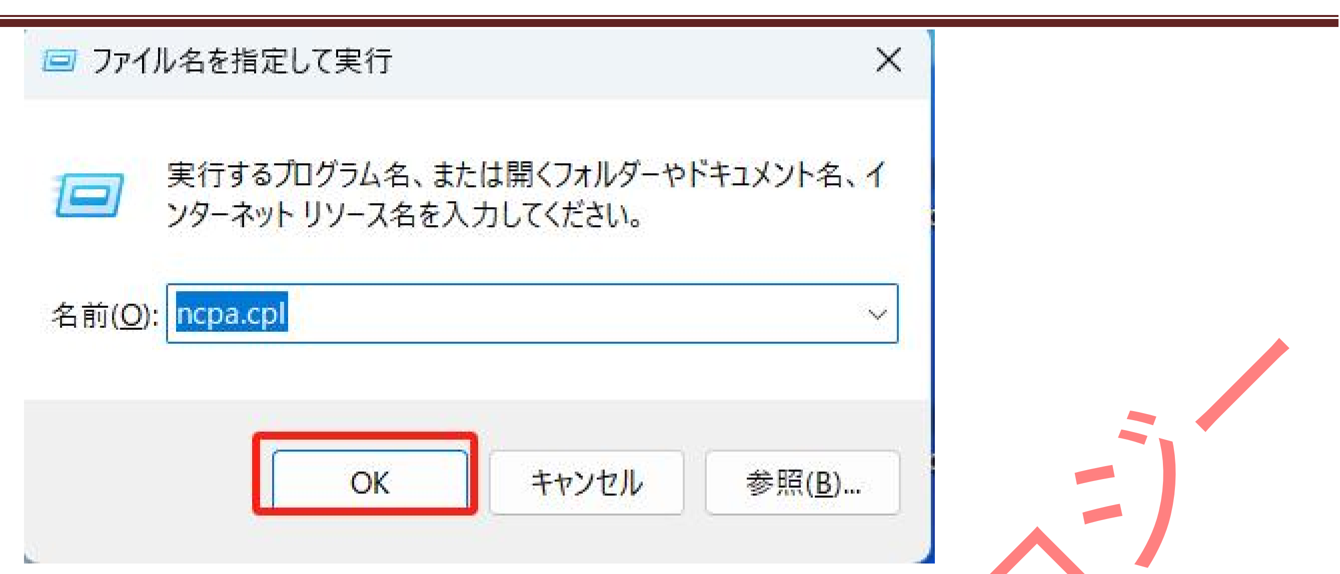
5.3.1 コンピュータネットワークの設定

注意: コンピューターは RNDIS 機能をサポートする必要があります(Win11 はデフォルトでサポートしています。他のシステムでは RNDIS ドライバーをインストールする必要があるかもしれません)。

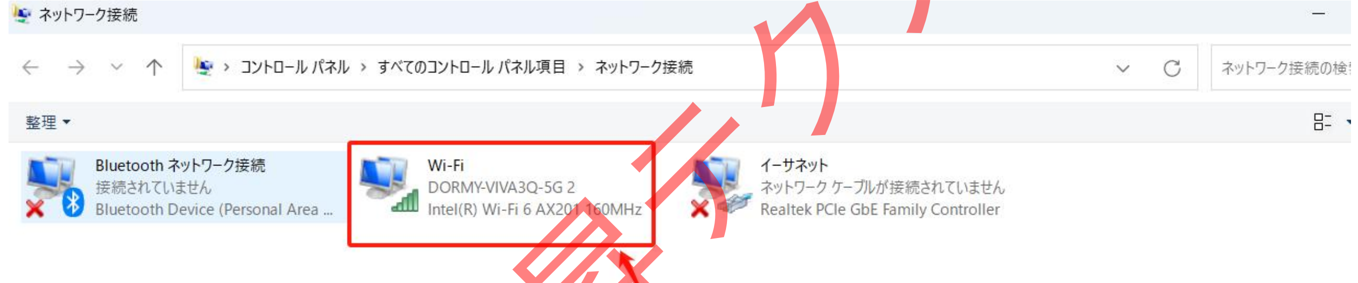
- 書き込み後、初めてボードを起動するときは、初期化操作が完了するまで約 1 分間待つ必要があります。

- キーボードの R+WIN キーを一緒に押して下のようなポップアップが表示されて ncpa.cpl を入力して ok をクリックします

株式会社日昇テクノロジー



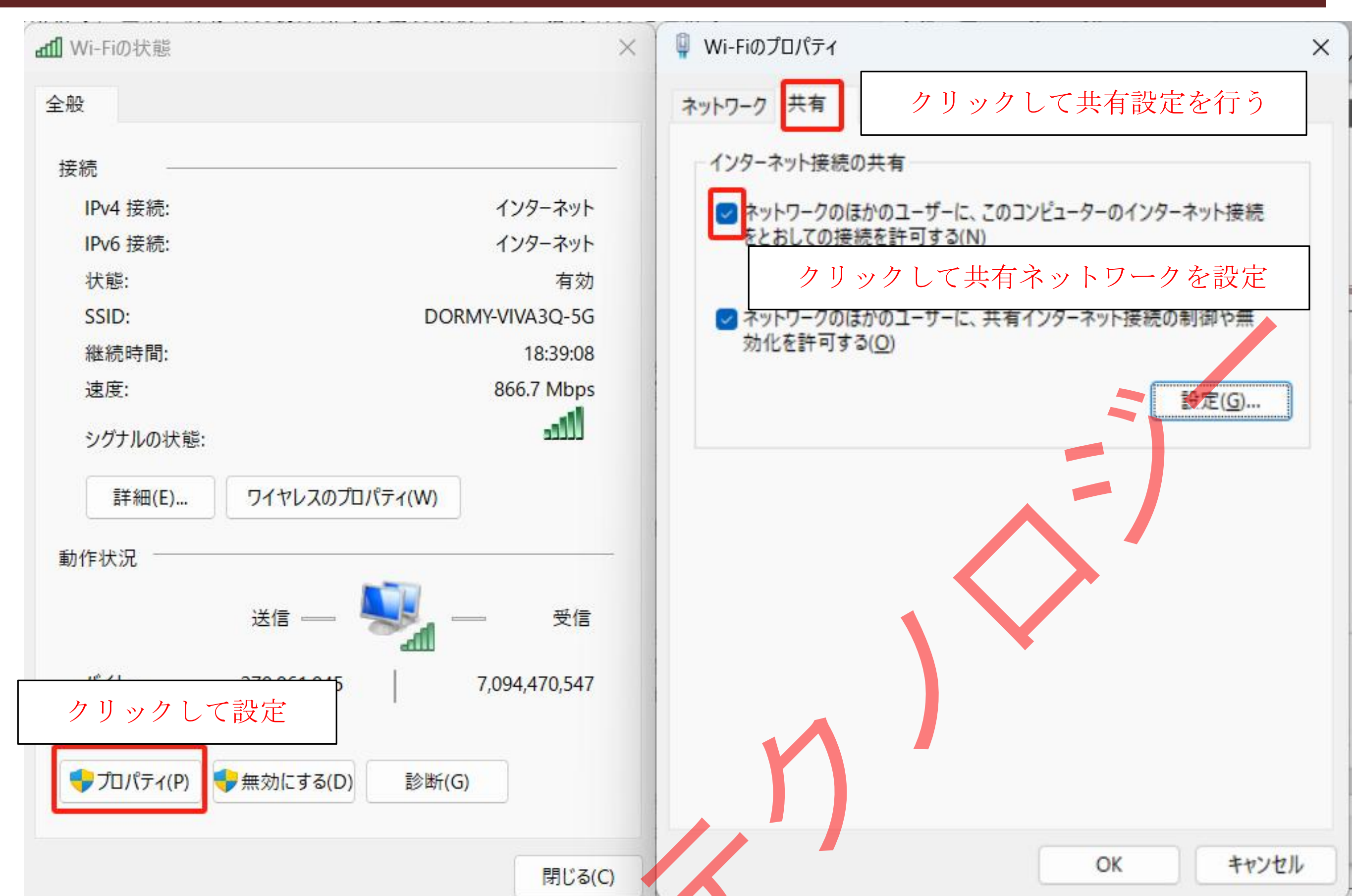
- 現在のネットワーク接続をすべて表示する（"Remote NDIS based Internet "と書かれたカードがカードです）。



現在使用中のネットワーク、ダブルクリックして設定します

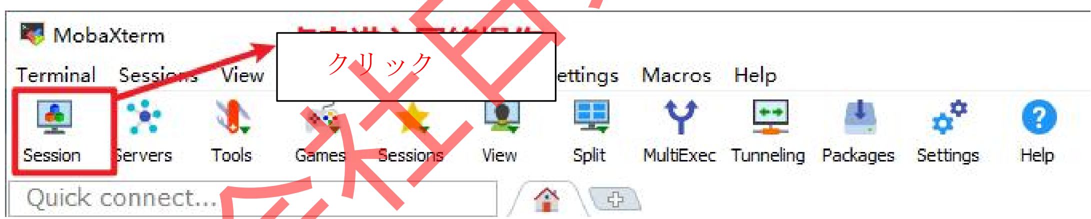
- ネットワーク共有の設定

株式会社日昇テクノロジー



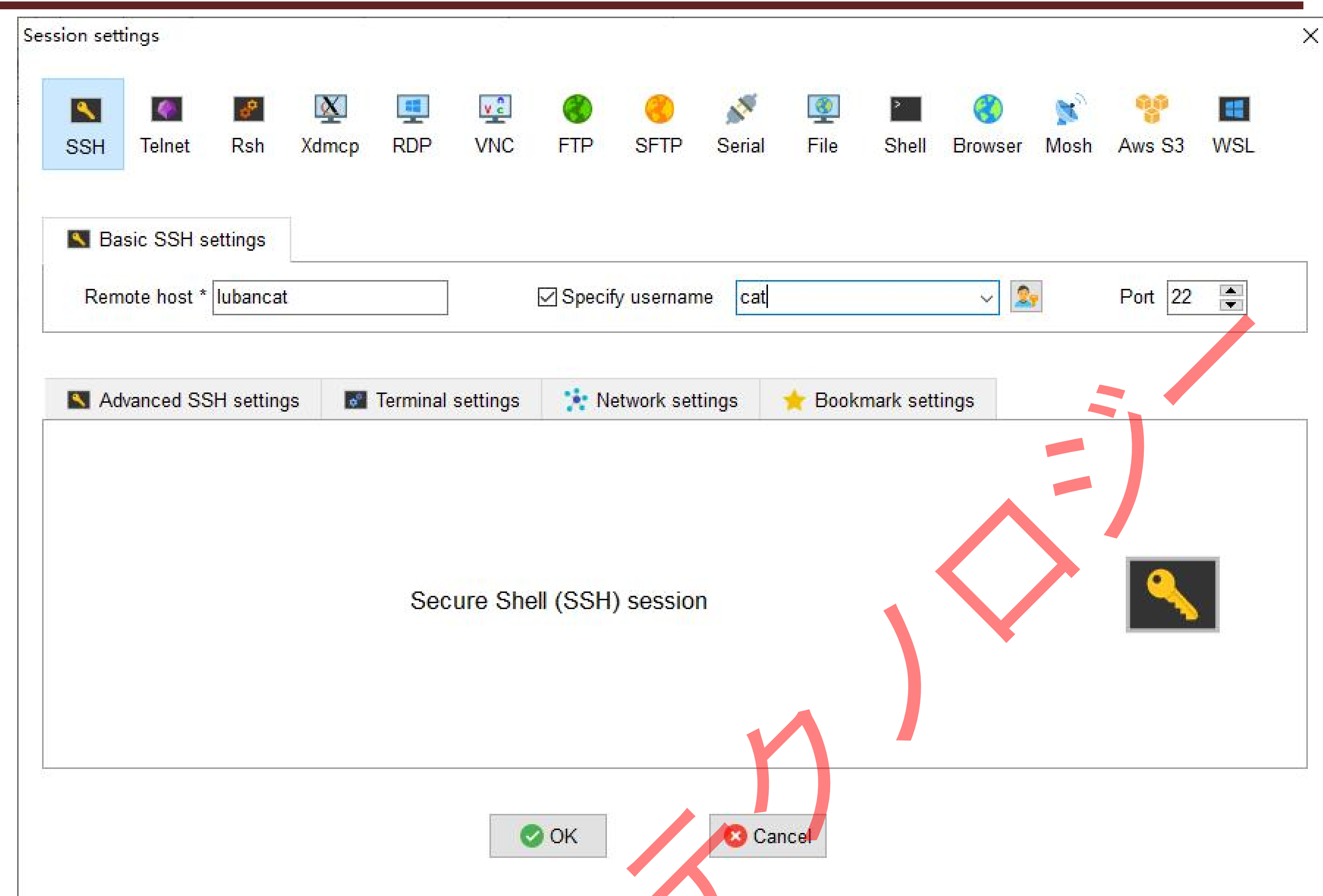
- それができたら、ボードに接続することができます。

MobaXterm ソフトウェアを開き、「セッション」をクリックして接続を作成します。



- 次に、以下のコンフィギュレーションを入力する。

株式会社日昇テクノロジー



- OK "をクリックして設定を完了する。

- 次に、ターミナルにアクセスするためにパスワード：temppwd を入力します（パスワードは平文では表示されませんので、入力が終わったら Enter キーを押してください）。

5.4 ネットワーク接続

この時点で、すでにボード上にネットワークがあり、ネットワークポートや無線 LAN に接続する必要はありません。

5.5 機能紹介

5.5.1 簡単なコマンドラインの使い方

5.5.1.1 ディレクトリ内のファイルの一覧表示

株式会社日昇テクノロジー

カレントディレクトリのファイルをリストする

ls

他のディレクトリのファイルのリスト

ls + 他のディレクトリ

```
root@lubancat:/home# ls
cat
root@lubancat:/home# ls /boot
Image-4.19.232      boot_init       dtb         logo.bmp     rk-kernel.dtb
System.map-4.19.232 config-4.19.232 kerneldeb   lost+found   uEnv
root@lubancat:/home#
```

5.5.1.2 ディレクトリの場所

カレントディレクトリの場所をリストする

Pwd

```
root@lubancat:/home#
root@lubancat:/home# pwd
/home
root@lubancat:/home#
```

5.5.1.3 ディレクトリの切り替え

ディレクトリの切り替え

cd + ディレクトリの場所

最後に切り替えたディレクトリを切り替える

cd -

ホームディレクトリを切り替える

cd

株式会社日昇テクノロジー

```
cd ~

# 前のディレクトリに切り替える

cd ..
```

```
root@lubancat:~# cd /boot/
root@lubancat:/boot# pwd
/boot
root@lubancat:/boot# ls
Image-4.19.232       boot_init         dtb         logo.bmp    rk-kernel.dtb
System.map-4.19.232  config-4.19.232   kerneldeb   lost+found  uEnv
root@lubancat:/boot#
```

5.6 ソフトウェアのアップデート

Debian と Ubuntu のソフトウェアのほとんどはインターネット上で入手可能で、自動的にアップデートされます。

image ファイルは古いバージョンのソフトウェアを使用している可能性があり、そのソフトウェアにはバグがある可能性があり、使用に影響を与えるかもしれません。ですから適切な時間に

で自分でソフトウェアを更新することができます。

注意: イメージを書き込んだ後の始めての起動の後、ネットワークを繋げてからソフトウェアを更新した後で、ほかのソフトウェアを更新できるになります。

```
# ソフトウェアのアップデート

sudo apt update


# アップデートしたソフトウェアをインストールする

# sudo apt upgrade
```

株式会社日昇テクノロジー

5.6.1 ソフトウェアのインストールとアンインストール

5.6.1.1 ソフトウェアのインストール

```
sudo apt install xxx
```

ここでは、tree ソフトウェアを例にとって説明します。tree は、フォルダを表示するためのツリー構造です。

```
# ソフトウェアのアップデート

sudo apt install tree



 # フォルダーを作成する

# mkdir a



# フォルダーを切り替える

  cd a



# 複数のファイルを作成する

touch 1 2 3

# 前のディレクトリに切り替える

cd ...

# b というフォルダをコピーする

cp -r a b

# ツリーを使ってフォルダを表示する

tree
```

株式会社日昇テクノロジー

以下のように

```
root@lubancat:~# ls
root@lubancat:~# mkdir a
root@lubancat:~# cd a
root@lubancat:~/a# touch 1 2 3
root@lubancat:~/a# cd ..
root@lubancat:~# cp -r a b
root@lubancat:~# tree
.
├── a
│   ├── 1
│   ├── 2
│   └── 3
└── b
    ├── 1
    ├── 2
    └── 3

2 directories, 6 files
root@lubancat:~#
```

5.6.1.2 ソフトウェアのアンインストール

```
sudo apt remove xxx
```

tree を例にとって説明しよう。

```
sudo apt remove tree
```

アンインストールすると、tree コマンドは使えなくなります

第 6 章 ターミナル Linux の体験（シリアル接続）

6.1 はじめに

この章の内容を体験したい場合は、次の条件が必要です。

1.image ファイル、工場出荷時に emmc 付きのボードで良い書き込みしています、sd カードを使用する場所、或いはデスクトップバージョンの image ファイルを書き込み必要がな場合は、 image 書き込みの部分 を参考してください。

2.電源、ボードが正常に動作するために十分な電力を供給する必要があり、ここで公式の電源アダプタをお勧めします。

株式会社日昇テクノロジー

3.シリアルモジュール、ここでは公式シリアルモジュールと CH340 をお勧めします。

4. 通信ソフト、MobaXterm を推奨します。

6.2 電源投入前の準備

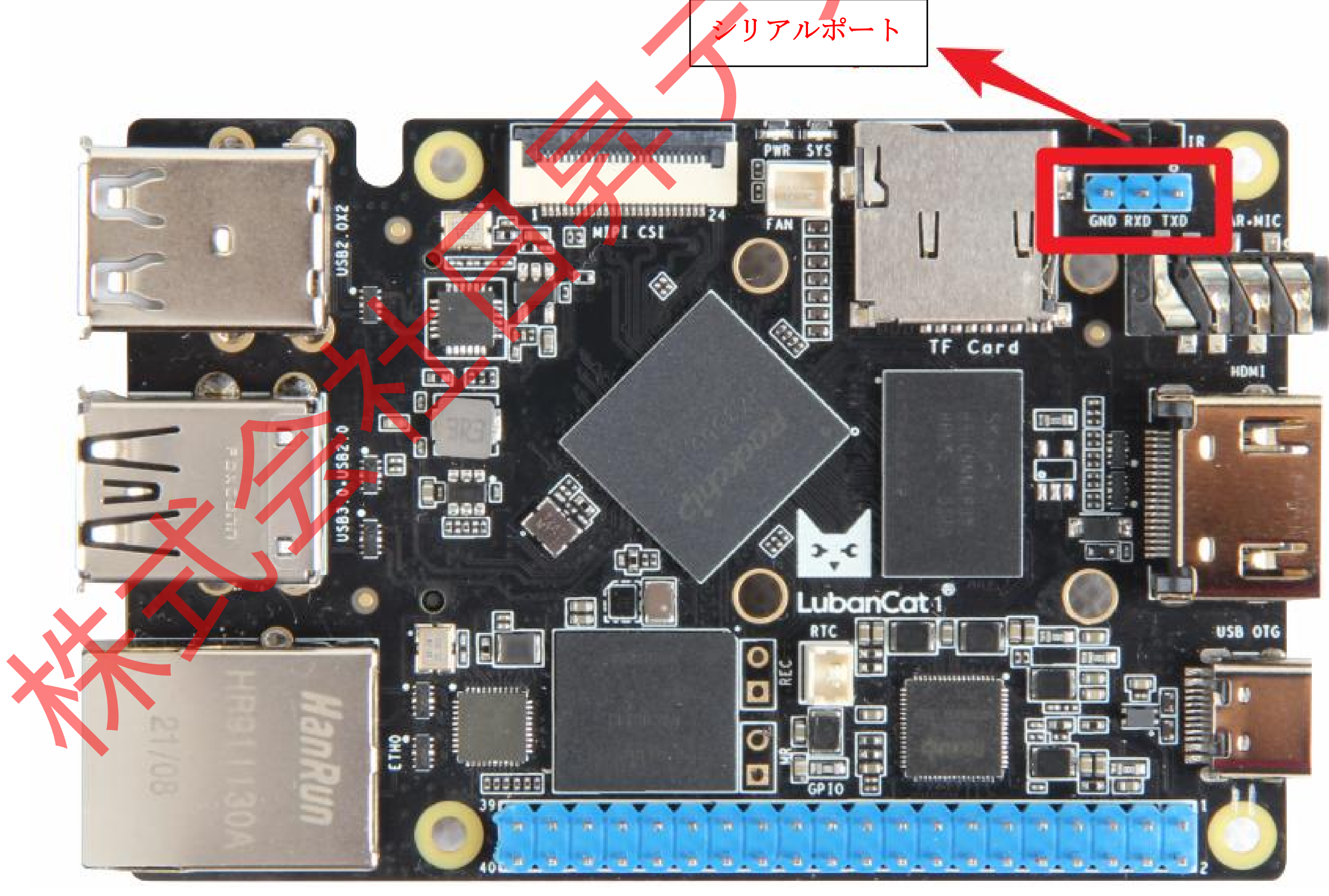
6.2.1 周辺機器の接続

デバイスの電源を入れる前に、シリアルモジュールと SD カード（ログインに SD カードを使用する場合）を接続する必要があります。

6.2.1.1 シリアル接続

LubanCat ボードには、2 種類のシリアル・ポート接続がある。

- 一つは 3 ピン・コネクターで、デュポン・ケーブルを使って接続することができる。



リスト1：ピン接続

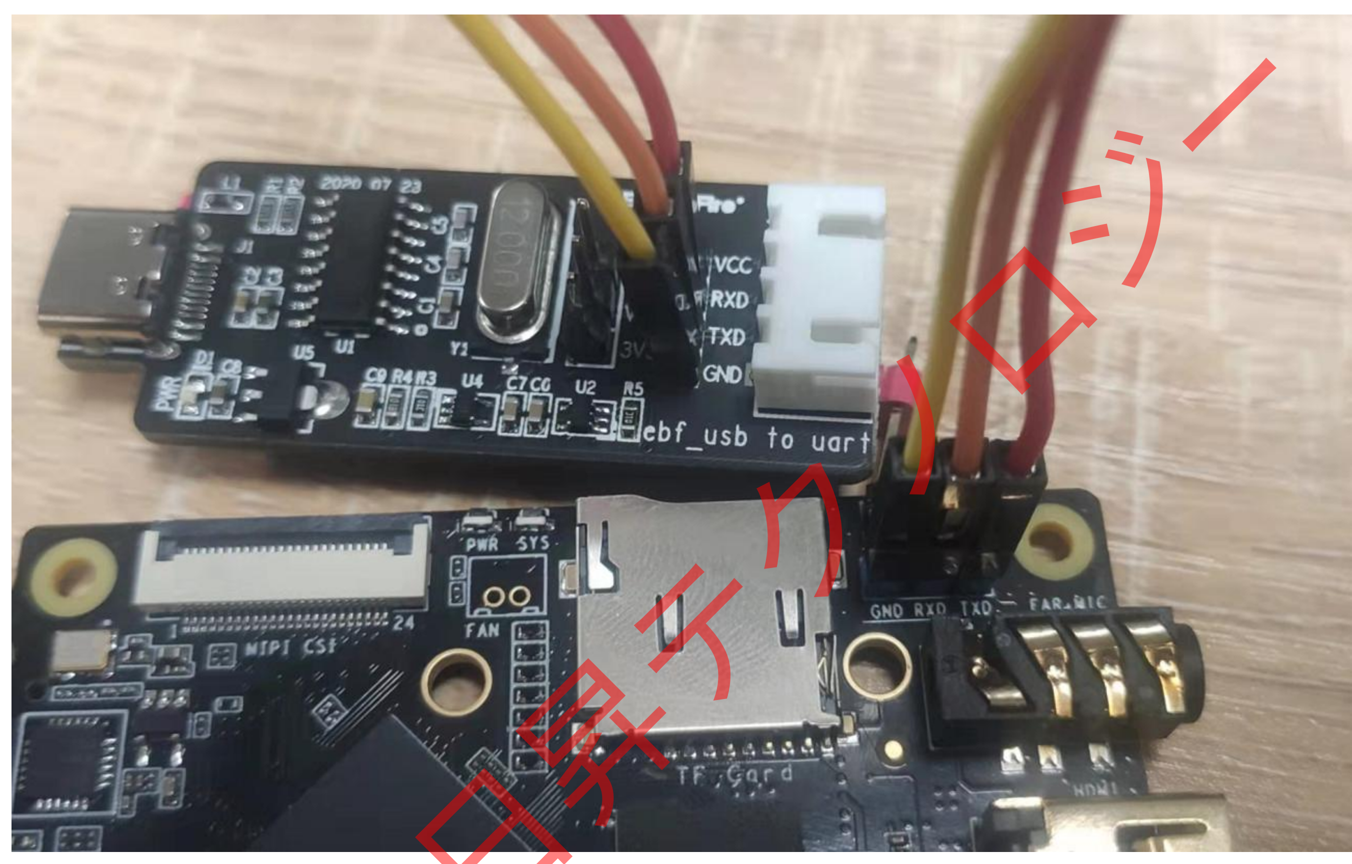
ボード----- シリアルポート

株式会社日昇テクノロジー

GND ------ GND

TXD ------ RXD

RXD ------ TXD



或いは

株式会社日昇テクノロジー

注意: ピンのシルクの位置は裏側にあることがあります。接続する際にシルクの位置に注意してください。

第二の方法は、XR 2.5mm 1x4P コネクタを使用することです。

6.2.2 ソフトウェアの準備

MobaXterm のインストールと使用

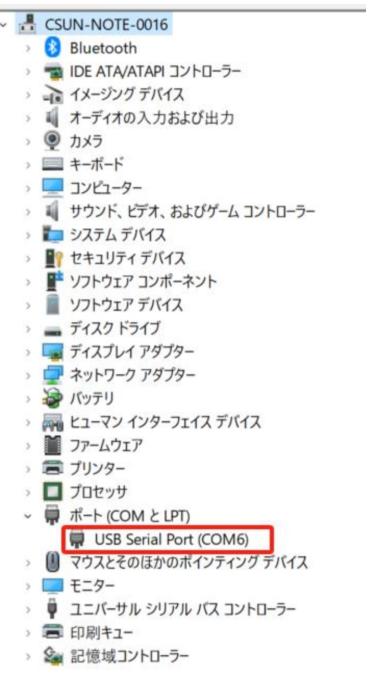
- ダウンロード：https://mobaxterm.mobatek.net/download.html

MobaXterm 端末ソフトウェアの使用方法については、「Linux 基本とアプリケーション開発実践ガイド - i.MX6ULL ベース」を参照してください。

MobaXterm ターミナルソフトの詳しい使い方は、「Linux 基礎とアプリケーション開発実践ガイド- i.MX6ULL ベース-」を参照してください。 パラメータ設定が異なるだけで、使い方は同じです。

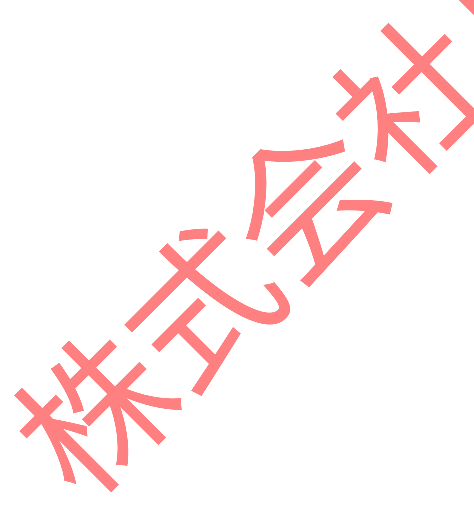
シリアルモジュールをパソコンに接続し、パソコンのデバイスマネージャーを開き、ポート名を確認します。

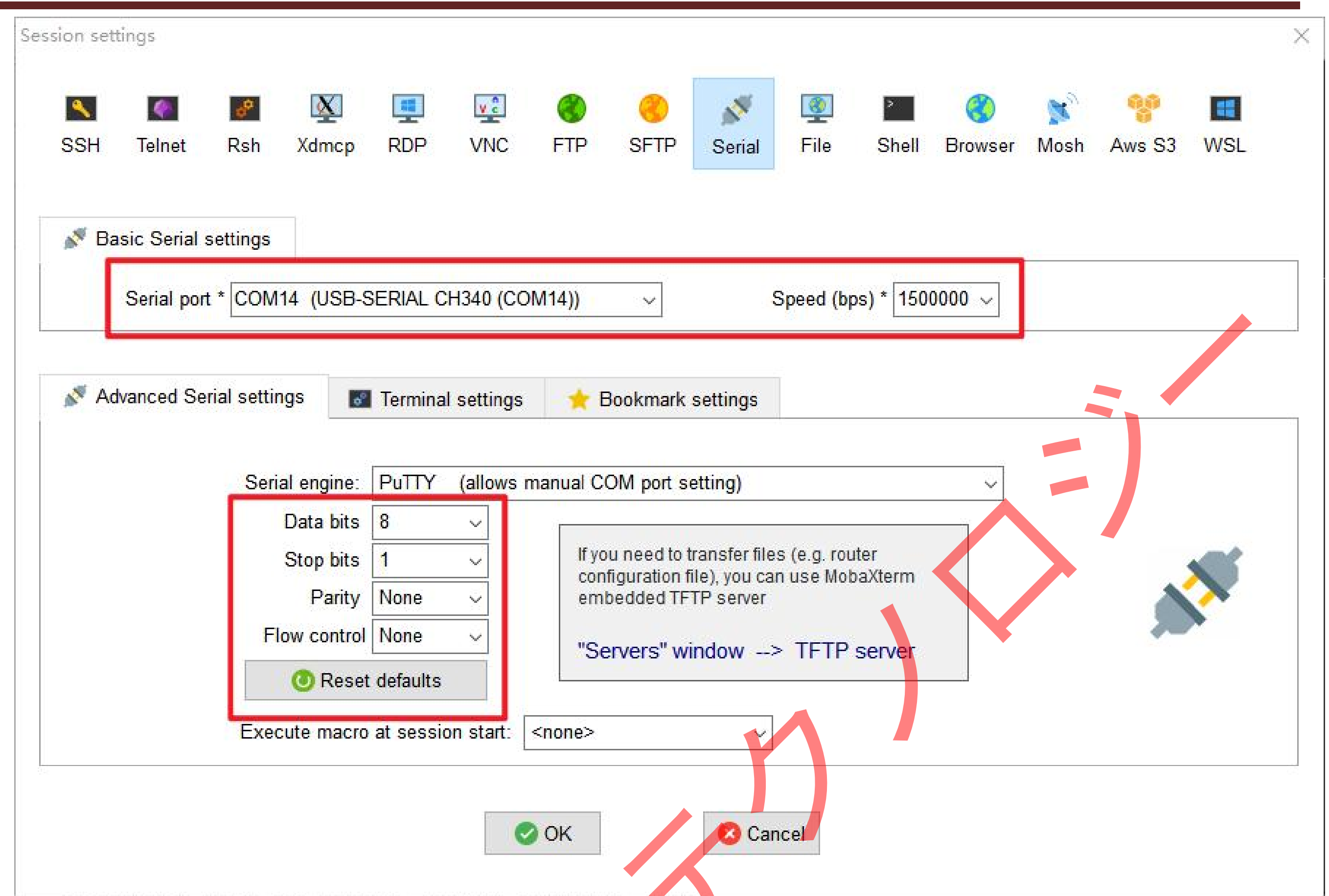
株式会社日昇テクノロジー



次に MobaXterm を開き、セッションのアイコンをクリックしてセッション設定を表示し、シリアルを選択します。

正しいシリアルポートを選択し、ボーレートを 1500000 に設定し、フロー制御をオフにします。


株式会社日昇テクノロジー



自分のシリアルポートを選択し、上の図に従ってソフトウェアの設定を行う。

OK をクリックして通信を開始すると、ボードの電源が入っていないため、応答がありません。

6.2.3 電源投入

ほとんどのボードは 5V Type-C 電源をサポートしていますが、中には DC12V または DC5V 電源を使用するボードもあります。

そのため、ボードのモデルに応じて適切な電源を選択する必要がある。

ボード給電

Type-C(5V@4A)	LubanCat-4
DC5V(5V@4A)	LubanCat-4

ボードに対応する電源が手に入ったら、それを接続するための正しいインターフェースを選択しなければならない。

ボードによっては Type-C コネクタが 2 つあるので、Type-C コネクタの近くにあるシルクスクリー

株式会社日昇テクノロジー

ンに pwr と書かれているか確認する必要がある。

pwr または OTG と書かれていれば、そのポートを使ってデバイスの電源を入れることができる。

電源を接続することで、ボードの電源を入れることができる。

6.3 デバイスの電源を入れる

注意:書き込みした後の最初の起動は再起動になるのが普通です。

起動後、多くのプリントメッセージが表示されますが、これは正常です。

```
[    1.516057] usb 5-1: New USB device found, idVendor=09da, idProduct=c10a, bcdDevice=94.06
[    1.516109] usb 5-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[    1.516123] usb 5-1: Product: USB Mouse
[    1.516135] usb 5-1: Manufacturer: A4Tech
[    1.537750] input: A4Tech USB Mouse Mouse as /devices/platform/usbhost/fd000000.dwc3/xhci-hcd.0.auto/usb5/5-1/5-1:1.0/0003:09DA:C10A.0001/input/input5
[    1.545495] devfreq fde60000.gpu: Couldn't update frequency transition information.
[    1.594654] input: A4Tech USB Mouse as /devices/platform/usbhost/fd000000.dwc3/xhci-hcd.0.auto/usb5/5-1/5-1:1.0/0003:09DA:C10A.0001/input/input6
[    1.596770] hid-generic 0003:09DA:C10A.0001: input,hiddev96,hidraw0: USB HID v1.10 Mouse [A4Tech USB Mouse] on usb-xhci-hcd.0.auto-1/input0
[    1.598024] ALSA device list:
[    1.598053]   #0: rockchip,rk809-codec
[    1.611157] EXT4-fs (mmcblk0p3): mounted filesystem with ordered data mode. Opts: (null)
[    1.611318] VFS: Mounted root (ext4 filesystem) on device 179:3.
[    1.612050] devtmpfs: mounted
[    1.616140] Freeing unused kernel memory: 1536K
[    1.637525] Run /sbin/init as init process
[    1.844211] systemd[1]: Failed to find module 'autofs4'

Welcome to Debian GNU/Linux 10 (buster)!

[    2.011430] systemd-fstab-generator[124]: Ignoring "noauto" for root device
[    2.144157] phy phy-fe8a0000.usb2-phy.1: charger = USB_DCP_CHARGER
[  OK  ] Listening on Journal Socket (/dev/log).
[  OK  ] Listening on Network Service Netlink Socket.
[  OK  ] Reached target Remote File Systems.
[  OK  ] Reached target Swap.
[  OK  ] Started Dispatch Password …ts to Console Directory Watch.
[  OK  ] Listening on Syslog Socket.
[  OK  ] Created slice system-getty.slice.
[  OK  ] Created slice system-systemd\x2dfsck.slice.
[  OK  ] Listening on initctl Compatibility Named Pipe.
[  OK  ] Listening on fsck to fsckd communication Socket.
[  OK  ] Started Forward Password R…uests to Wall Directory Watch.
[  OK  ] Reached target Local Encrypted Volumes.
[  OK  ] Reached target Remote Encrypted Volumes.
[  OK  ] Listening on udev Control Socket.
[  OK  ] Listening on udev Kernel Socket.
[  OK  ] Created slice system-serial\x2dgetty.slice.
[  OK  ] Listening on Journal Socket.
         Starting Nameserver information manager...
         Starting udev Coldplug all Devices...
         Starting Journal Service...
         Mounting POSIX Message Queue File System...
[    2.274194] rk-pcie 3c0000000.pcie: PCIe Linking... LTSSM is 0x3
         Mounting /sys/kernel/debug...
```

しばらく待つと、下の写真のようにログインできるようになります。

株式会社日昇テクノロジー

```
[    6.697880] IPv6: ADDRCONF(NETDEV_UP): eth1: link is not ready
[    6.698410] JL2101 Gigabit Ethernet stmmac-1:00: attached PHY driver [JL2101 Gigabit Ethernet] (mii_bus:phy_addr=stmmac-1:00, irq=POLL)
[    7.341107] rk-pcie 3c0800000.pcie: PCIe Linking... LTSSM is 0x0
[    7.341147] rk-pcie 3c0000000.pcie: PCIe Linking... LTSSM is 0x3
[    7.997991] Freeing drm_logo memory: 4748K
[    8.221147] dwmac4: Master AXI performs any burst length
[    8.221242] rk_gmac-dwmac fe010000.ethernet eth1: No Safety Features support found
[    8.221277] rk_gmac-dwmac fe010000.ethernet eth1: IEEE 1588-2008 Advanced Timestamp supported
[    8.221827] rk_gmac-dwmac fe010000.ethernet eth1: registered PTP clock
[    8.222687] IPv6: ADDRCONF(NETDEV_UP): eth1: link is not ready
[  OK  ] Created slice User Slice of UID 1000.
         Starting User Runtime Directory /run/user/1000...
[  OK  ] Started User Runtime Directory /run/user/1000.
         Starting User Manager for UID 1000...
[    8.354395] rk-pcie 3c0000000.pcie: PCIe Linking... LTSSM is 0x3
[    8.354398] rk-pcie 3c0800000.pcie: PCIe Linking... LTSSM is 0x0
[  OK  ] Started User Manager for UID 1000.
[  OK  ] Started Session c1 of user cat.
[    9.367877] rk-pcie 3c0000000.pcie: PCIe Linking... LTSSM is 0x3
[    9.367931] rk-pcie 3c0800000.pcie: PCIe Linking... LTSSM is 0x0
[    9.842363] ttyFIQ ttyFIQ0: tty_port_close_start: tty->count = 1 port count = 2
[   10.381189] rk-pcie 3c0000000.pcie: PCIe Linking... LTSSM is 0x3
[   10.381192] rk-pcie 3c0800000.pcie: PCIe Linking... LTSSM is 0x0

Debian GNU/Linux 10 lubancat ttyFIQ0

[username:password] root:root cat:temppwd

Modify information : /etc/issue

lubancat login: [   11.394632] rk-pcie 3c0000000.pcie: PCIe Link Fail
[   11.394669] rk-pcie 3c0000000.pcie: failed to initialize host
[   11.394830] rk-pcie 3c0800000.pcie: PCIe Link Fail
[   11.394842] rk-pcie 3c0800000.pcie: failed to initialize host
[   12.275452] rk_gmac-dwmac fe010000.ethernet eth1: Link is Up - 1Gbps/Full - flow control off
[   12.275556] IPv6: ADDRCONF(NETDEV_CHANGE): eth1: link becomes ready
```

注：ログイン画面を入力するには、画像のテキストを参照してください。時にはいくつかのエラーがあるかもしれませんが、使用には影響しません。

ログインにはユーザー名とパスワードが必要です

ユーザー　-　　ユーザー名 - パスワード

スーパーユーザー - root -- root

一般ユーザー　-　　cat -- temppwd

ユーザー名（大文字と小文字を区別必要）を入力し、次にパスワードを入力する（パスワードのテキストプロンプトはないので、大文字小文字に注意する必要がある）。

（パスワードのテキストプロンプトはありませんので、大文字小文字に注意する必要があります）。

ユーザー名とパスワードが正しければ、ログインできます。

6.4 ネットワーク接続

6.4.1 ネットワークケーブルの接続

ネットワークケーブルを接続するには、以下のインターフェイスを選択し、ケーブルの形状に合わせてインターフェイスにケーブルを接続する必要があります。

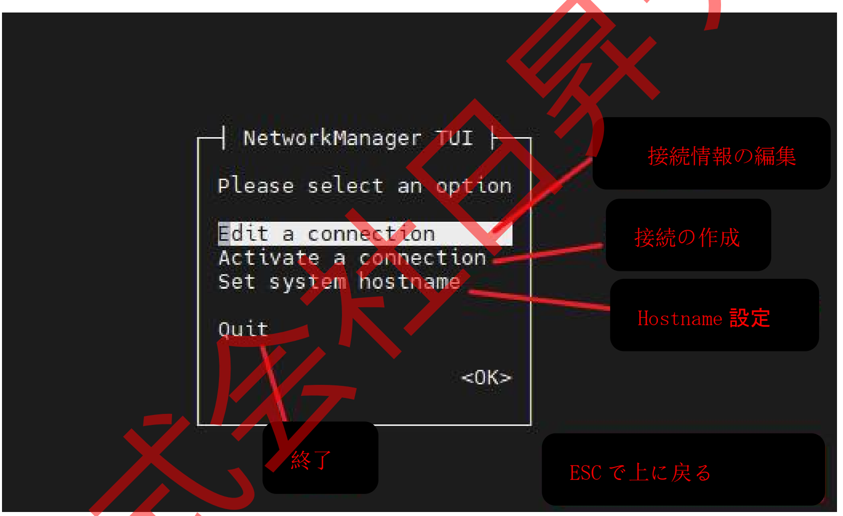
株式会社日昇テクノロジー

6.4.2 Wifi 接続

- グラフィカル設定に入るには

```
sudo nmtui
```

- キーボードの矢印キーをアクティブ接続に移動し、Enter キーを押して Wifi 設定に入ります。

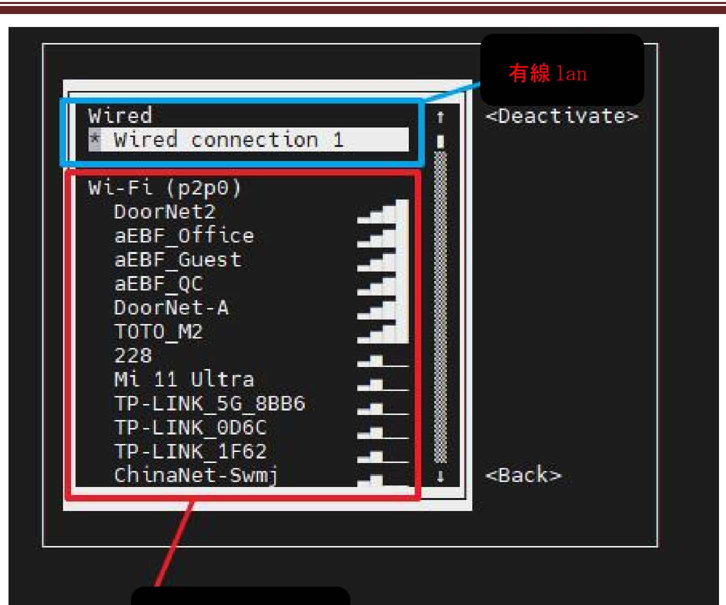


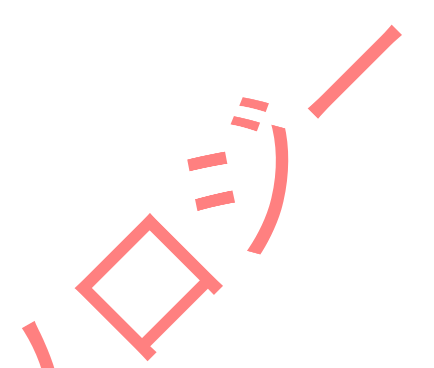
- 次に、矢印キーで接続したい無線 LAN に移動し、Enter キーを押します。

- 未接続のホットスポットにパスワード付きで接続必要の場合は、パスワード入力画面に入ります。

- パスワード必要ない未接続のホットスポットに接続している場合は、直接接続できます。

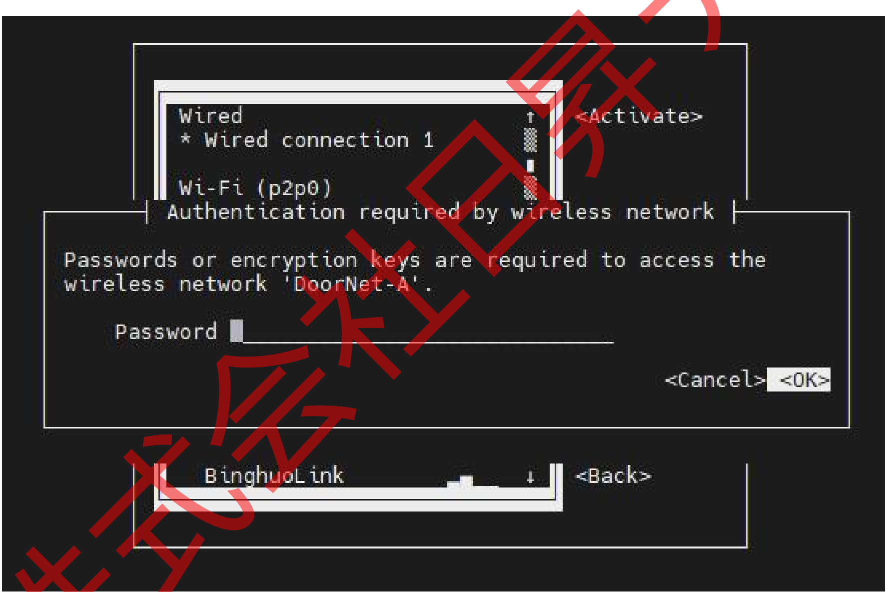
株式会社日昇テクノロジー





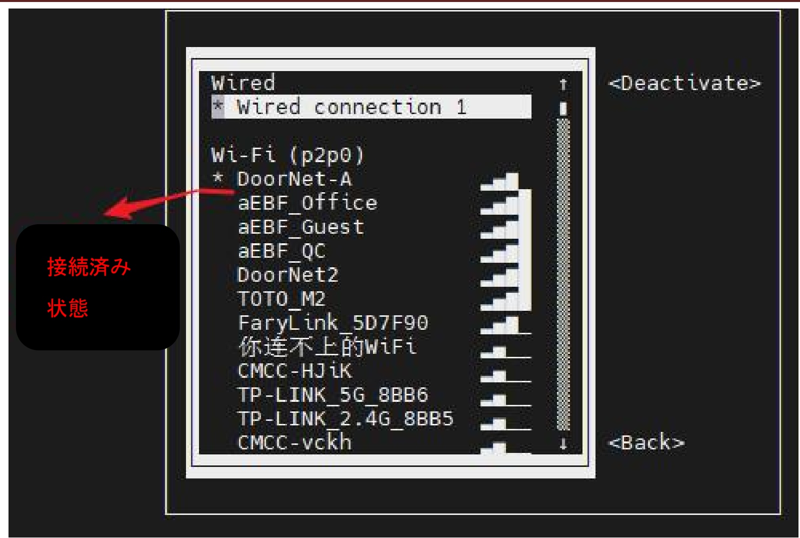
注意: いくつかの wifi チープ (例えば rtl8821cu)を使用するとき、システムは p2p0 と wlan0 の 2 つのネットワークアクセスポイントを生成し、wifi はどちらのポイントとも接続できます。

初めてパスワード付きホットスポットに接続する場合は、以下のようになります。



- パスワードを入力すると、無線 LAN が接続されていることが確認できます。

株式会社日昇テクノロジー



wifi を切断したい場合は、接続状態で Enter キーを押すと切断されます。

6.5 機能の紹介

6.5.1 シンプルなコマンドラインを使う

6.5.1.1 ディレクトリ内のファイルをリストする

```
# カレントディレクトリのファイルをリストする

ls
```

他のディレクトリのファイルをリストする

ls + 他のディレクトリ

```
root@lubancat:/home# ls
cat
root@lubancat:/home# ls /boot
Image-4.19.232      boot_init          dtb         logo.bmp     rk-kernel.dtb
System.map-4.19.232 config-4.19.232    kerneldeb   lost+found   uEnv
root@lubancat:/home#
```

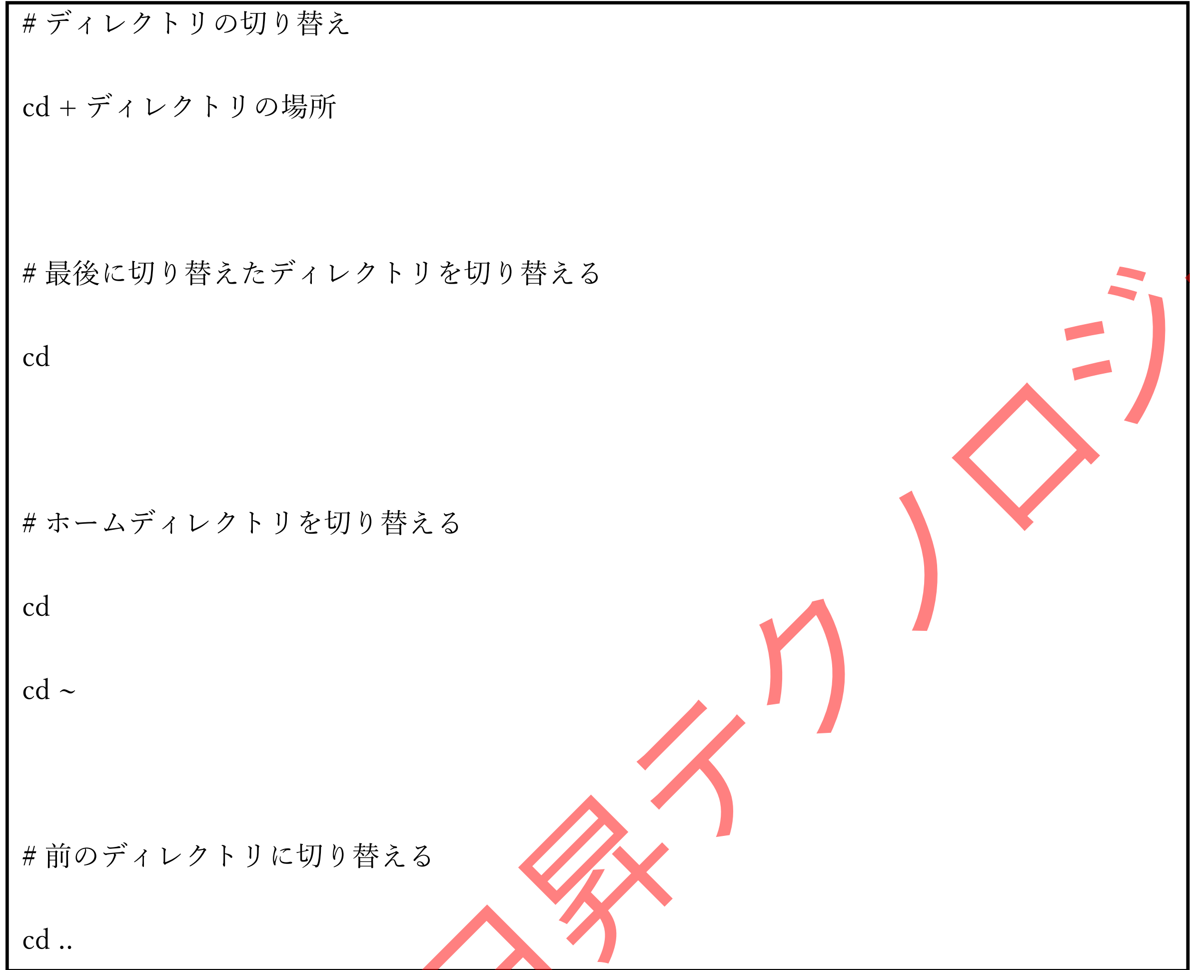
6.5.1.2 ディレクトリの場所

```
# カレントディレクトリの場所をリストする

Pwd
```

```
root@lubancat:/home#
root@lubancat:/home# pwd
/home
root@lubancat:/home#
```

株式会社日昇テクノロジー

6.5.1.3 ディレクトリの切り替え



ディレクトリの切り替え

cd + ディレクトリの場所

最後に切り替えたディレクトリを切り替える

cd

ホームディレクトリを切り替える

cd

cd ~

前のディレクトリに切り替える

cd ..

```
root@lubancat:~# cd /boot/
root@lubancat:/boot# pwd
/boot
root@lubancat:/boot# ls
Image-4.19.232       boot_init         dtb         logo.bmp     rk-kernel.dtb
System.map-4.19.232  config-4.19.232   kerneldeb   lost+found   uEnv
root@lubancat:/boot#
```

6.5.2 ソフトウェアの更新

Debian と Ubuntu に含まれるソフトウェアのほとんどは、インターネット上で入手可能で、自動的にアップデートされます。

image ファイルは古いバージョンソフトウェアを使用している可能性があり、通常の使用に影響するバグがあるかもしれません。適切なタイミングでシステムにインストールされているソフトウェアをアップデートしてください。

株式会社日昇テクノロジー

注意: イメージを書き込みした後の最初の起動後、ソフトウェアを更新する必要があります。

```
# ソフトウェアのアップデート

sudo apt update



# アップデートしたソフトウェアをインストールする

sudo apt upgrade
```

6.5.3 ソフトウェアのインストールとアンインストール

6.5.3.1 ソフトウェアのインストール

```
sudo apt install xxx
```

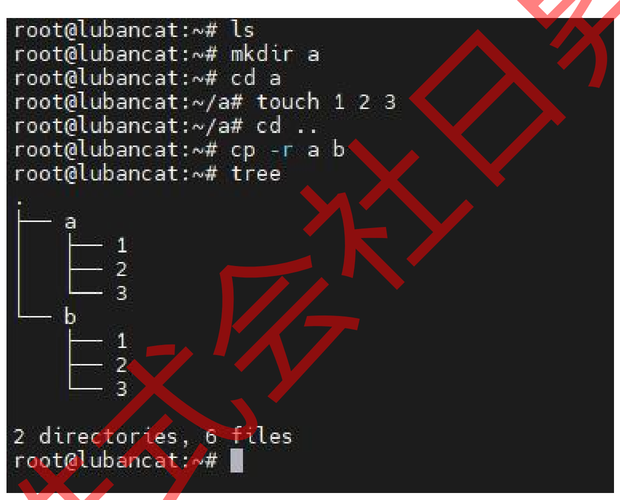
ここでは、tree ソフトウェアを例にとって説明します。tree は、フォルダを表示するためのツリー構造です。

```
# ソフトウェアのアップデート

sudo apt install tree



 # フォルダーを作成する

# mkdir a



# フォルダーを切り替える

cd a
```

株式会社日昇テクノロジー

```
# 複数のファイルを作成する

touch 1 2 3


# 前のディレクトリに切り替える

cd ..


# b というフォルダーをコピーする

cp -r a b


# ツリーを使ってフォルダを表示する

tree
```

下図のように


```
root@lubancat:~# ls
root@lubancat:~# mkdir a
root@lubancat:~# cd a
root@lubancat:~/a# touch 1 2 3
root@lubancat:~/a# cd ..
root@lubancat:~# cp -r a b
root@lubancat:~# tree
.
├── a
│   ├── 1
│   ├── 2
│   └── 3
└── b
    ├── 1
    ├── 2
    └── 3

2 directories, 6 files
root@lubancat:~#
```

株式会社日昇テクノロジー

6.5.3.2 ソフトウェアのインストール

```
sudo apt remove xxx
```

tree を例にとって説明しよう。

```
sudo apt remove tree
```

アンインストールすると、tree コマンドは使えなくなります。

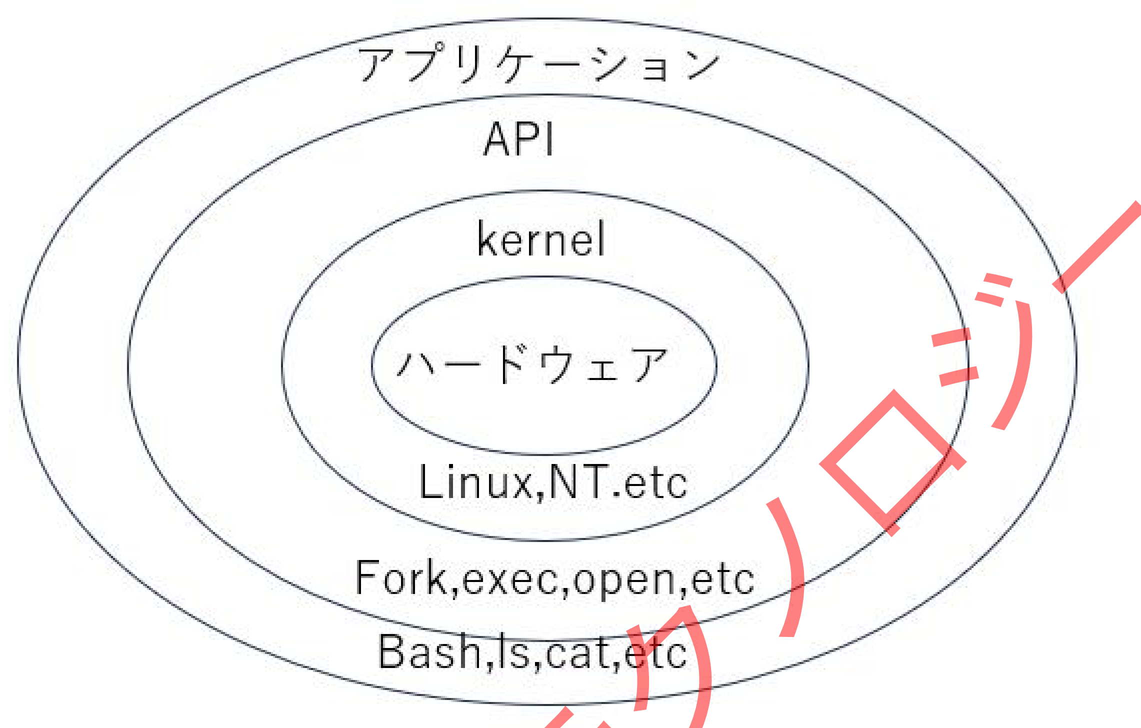
第 7 章 Linux コマンドライン

LubanCat-RK のボードのデスクトップには多くの機能があります。グラフィカルインターフェイスを通じてほとんどの作業を完了することができ、日常の基本的なアプリケーションもそのグラフィックインターフェースを通じて完了することができます。しかし、どんなにインターフェイスが完璧に作られていても、インターフェイスを通じて完了できない操作がたくさんあります。なぜなら、ほとんどのプログラム自体がインターフェイスを提供していないか、またはインターフェイスがいくつかの一般的な操作のみを提供しているからです。しかし、ほぼすべてのプログラムはコマンドラインを通じて呼び出し、実行することができ、コマンドラインを通じて豊富な操作オプションを提供できます。特に開発を行う際には、IDE があると考えることは良いことですが、コマンドラインのみを使用する状況を受け入れる必要があります。実際には、使用すると慣れてきます。

Linus Torvalds（リーナス・トーバルズ）は自伝の中で、Linux オペレーティングシステムのカーネルを書いているとき、最初の目標は Linux システムが Unix の Shell プログラムを正常に実行できるようにすることだったと述べています。Shell はコマンドラインインタープリタであり、Linux システムが Unix の Shell と互換性があれば、多くの Unix プログラムも簡単に互換性を持たせることができます。

Unix の初期システムでは、ユーザーは Shell を介してシステムと対話していました。下の図に示されているように、Shell はユーザーからの入力コマンドを受け取り、内部では「システムコール」を通じてカーネルに伝え、カーネルが操作を実行した後、出力を Shell を通じてユーザーに表示します。つまり、Shell は仲介者のようなものです。Shell の英語の原意は「殻」であり、それをカーネルと区別するためです。

株式会社日昇テクノロジー



通常のコミュニケーションでは、Shell、ターミナル（Terminal）、またはコンソール（Console）を開くと言うことがあります。厳密に言えば、これらは同じものではありませんが、Shell、ターミナル、またはコンソールを開くと言うときは、通常、コマンドラインを使用してシステムを制御するためであると理解すれば十分です。それらの厳密な区別は以下の通りですが、知っておくだけで良いでしょう：

- Shell：コマンドラインインタープリタを指します。一般的なインタープリタには bash、sh があり、Ubuntu システムではデフォルトで bash インタープリタが使用されています。そのため、bash と言う場合もコマンドラインを指すことがあります。

- ターミナル（Terminal）：Shell を実行するためのプログラムを通常指します。シーンによ異なる名称を持っています。例えば Ubuntu システムには「ローカルターミナル」と呼ばれるものが標準で含まれていますし、組み込み開発ボードでは、入出力のためのシリアルポートターミナルがよく提供されます。また、ネットワーク経由の SSH ターミナルもあります。

株式会社日昇テクノロジー

- コンソール（Console）：特定のターミナルを指し、通常はその物理的な形態を指します。例えば、キーボードとディスプレイを備えた物理的なデバイスです。

7.1 ターミナルの開き方

ターミナルを開く方法は 3 つあります。最も便利な方法を選んでターミナルにログインしてください。

1. シリアル接続（初心者やボードが手元にある人向け）

2. SSH 接続（リモートユーザー向け）

3. デスクトップからターミナルを開く（デスクトップユーザー向け）

7.1.1 シリアル接続

シリアル接続の方法

1. シリアル通信が可能なツールをインストールします。ここでは Mobaxterm を推奨します。

2. シリアルケーブルを使用してボードに接続します。

3. Mobaxterm を設定し、Mobaxterm がボードと通信できるようにします。

具体的な方法は「システムの起動とログイン」を参照してください。

7.1.2 SSH 接続

SSH 接続の方法

1. ボードが起動している状態であること。

2. ボードが通信するコンピューターと同じローカルエリアネットワークに接続されていること。

3. SSH 通信が可能なソフトウェアを開きます。ここでは Mobaxterm を推奨します。

4. Mobaxterm を設定し、Mobaxterm がボードと通信できるようにします。

具体的な方法は「SSH ターミナルログイン」を参照してください。

株式会社日昇テクノロジー

7.1.3 デスクトップからターミナルを開く

- ソフトウェアマネージャーから Terminal を開くことができます。

- キーボードで「Ctrl + Alt + t」を押すことでデスクトップターミナルを開くことができます。

7.2 コマンドプロンプト

ターミナルを開いた後、ターミナル自体が一行の文字を表示し、Enter キーを押すと再度表示されるのを見ることができます：

```
Welcome to Debian GNU/Linux 10 (buster) (GNU/Linux 4.19.232 aarch64)

 * Documentation:  http://doc.embedfire.com
 * Management:     http://www.embedfire.com
Linux lubancat 4.19.232 #18 SMP Sat Nov 12 16:24:03 CST 2022 aarch64

System information as of Thu Feb 14 18:12:13 CST 2019

System load:    1.53 0.35 0.12    Up time:        0 min
Memory usage:   12 % of 1967MB    IP:
CPU temp:       30°C              GPU temp:       30°C
Usage of /:     10% of 29G

root@lubancat:~#
```

```
root@lubancat:~#
```

実際には、このプロンプトは以下の部分に分かれています：

- root ：「@」記号の左側で、現在ログインしているユーザーを示しています。上の例では root ユーザーでログインしています。

- @ ：区切り記号で、at と理解できます。これは root ユーザーが lubancat ホスト上にいることを示します。

- lubancat ：現在のシステムのホスト名です。

- 「:」：区切り記号です。

- 「~」：コロンの後には、ユーザーが現在いるディレクトリを示します。ここでのチルダは、現在のユーザーのホームディレクトリをリストしています。root ユーザーのホームディレクトリは/root で、通常のユーザーのディレクトリは/home/（ユーザー名）です。

- 「#」：コマンドプロンプトです。Linux では、この記号を使用して、ログインしたユーザーの権限レ

株式会社日昇テクノロジー

ベルを示します。スーパーユーザーであれば、プロンプトは「#」になり、通常のユーザーであれば、「$」になります。

7.3 コマンドライン体験

余計な話は抜きにして、以下のコマンドをターミナルに入力してみてください。日本語入力を使用している場合は、英語モードに切り替えることを忘れないでください。また、以下のコマンド中のチルダ「~」も英語の記号です。さらに、ターミナル内のすべての内容は大文字と小文字が区別されることに注意してください：

```
1 # 以下の説明で、「#」記号はコメントを意味し、その後の内容はターミナルに入力しないでください
2 # 以下の説明で、「#」記号はコメントを意味し、その後の内容はターミナルに入力しないでください
3 # 以下からは、ターミナルに入力するコマンドです（「#」記号の後の内容は含みません）：
4
5 cd /home # /home ディレクトリに移動します
6 pwd # 現在のディレクトリを表示します
7 cd ~ # ホームディレクトリに移動します。~はホームディレクトリをリストし、ユーザーごとに異なります
8 pwd # 現在のディレクトリを表示します
9 touch hello # hello という名前のファイルを作成します
10 touch abc.c # abc.c という名前のファイルを作成します
11 ls # l は小文字の L であり、数字の 1 や大文字の I と間違えないでください
12 ls -l # 同上、l はすべて小文字の L です
```

各行のコマンドを入力して Enter キーを押すと、Shell が入力されたコマンドを解釈し、実行後に結果を出力します。

株式会社日昇テクノロジー

```
root@lubancat:~# cd /home/
root@lubancat:/home# pwd
/home
root@lubancat:/home# cd ~
root@lubancat:~# pwd
/root
root@lubancat:~# touch hello
root@lubancat:~# touch abc.c
root@lubancat:~# ls
abc.c  hello
root@lubancat:~# ls -l
total 0
-rw-r--r-- 1 root root 0 Feb 14 18:21 abc.c
-rw-r--r-- 1 root root 0 Feb 14 18:21 hello
root@lubancat:~#
```

上記の各コマンドの意味は以下の通りです：

- cd /home：「cd」コマンドを実行し、「/home」をパラメータとして使用します。これは、/home ディレクトリに切り替えることを意味します。コマンド実行後、コマンドプロンプトが root@lubancat:~# から root@lubancat:/home# に変わります。つまり、チルダ「~」が「/home」に変わります。

- pwd：「pwd」コマンドを実行し、現在のディレクトリを出力します。出力結果は「/home」であり、コマンドプロンプトに表示される内容と一致します。

- cd ~：再度「cd」コマンドを実行し、「~」をパラメータとして使用します。これは、ホームディレクトリに切り替えることを意味します。コマンド実行後、コマンドプロンプトのパスが「/home」から「~」に戻ります。

- pwd：「pwd」コマンドを再実行し、現在のディレクトリを出力します。出力結果は「/root」で、これは現在のユーザーのホームディレクトリです。これは、コマンドプロンプトに表示される「~」と同じ意味です。

- touch + ファイル名：ファイルを作成します。

- ls：「ls」コマンドを実行します（これは小文字の L であり、数字の 1 や大文字の I ではありません）。このコマンドは、現在のディレクトリ下の内容をリストアップします。

- ls -l：「ls」コマンドを実行し、「-l」オプションを付けます（これも小文字の L であり、数字の 1 や大文字の I ではありません）。このオプションは、ディレクトリ下の詳細内容をリスト形式で表示します。これにより、通常の ls コマンドよりも多くの内容が表示されます。詳細は後ほど分析します。

株式会社日昇テクノロジー

7.4 コマンドの形式とヘルプ

7.4.1 コマンドの形式

上記のいくつかのコマンド例を通じて、コマンドの形式を大まかにまとめることができます：

```
1 command [-options] [argument]
```

コマンドは一般的に 3 部分で構成され、各部分は空白で区切られます。説明は以下の通りです：

- command：コマンド名です。前述の cd コマンド、pwd コマンド、ls コマンドなどがこれに該当します。

- -options：コマンドのオプションで、「-」または「--」で始まります。例えば、前述の ls -l コマンドでの「-l」オプションのように、コマンドは具体的なオプションに基づいて異なる操作を実行します。「-」を使用するときは一般にオプションの省略形で、いくつかのオプションを同時に使用することができます。例えば、ls -la は ls -l -a と等価で、「l」と「a」オプションを同時に使用しています。一方、「--」の後にはオプションの全名が来ることが一般的です。例えば、ls -a は ls --all と等価です。

- argument：コマンドの引数です。例えば、前述の cd コマンドでは「/home」や「~」をパス名として使用しています。

コマンドの形式では、「[ ]」で囲まれた内容は必須ではないことを示しています。例えば、cd コマンドをオプションなしで使用したり、pwd コマンドをオプションや引数なしで使用する場合などです。

7.4.2 コマンドのヘルプ

Linux には数え切れないほどのコマンドがあり、異なるコマンドには異なるオプションがあり、入力される引数にも異なる意味があります。ただ「使えば覚える」という無責任な言葉を投げかけるのではなく、コマンドを使用するにはいくつかのコツがあります。

各コマンドには「-h」または「--help」という引数があり、いくつかのヘルプ説明を表示するために使用できます。例えば、現在 ls コマンドのオプション a がどのように使用されるのかわからない場合は、以

株式会社日昇テクノロジー

下のコマンドを実行できます：

```
1 ls –help
```

```
root@lubancat:~# ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print C-style escapes for nongraphic characters
      --block-size=SIZE      with -l, scale sizes by SIZE when printing them;
                               e.g., '--block-size=M'; see SIZE format below
  -B, --ignore-backups       do not list implied entries ending with ~
  -c                         with -lt: sort by, and show, ctime (time of last
                               modification of file status information);
                               with -l: show ctime and sort by name;
                               otherwise: sort by ctime, newest first
  -C                         list entries by columns
```

上記の説明から、引数 a が隠された内容のみをリストアップすることを知ることができます。今、以下のコマンドを実行します：

```
1 ls -a
```

```
root@lubancat:~#
root@lubancat:~#
root@lubancat:~# ls -a
.  ..  .bash_history  .bashrc  .cache  .config  .gnupg  .profile  abc.c  hello
root@lubancat:~# 
```

上図の説明から、「ls」コマンドだけを使用する場合と比べて、「.」で始まる多くの内容がリストアップされていることがわかります。Linux では、「.」で始まるファイル名やディレクトリ名はデフォルトで隠しファイルとされており、「ls」の「-a」オプションを使用すると、これらを表示することができます。

補足説明ですが、ヘルプ情報には「-a」の後に「--all」パラメータもあり、これら二つのパラメータは等価ですが、「--all」の方が覚えやすいかもしれません。

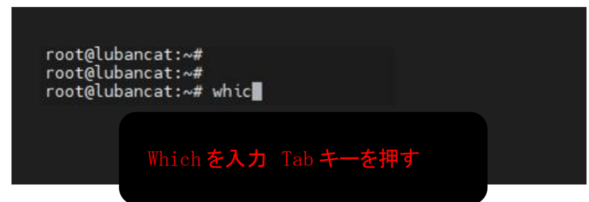
7.4.3 オートコンプリート

時にはコマンドの全名さえ思い出せないことがあります。そんな時には、コマンドラインのオートコンプリート機能が役立ちます。それは、キーボード上の神秘的な「Tab」キーです。

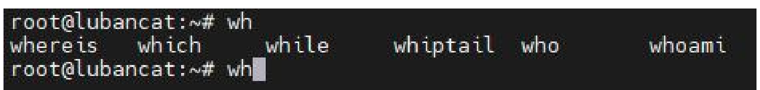
例えば、「whic」で始まるコマンドがあることは知っているが、全名が思い出せない場合、ターミナルに「whic」と入力してから「Tab」キー（キーボード上の Q の左側）を 1 回押すと、「which」というコ

株式会社日昇テクノロジー

マンドに自動補完されます。

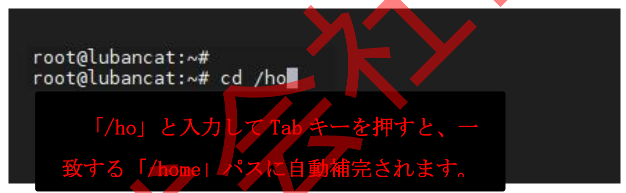


もし、コマンドが「wh」で始まることしか覚えていない場合、「Tab」キーを押しても反応がないかもしれません。その場合は、「Tab」キーを 2 回押してみてください。すると、以下のように「wh」で始まる多くのコマンド、例えば「which」、「who」、「whoami」などが表示されます。



したがって、「Tab」キーを 1 回押すと、一致する内容が 1 つだけの場合には自動補完され、2 回押すと、すべての一致する項目がリストアップされます。

「Tab」キーはコマンド名の補完だけでなく、パスの自動補完にも使用できます。たとえば、「cd」コマンドで「/home」というパラメータを入力する場合、「/ho」と入力してから「Tab」キーを押すと、「/ho」が自動的に「/home」のパス名に補完されます。これはコマンドラインを使用する際に頻繁に使用されるテクニックで、入力を減らし、誤りを避けるのに役立ちます。



- ファイル名の補完も可能です。ファイルを操作するときに全名を入力する必要はなく、名前の一部を入力してから Tab キーを押すことで補完できます。

```
1 # 新しいファイルを作成
2 touch embedfire
3
4 # cat em と入力
5 # Tab キーを押すと、ファイル名が自動的に補完されます。
```

株式会社日昇テクノロジー

```
root@lubancat:~# ls
abc.c  hello
root@lubancat:~# touch embedfire
root@lubancat:~# cat e
```

「cat em」と入力し、Tab キーを押すと自動補完されます。

7.4.4 コマンドの終了とキャンセル

アプリケーションでコマンドの実行を中断したい場合や、コマンドを半分入力した後に間違えたと感じて続行したくない場合は、「Ctrl」+「c」のキーの組み合わせで終了できます。

例えば、ターミナルに以下を入力します：

```
1 # 以下の説明で、「#」記号はコメントを意味し、その後の内容はターミナルに入力しないでください
2 ping 127.0.0.1 # 自分のネットワークアドレス 127.0.0.1 への ping テストを試みます
```

「ping」コマンドは実行すると継続して出力されますが、「Ctrl」+「c」のキーコンビネーションを押すことで終了できます。

```
root@lubancat:~# ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.234 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.349 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.354 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.286 ms
^C
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 56ms
rtt min/avg/max/mdev = 0.234/0.305/0.354/0.053 ms
root@lubancat:~#
```

Ctrl+C を押す

プログラムを中断して終了する

また、たとえば、以下のようなコマンドをターミナルに入力した後、実行したくなくなった場合、バックスペースキーを使って一文字ずつ消去するのは面倒です。その場合も、「Ctrl」+「c」のキーコンビネーションでキャンセルできます：

```
1 ls adfadfadsfadsfa
```

上記のコマンドを入力した後に実行したくない場合は、「Ctrl」+「c」を押して終了します。

株式会社日昇テクノロジー

7.4.5 コマンドとは実際には何か

「ls」コマンドを打った後、Shell は実際に何をしているのでしょうか。以前「Linux ファイルディレクトリ/bin ディレクトリ」のセクションで言及されたように、/bin ディレクトリには多くのコマンドが含まれています。たとえば、ls、lsblk、lsmod、mkdir などのコマンドプログラムがあります。

```
root@lubancat:~# ls /bin
bash           fusermount     ntfs-3g        systemd
bunzip2        grep           ntfs-3g.probe  systemd-ask-password
busybox        gunzip         ntfscat        systemd-escape
bzcat          gzexe          ntfscluster    systemd-hwdb
bzcmp          gzip           ntfscmp        systemd-inhibit
bzdiff         hciconfig      ntfsfallocate  systemd-machine-id-setup
bzegrep        hostname       ntfsfix        systemd-notify
bzexe          ip             ntfsinfo       systemd-sysusers
bzfgrep        journalctl     ntfsls         systemd-tmpfiles
bzgrep         kbd_mode       ntfsmove       systemd-tty-ask-password-agent
bzip2          kill           ntfsrecover    tar
bzip2recover   kmod           ntfssecaudit   tempfile
bzless         less           ntfstruncate   touch
bzmore         lessecho       ntfsusermap    true
cat            lessfile       ntfswipe       udevadm
chgrp          lesskey        openvt         ulockmgr_server
chmod          lesspipe       pidof          umount
chown          ln             ping           uname
chvt           loadkeys       ping4          uncompress
cp             login          ping6          unicode_start
cpio           loginctl       ps             vdir
dash           lowntfs-3g     pwd            wdctl
date           ls             rbash          which
dd             lsblk          readlink       ypdomainname
df             lsmod          rm             zcat
dir            mkdir          rmdir          zcmp
dmesg          mknod          run-parts      zdiff
dnsdomainname  mktemp         sed            zegrep
domainname     more           setfont        zfgrep
dumpkeys       mount          setupcon       zforce
echo           mountpoint     sh             zgrep
egrep          mt             sleep          zless
false          mt-gnu         ss             zmore
fgconsole      mv             stty           znew
fgrep          netstat        su
findmnt        networkctl     sync
fuser          nisdomainname  systemctl
root@lubancat:~#
```

ターミナルで、「which」コマンドを使用すると、特定のコマンドのパスを確認できます。たとえば、ls、pwd、cd コマンドの場合は以下のようになります：

株式会社日昇テクノロジー

```
1 # 以下の説明で、「#」記号はコメントを意味し、その後の内容はターミナルに入力しないでください

2

3 which ls # 使用している ls コマンドのパスを確認

4 which pwd # 使用している pwd コマンドのパスを確認

5 which sudo # 使用している sudo コマンドのパスを確認

6 which cd # 使用している cd コマンドのパスを確認
```

```
root@lubancat:~# which ls
/bin/ls
root@lubancat:~# which pwd
/bin/pwd
root@lubancat:~# which sudo
/usr/bin/sudo
root@lubancat:~# which cd
root@lubancat:~#
```

上図から分かるように、ls コマンドは実際に/bin/ls プログラムであり、pwd コマンドは実際に/bin/pwd プログラムであり、「which cd」コマンドには出力がないのは、cd コマンドが Shell の内部プログラムだからです。

Shell で「ls」コマンドを入力するのと「/bin/ls」を入力するのは同じ効果があります。Shell は予め設定されたディレクトリでプログラムを探し、見つかればコマンドオプションとパラメーターを使ってそのプログラムを実行し、プログラムの出力を表示します。この予め設定されたディレクトリは、以下のコマンドで確認できます：

```
1 echo $PATH # echo は出力コマンドで、"$PATH"は環境変数で、その変数の内容を出力します
```

```
root@lubancat:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
root@lubancat:~#
```

「$PATH」はターミナルが使用するパス環境変数で、「:」で区切られています。Shell はこれらのパスでコマンドプログラムを探します。ここで、/bin パスが含まれていることがわかります。つまり、「$PATH」環境変数は、完全なパスを入力する手間を省いてくれますが、コマンドの本質はほとんどがファイルシステム内のアプリケーションです。

株式会社日昇テクノロジー

7.5 一般的なコマンド

以下に一般的なコマンドの説明をリストアップし、それらを実際にターミナルで試してみてください。

7.5.1 pwd コマンド

現在のディレクトリの場所を表示します。

```
pwd
```

以下のように表示されます。

```
cat@lubancat:~$ pwd
/home/cat
cat@lubancat:~$
```

7.5.2 cd コマンド

cd コマンドは change directory の略で、ターミナルの現在のパスを目的のパスに変更します。このコマンドの形式は以下の通りです：

```
cd [ディレクトリ名]
```

形式説明の「ディレクトリ名」は、移動先のパスです。ディレクトリ名が省略された場合は、現在のユーザーのホームディレクトリに移動します。

特定のアプリケーションを除き、Linux に内蔵されているすべてのコマンドは、ターミナルで実行する際に、ディレクトリ名に絶対パスまたは相対パスのどちらも使用できます。

例えば、現在のターミナルのパスが「/home/embedfire」である場合、以下の二つのコマンドは同等です：

```
1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください
2. # 現在のディレクトリが/home/embedfire である場合
3.
4. cd test
5. cd /home/embedfire/test
```

株式会社日昇テクノロジー

これらはいずれも「/home/embedfire/test」ディレクトリに移動することを意味します。ここでの「test」は相対パスで、「/home/embedfire/test」は絶対パスです。

特に、以下のディレクトリリスト現を理解する必要があります：

- 「~」：現在のユーザーのホームディレクトリを示します。
- 「.」：ピリオド 1 つ、現在のディレクトリを示します。
- 「..」：ピリオド 2 つ、現在のディレクトリの一つ上のディレクトリを示します。
- 「/」：スラッシュ、ルートディレクトリを示します。
- 「-」：ハイフン、ディレクトリではありませんが、cd コマンドのパラメータとして使用すると、前回の cd コマンドで移動する前のディレクトリに戻ることができます。

以下のコマンドを実行して、ディレクトリの移動を試してみましょう。ここでの embedfire は現在のユーザー名ですが、実際にはご自身のユーザー名に合わせてください。

```
1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください
2. cd / # ルートディレクトリに移動
3. ls # 現在のディレクトリの内容をリストアップ（ルートディレクトリ）
4. cd - # 前回のディレクトリに戻る（ホームディレクトリ）
5. cd .. # 上のディレクトリに移動
6. ls # 現在のディレクトリの内容をリストアップ（ホームディレクトリ）
7. cd /home/cat/ # cat のホームディレクトリに移動
```

コマンド名とディレクトリ名の間にはスペースが必要です。今後使用するすべてのコマンドで、コマンド名、コマンドオプション、そしてコマンドパラメータの間にはスペースを空ける必要があります。

株式会社日昇テクノロジー

```
root@lubancat:~# cd /
root@lubancat:/# ls
bin   dev   lib         media  opt           root  sdcard  system  userdata
boot  etc   lost+found  mnt    proc          run   srv     tmp     usr
data  home  md5sum.txt  oem    rockchip-test sbin  sys     udisk   var
root@lubancat:/# cd -
/root
root@lubancat:~# cd ..
root@lubancat:/# ls
bin   dev   lib         media  opt           root  sdcard  system  userdata
boot  etc   lost+found  mnt    proc          run   srv     tmp     usr
data  home  md5sum.txt  oem    rockchip-test sbin  sys     udisk   var
root@lubancat:/# cd /home/cat/
root@lubancat:/home/cat#
```

上記のコマンドでルートディレクトリに移動した後、コマンドラインプロンプトのディレクトリが「/」になり、直接 ls を使用して現在のファイルをリストアップし、最後に「-」パラメータを使用してユーザーのホームディレクトリに戻ります。

7.5.3 mkdir コマンド

mkdir コマンドは、make directory の略で、ファイルシステムに新しいディレクトリを作成します。そのコマンド形式は以下の通りです：

```
mkdir [-p] ディレクトリ名
```

コマンド形式の「ディレクトリ名」は作成するディレクトリのパスです。「-p」オプションは入力しなくても良いですが、「-p」オプションを使用すると、作成しようとしているディレクトリの親ディレクトリが存在しない場合、存在しないすべてのディレクトリを自動的に作成します。

以下のコマンドを実行してみましょう：

```
1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください
2. ls # 現在のディレクトリの内容をリストアップし、この時点で testdir ディレクトリは存在しません
3. mkdir testdir # testdir ディレクトリを作成
4. ls # 現在のディレクトリの内容をリストアップし、testdir ディレクトリが追加されたことを確認
5. mkdir other/test # other/test ディレクトリを作成しようとすると、other が存在しないためエラー
6. mkdir -p other/test # -p オプションを使用して other/test ディレクトリを作成
```

株式会社日昇テクノロジー

7. ls # 現在のディレクトリの内容をリストアップし、other ディレクトリが追加されたことを確認

8. ls other # other ディレクトリの内容をリストアップし、test ディレクトリが含まれていることを確認

```
root@lubancat:~# ls
abc.c  embedfire  hello
root@lubancat:~# mkdir testdir
root@lubancat:~# ls
abc.c  embedfire  hello  testdir
root@lubancat:~# mkdir other/test
mkdir: cannot create directory ‘other/test’: No such file or directory
root@lubancat:~# mkdir -p other/test
root@lubancat:~# ls
abc.c  embedfire  hello  other  testdir
root@lubancat:~# ls other
test
root@lubancat:~#
```

7.5.4 touch コマンド

touch コマンドは、存在しないファイルを作成したり、パラメータを通じてディレクトリやファイルの日付と時間を変更することができます。つまり、ファイルに「触れて」その時間を更新することです。

そのコマンド形式は以下の通りです：

touch ファイル名

以下のコマンドを実行してみましょう：

1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください

2. touch helloworld # 現在のディレクトリに helloworld という名前のファイルを作成

3. ls # 現在のディレクトリの内容を表示

7.5.5 ls コマンド

ls コマンドは、list の略で、指定されたディレクトリ下の内容（ファイルおよびサブディレクトリ）を表示する Linux で最もよく使用されるコマンドの一つです。また、ファイルのサイズや変更日などの情報を表示することもできます。

ls コマンドの形式は以下の通りです：

ls [オプション] [ディレクトリ]

「ディレクトリ」パラメータを省略すると、現在のディレクトリの内容をリストアップします。

株式会社日昇テクノロジー

基本的な表示に加えて、ls コマンドには以下のようなよく使われるオプションがあります：

- -a：すべてのファイルとディレクトリを表示します（ファイル名やディレクトリ名が「.」で始まる隠しファイルも含む
- -l：これは小文字の L です。ファイル名以外に、ファイルタイプ、権限、所有者、ファイルサイズなどの情報を詳細にリストアップします。
- -t：ファイルを作成時間の新しい順にリストアップします。
- -A：-a と同様ですが、"."（現在のディレクトリ）および".."（親ディレクトリ）はリストアップしません。
- -R：ディレクトリ下にファイルがある場合、そのディレクトリ下のファイルもリストアップします。つまり、再帰的に表示します。

以下のコマンドを実行してみましょう：

```
1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください
2. ls # 現在のディレクトリの内容をリストアップ
3. ls -a # 現在のディレクトリ下のすべての内容を表示
4. ls -A # 現在のディレクトリ下の"."および".."を除くすべての内容を表示
```

```
root@lubancat:~# ls
abc.c  embedfire  hello  helloworld  other  testdir
root@lubancat:~# ls -a
.   .bash_history  .cache   .gnupg   abc.c      hello       other
..  .bashrc        .config  .profile embedfire  helloworld  testdir
root@lubancat:~# ls -A
.bash_history  .cache   .gnupg   abc.c      hello       other
.bashrc        .config  .profile embedfire  helloworld  testdir
root@lubancat:~#
```

デフォルトの ls コマンドでは、隠しファイルやフォルダーを除く非隠しファイルやフォルダーのみをリストアップします。しかし、ls -a または ls -A コマンドを使用すると、隠しファイルを含むすべてのファイルが表示されます。Linux では、"."で始まるファイル名はシステムによって隠しファイルとして扱われます。ls -a と ls -A の違いは、"."と".."が表示されるかどうかです。

また、ターミナルで表示されるファイルは白色のフォントで、フォルダーは淡い青色のフォントで表示さ

株式会社日昇テクノロジー

れ、ファイルとフォルダーを区別します。

特に、ファイルの詳細情報を確認する必要がある場合は、「-l」オプションを使用できます。以下のコマンドを実行してみましょう：

```
ls -l
```

これにより、現在のディレクトリ下のすべてのファイルが占める総スペースと 7 つのフィールドからなるリストが表示されます。

```
root@lubancat:~# ls -l
total 8
-rw-r--r-- 1 root root    0 Feb 14 18:21 abc.c
-rw-r--r-- 1 root root    0 Feb 14 18:41 embedfire
-rw-r--r-- 1 root root    0 Feb 14 18:21 hello
-rw-r--r-- 1 root root    0 Feb 14 19:35 helloworld
drwxr-xr-x 3 root root 4096 Feb 14 19:28 other
drwxr-xr-x 2 root root 4096 Feb 14 19:28 testdir
root@lubancat:~#
```

7.5.6 cat コマンド

cat コマンドは concatenate の略で、内容を連結することを意味します。通常、ファイルの内容をターミナルで表示するために使用されます。

コマンドの形式は以下の通りです：

```
cat ファイル名
```

例えば、/etc/hostname の内容をターミナルに表示させることができます：

```
cat /etc/hostname
```

```
root@lubancat:~# cat /etc/hostname
lubancat
root@lubancat:~#
```

7.5.7 echo コマンド

echo コマンドは、ターミナル上にテキストを表示したり、ターミナルの変数内容を表示するために使用されます。

コマンドの形式は以下の通りです：

株式会社日昇テクノロジー

```
1. echo "文字列"

2. echo 文字列

3. echo $変数名
```

echo コマンドを使用する際、ダブルクオーテーションを使用しても使用しなくても効果は同じですが、クオーテーションを使用する場合は英語の記号を使用することに注意してください。

以下のコマンドを試してみましょう：

```
1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください

2. echo "test" # 文字列 test を出力

3. echo test # 文字列 test を出力

4. echo $PATH # 環境変数 PATH を出力

5. echo "$PATH" # 環境変数 PATH を出力
```

```
root@lubancat:~# echo "test"
test
root@lubancat:~# echo test
test
root@lubancat:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
root@lubancat:~# echo "$PATH"
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
root@lubancat:~#
```

7.5.8 ファイルへの出力リダイレクト

上記で見たように、コマンドの実行結果は通常、ターミナルに表示されますが、時にはコマンドの実行結果をファイルに保存して分析したい場合があります。その場合は、「>」または「>>」のリダイレクト演算子を使用してファイルへ出力をリダイレクトできます。「>」は出力で元のファイルを直接上書きし、「>>」は出力を元のファイルの末尾に追加します。

使用方法は以下の通りです：

```
1. コマンド > ファイル名

2. コマンド >> ファイル名
```

株式会社日昇テクノロジー

ファイルが存在しない場合は自動的に作成されます。

以下のコマンドを実行してみましょう：

1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください

2. echo test > file.txt # echo test の出力を file.txt ファイルにリダイレクト

3. cat file.txt # file.txt ファイルの内容を確認

4. echo abc > file.txt # echo abc の出力を file.txt ファイルにリダイレクト

5. cat file.txt # file.txt ファイルの内容を確認

6. echo 123456 >> file.txt # echo 123456 の出力を file.txt ファイルに追加でリダイレクト

7. cat file.txt # file.txt ファイルの内容を確認

8. ls > file.txt # ls コマンドの出力を file.txt ファイルにリダイレクト

9. cat file.txt # file.txt ファイルの内容を確認

```
root@lubancat:~# echo test > file.txt
root@lubancat:~# cat file.txt
test
root@lubancat:~# echo abc > file.txt
root@lubancat:~# cat file.txt
abc
root@lubancat:~# echo 123456 >> file.txt
root@lubancat:~# cat file.txt
abc
123456
root@lubancat:~# ls > file.txt
root@lubancat:~# cat file.txt
abc.c
embedfire
file.txt
hello
helloworld
other
testdir
root@lubancat:~#
```

出力リダイレクトを行った後、結果はターミナルに表示されず、指定したファイルに保存されます。

7.5.9 rm コマンド

rm コマンドは remove の略で、一つまたは複数のファイルやディレクトリを削除する機能を持っています。

そのコマンド形式は以下の通りです：

株式会社日昇テクノロジー

1. rm [オプション] 一つまたは複数のファイル/ディレクトリ名

rm コマンドで削除すると、ファイルは直接永久に削除され、ゴミ箱に移動して再確認することはありません。そのため、rm コマンドを使用する前には、よく考える必要があります。特にシステム管理者アカウントを使用している場合や、後で紹介される sudo コマンドでシステム権限を取得している場合は、一つの コマンドで大きな被害を引き起こす可能性があります。

サポートされているオプションは以下の通りです：

- -i：ファイルまたはディレクトリを削除する前に、一つずつ確認を求めます。
- -r：ディレクトリとその中に含まれるサブディレクトリやファイルをすべて削除します。
- -f：存在しないファイルの警告を無視し、一つずつの確認を求めません。

rm コマンドを誤って使用すると重大な結果を招く可能性があるため、注意が必要ですが、システムが新しくインストールされたばかりで重要な内容がない場合や、ホームディレクトリで操作を行う場合は、システムの正常な動作に影響を与えることはありませんので、以下のコマンドを実行して実験してみることができます。

まず、後で削除するためのテストファイルをいくつか作成します。

```
1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください
2. mkdir -p ABC/test # ABC/test ディレクトリを作成
3. ls # 内容をリストアップ
```

```
1. touch a.txt b.txt c.txt d.txt # 複数のテストファイルを作成
2. ls # 内容をリストアップ
```

現在のディレクトリに ABC ディレクトリを作成し、その中に test サブディレクトリと a.txt、b.txt、c.txt、d.txt ファイルを作成します。

株式会社日昇テクノロジー

```
root@lubancat:~# mkdir -p ABC/test
root@lubancat:~# ls
ABC
root@lubancat:~# touch a.txt b.txt c.txt d.txt
root@lubancat:~# ls
ABC  a.txt  b.txt  c.txt  d.txt
root@lubancat:~#
```

7.5.9.1 ファイル削除テスト

```
1. # 以下の説明で、「#」はコメントを意味し、その後の内容はターミナルに入力しないでください

2. rm a.txt b.txt # a.txt と b.txt を直接削除

3. rm -i c.txt d.txt # c.txt と d.txt を削除する前に確認
```

上記のいずれかのコマンドを実行することで、ファイルを削除することができます。パラメータ i を使用すると、ファイルを削除するかどうかを尋ねられます。削除する場合は y を、しない場合は n を入力します。以下の例では、a.txt と b.txt を直接削除し、c.txt を削除して d.txt を残しました。

```
root@lubancat:~# ls
ABC  a.txt  b.txt  c.txt  d.txt
root@lubancat:~# rm a.txt b.txt
root@lubancat:~# ls
ABC  c.txt  d.txt
root@lubancat:~# rm -i c.txt d.txt
rm: remove regular empty file 'c.txt'? y
rm: remove regular empty file 'd.txt'? y
root@lubancat:~# ls
ABC
root@lubancat:~#
```

7.5.9.2 ディレクトリの削除

rm コマンドに「-r」オプションを使用することで、以下のようにディレクトリを削除できます：

```
rm -r ABC/
```

```
root@lubancat:~# ls
ABC
root@lubancat:~# rm -r ABC/
root@lubancat:~# ls
root@lubancat:~#
```

このコマンドを実行すると、ABC ディレクトリとそのサブディレクトリ test、および ABC ディレクトリ内のすべての内容が削除されます。そのため、誤ってファイルを削除しないように、rm コマンドを使用する際は可能な限り i オプションを使用することをお勧めします。

株式会社日昇テクノロジー

7.5.9.3 -f オプションの役割

以下の二つのコマンドを実行して、結果の違いを比較してみましょう：

```
1. rm a.txt

2. rm -f a.txt
```

```
root@lubancat:~# rm a.txt
rm: cannot remove 'a.txt': No such file or directory
root@lubancat:~# rm -f a.txt
root@lubancat:~#
```

最初のコマンドを使用すると、「ファイルが見つからない」というエラーが表示されますが、-f オプションを使用すると、そのようなエラーメッセージは無視されます。このオプションは、日常使用では慎重に使用する必要があります。

7.5.10 su コマンド

su コマンドを使用してユーザーを切り替えることができます：

```
1. su + ユーザー名

2. そのユーザーのパスワードを入力すると、ユーザーが切り替わります。
```

7.5.11 sudo コマンド

sudo コマンドは、switch user do の略で、特定の操作を行うためにユーザーを切り替えることを意味します。Linux には root というスーパーユーザーがおり、何でもできますが、その権限を誤って使用するとシステムがクラッシュする可能性があります。しかし、多くの場合、ソフトウェアのインストールなど、root 権限が必要な作業があります。そのような場合には、sudo コマンドを使用して、現在のユーザーに root 権限を与え、後続のコマンドを実行することができます。

sudo コマンドの形式は以下の通りです：

```
sudo コマンド
```

sudo コマンドを使用すると、現在のユーザーのパスワードを入力するよう求められますが、root ユーザ

株式会社日昇テクノロジー

ーのパスワードではありません。パスワードはターミナルに表示されず、入力しても星印（*）は表示されません。信頼してキーボードにパスワードを入力し、Enter キーを押してください。

以下のコマンドを実行して、root 権限が必要なディレクトリで操作を試みましょう：

```
root ユーザーログインであれば cat ユーザーに切り替えて再試行できます
su cat # cat ユーザーに切り替え
cd # ホームディレクトリに移動
```

```
root@lubancat:~# su cat
cat@lubancat:/root$ cd
cat@lubancat:~$ ls
Desktop
cat@lubancat:~$ pwd
/home/cat
cat@lubancat:~$
```

```
cd /home # /home ディレクトリに移動
touch test.txt # 現在のディレクトリに test.txt ファイルを作成しようと試み
sudo touch test.txt # sudo を使用して権限を高め、test.txt ファイルを作成
# パスワードを入力しても (*)が表示されないので、入力してから enter キーを押してください
# パスワードを入力しても (*)が表示されないので、入力してから enter キーを押してください
# パスワードを入力しても (*)が表示されないので、入力してから enter キーを押してください
```

rm test.txt # test.txt ファイルを削除しようと試みる

sudo rm test.txt # sudo を使用して権限を高め、test.txt ファイルを削除

```
cat@lubancat:~$ cd /home/
cat@lubancat:/home$ touch test.txt
touch: cannot touch 'test.txt': Permission denied
cat@lubancat:/home$ sudo touch test.txt
cat@lubancat:/home$ ls
cat  test.txt
cat@lubancat:/home$ rm test.txt
rm: remove write-protected regular empty file 'test.txt'? y
rm: cannot remove 'test.txt': Permission denied
cat@lubancat:/home$ sudo rm test.txt
cat@lubancat:/home$ ls
cat
cat@lubancat:/home$
```

上記の例では、cat ユーザーは/home ディレクトリにファイルを作成する権限がないため、直接 touch コマンドを使用してファイルを作成しようとすると、権限不足のメッセージが表示されます。sudo コマン

株式会社日昇テクノロジー

ドを使用してパスワードを入力すると、ファイルが正常に作成されます。同様に、rm コマンドを直接使用すると権限不足が表示されますが、sudo を使用するとファイルを削除できます。そして、パスワードを再度入力する必要はありません。

上記のコマンドを実行すると、権限がないために「test.txt」ファイルの作成に失敗しますが、「sudo !!」コマンドを使って、直前のコマンドを sudo 権限で再実行することができます。

#後の説明では、番号はコメントをリストし、その後ろの内容は端末に入力しないでください

```
cd/home        #/home ディレクトリに切り替え
touch test.txt    #権限なしを表示
sudo !!            # sudo は!を 2 つ付けて、sudo 権限を再使用して前のコマンドを実行します
```

```
cat@lubancat:~$ cd /home
cat@lubancat:/home$ touch test.txt
touch: cannot touch 'test.txt': Permission denied
cat@lubancat:/home$ sudo !!
sudo touch test.txt
cat@lubancat:/home$ ls
cat  test.txt
cat@lubancat:/home$
```

7.5.12 clear コマンド

ターミナルで様々なコマンドを実行した後、画面上に前の内容が残っていると、見づらくなることがあります。その場合は、「clear」コマンドを実行して画面をクリアすることができます。

```
1. clear
```

7.5.13 reboot/poweroff コマンド

ターミナルからシステムの再起動やシャットダウンを行うには、「reboot」や「poweroff」コマンドを使用します。

```
1. reboot
2. poweroff
```

株式会社日昇テクノロジー

第 8 章 ユーザーとファイル

8.1 ユーザーとユーザーグループ

Linux オペレーティングシステムもまた、他のユーザーやコンピュータの全リソースを管理できるユーザーを含む、マルチユーザーシステムです。これを root ユーザーと呼びます。

Android システムは実際には Linux に基づいており、Android 携帯でよく言及される root 権限も、最高の権限を得ることと同じです。

Linux では、各ユーザーには UID（ユーザー ID）が割り当てられ、システムユーザーを識別します。Linux は UID0 を root ユーザーに割り当て、各ユーザーに異なる権限を付与することができます。そのため、各ユーザーの操作は異なります。

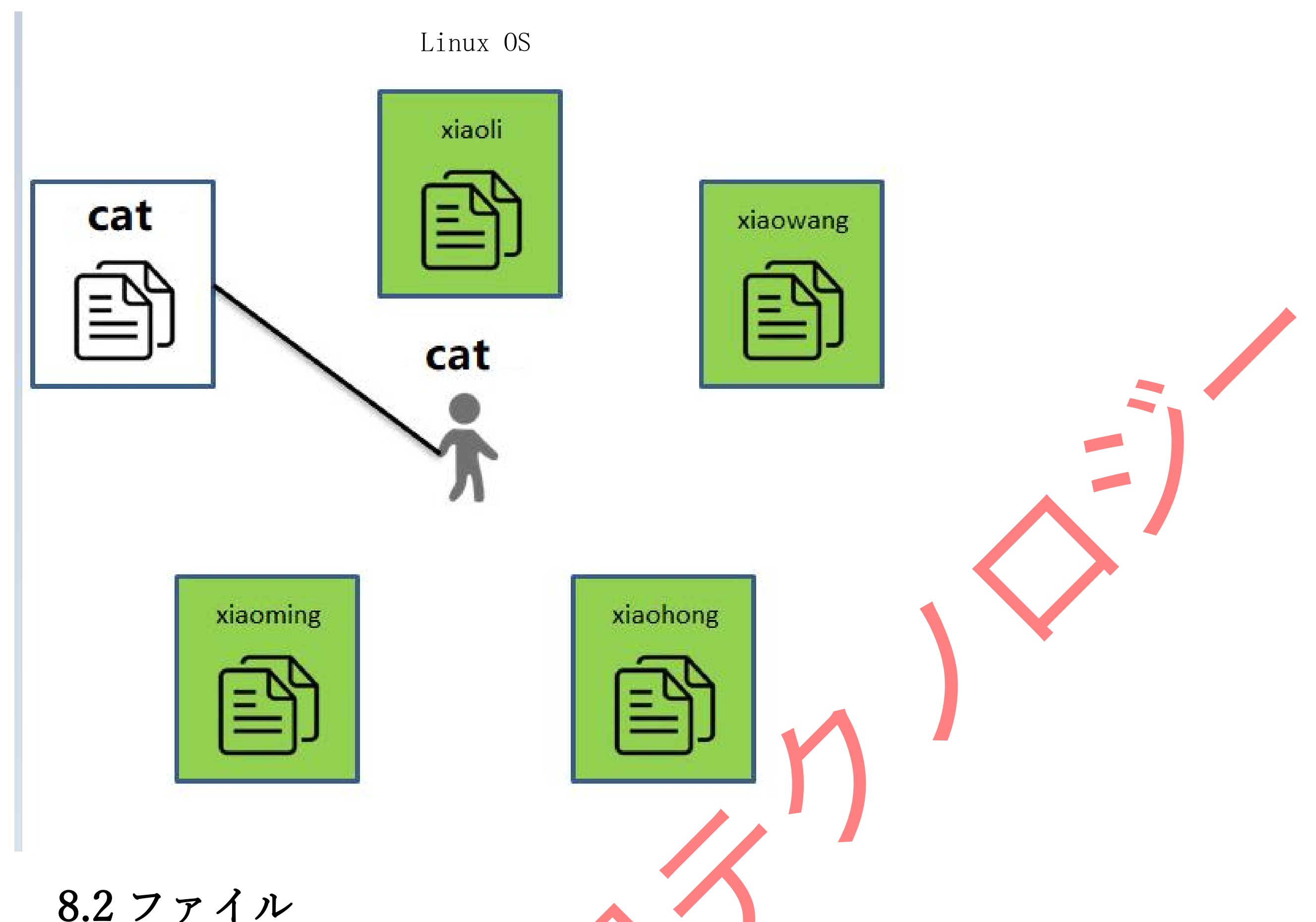
「id」コマンドを使用して、現在のユーザーの UID を確認できます（コマンドの使用方法は次の章で紹介しますが、ここでは理解しておきましょう）：

```
id
```

```
cat@lubancat:~$ id
uid=1000(cat) gid=1000(cat) groups=1000(cat),4(adm),20(dialout),24(cdrom),27(sudo),
29(audio),30(dip),44(video),46(plugdev),50(staff),101(systemd-journal),109(bluetoot
h),110(netdev),116(pulse-access)
cat@lubancat:~$
```

上記の gid は、現在のユーザーが所属しているグループ（Group）を識別するために使用されます。各ユーザーは複数のグループに対応することができます。これは、学校に音楽クラブ、アニメクラブ、文学クラブなどのサークルがあり、各生徒が複数のサークルに参加できるように、さまざまな興味深いものに触れることができるのと同じです。Linux システムには複数のグループがあり、各ユーザーグループはサークルに相当します。ユーザーが複数のユーザーグループのメンバーである場合、他のグループに対応するファイルにアクセスできますが、そのグループのファイルが他のユーザーのアクセスを許可している場合に限ります。これには、別の知識点であるファイル権限について理解する必要があります。

株式会社日昇テクノロジー



8.2 ファイル

Linux ではすべてがファイルとして扱われます。

Linux のファイル属性には、読み取り権限、書き込み権限、実行権限があります。実行権限とは、メモリにロードしてオペレーティングシステムのローダーによって実行できるファイルを指します。Windows オペレーティングシステムでは、最も一般的に接触するのは.exe ファイルの拡張子を持つファイルです。しかし、Linux ではファイルタイプを拡張子で識別するわけではなく、特定の実行ファイルを実行するには実行権限を付与する必要があります。

さらに、Linux ファイル権限は、ファイルの所有者（owner）、グループメンバー（groups）、その他のグループメンバー（other）の 3 種類に分けられます。例えば、ファイルの所有者である cat は、そのファイルを読み書きする権限がありますが、cat グループおよび他のグループのユーザーは、そのファイルを読むことはできますが、変更することはできません。他のグループのファイルを変更したい場合は、そのファイルグループの権限を変更する必要があります。

株式会社日昇テクノロジー

cat ユーザーとしてシステムにログインし、新しいファイルを作成してみましょう。

```
# root ユーザーの場合は普通のユーザーに切り替えます

su cat

# パスワードを入力

# ディレクトリを切り替え

cd /home/cat

# ファイルの権限を確認

ls -l
```

```
cat@lubancat:~$ touch helloworld
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat cat 4096 Feb 14  2019 Desktop
-rw-r--r-- 1 cat cat    0 Nov 15 14:03 helloworld
cat@lubancat:~$
```

例えば、Desktop ディレクトリには以下のような権限が設定されています。

- drwxr-xr-x 2 cat cat 4096 Feb 14 2019 Desktop

各フィールドの説明は以下の通りです：

- 第一フィールド：ファイル属性

ファイル属性は合計 10 文字で、最初の文字はファイルタイプをリストします。"-"は通常のファイルを、"d"はディレクトリを、"l"はリンクファイルを示します。

残りの 9 文字は 3 文字ごとに 1 セットとなり、ファイルの所有者、グループ、その他のユーザーの権限をリストします。"r"は読み取り（read）権限、"w"は書き込み（write）権限、"x"は実行（execute）権限を意味します。

- 第二フィールド：リンクが使用するノード/サブディレクトリの数

第二フィールドは、ファイルのタイプに依存してその意味が異なります。ファイルである場合は、そのファイルが持っているハードリンクの数を示します。あるファイルの第二フィールドが 1 である場合、そのファイルへの他のハードリンクがないことを意味します。

株式会社日昇テクノロジー

Linux システムでファイルを保存する際の特徴として、リンクはハードリンクとシンボリックリンク（ソフトリンク）の二種類に分けられます。シンボリックリンクは、Windows オペレーティングシステムのショートカットに似ています。一方、ハードリンクは、ファイルのコピーを作成し、自動的に更新されるようなものです。ハードリンクファイルの内容を変更すると、元のファイルも変更されます。ln コマンドを使用してシンボリックリンクとハードリンクを作成できますが、ここではその概念を深く考える必要はありません。

ディレクトリの場合、第二フィールドはそのディレクトリ内のサブディレクトリの数を示します。空のディレクトリであれば、その値はデフォルトで 2 になります。これは、「.」と「..」のサブディレクトリが含まれているためです。

- 第三フィールドと第四フィールド：ファイルの所有者とそのグループ

ファイルの所有者が全ての実行権限を持ち、他のユーザーや同じグループのユーザーは読み取り権限のみを持っています（スーパーユーザーを除く）。

このフォルダの所有者、つまり現在ログインしているユーザーが全ての実行権限を持っていることがわかりますが、他のユーザーや同じグループに属するユーザーは読み取り権限のみを持っています（スーパーユーザーを除く）。スーパーユーザーによって作成されたファイルも同様です。

株式会社日昇テクノロジー

```
1 # スーパーユーザーでファイルを作成

2 sudo touch hello



3 # ファイル属性を確認

4 ls -l
```

```
cat@lubancat:~$ sudo touch hello
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rw-r--r-- 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$
```

cat ユーザーが読み取り権限のみを持っていることがわかります。

```
1 # ファイルを見る

2 cat hello



3 # ファイルに書き込む

4 echo cat > hello
```

```
cat@lubancat:~$ cat hello
cat@lubancat:~$ echo cat > hello
-bash: hello: Permission denied
cat@lubancat:~$
```

・読み取りは正常に行われますが、内容がないため他の内容は表示されません。

・書き込みエラーが発生し、内容を書き込む権限がないことがわかります。

8.3 chmod コマンド

このコマンドはファイルの権限を変更するために使用されます。ファイルの権限は 3 つの部分で構成されており、各部分は読み取り、書き込み、実行の 3 つの権限に対応しています。

rwx は 2 進数で以下のようにリストされます：

- r:100–4

- w:010–2

株式会社日昇テクノロジー

- x:001–1

そのため、ファイルの権限が rwx であれば、その数値リスト現は 7、rw であれば 6 になります。

ファイルの全権限を変更するには以下のコマンドを使用します：

```
#すべての権限の変更

sudo chmod xxx ファイル
```

例えば、xxx=777 の場合、ファイルの権限は-rwxrwxrwx になります。

```
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rw-r--r-- 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$ sudo chmod 777 hello
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rwxrwxrwx 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$
```

xxx=666 の場合、ファイルの権限は-rw-rw-rw-になります。

```
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rwxrwxrwx 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$ sudo chmod 666 hello
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rw-rw-rw- 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$
```

chmod には、ユーザー、グループ、その他のユーザーに個別に権限を設定する方法もあります：

```
#権限を個別に追加・削除する

sudo chmod [ugoa][+-][rwx] ファイル
```

[ugoa]

- u: ユーザー

- g: ユーザーのグループ

- o: その他のユーザー

- a: 全ユーザー

[+-]

株式会社日昇テクノロジー

- +: 権限を追加

- -: 権限を削除

[rwx]

- r: 読み取り

- w: 書き込み

- x: 実行

例：

```
#全ユーザーから実行権限を削除

 sudo chmod a-x hello



# その他のユーザーから書き込み権限を削除

 sudo chmod o-w hello



# ユーザーから書き込み権限を削除

 sudo chmod u-w hello



# ユーザーに書き込み権限を追加

 sudo chmod u+w hello
```

株式会社日昇テクノロジー

```
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rwxrwxrwx 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$ sudo chmod a-x hello
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rw-rw-rw- 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$ sudo chmod o-w hello
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rw-rw-r-- 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$ sudo chmod u-w hello
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-r--rw-r-- 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$ sudo chmod u+w hello
cat@lubancat:~$ ls -l
total 4
drwxr-xr-x 2 cat  cat  4096 Feb 14  2019 Desktop
-rw-rw-r-- 1 root root    0 Nov 15 14:43 hello
cat@lubancat:~$
```

第 9 章 Linux ファイルディレクトリ

Windows から Ubuntu への移行者にとって、最も慣れない点の一つがファイルディレクトリの違いかもしれません。しかし、実際には多くの共通点があり、比較して学ぶことができます。

注意：Linux ではすべてがファイルです。Linux を学ぶ上で忘れてはならないのは、ファイルに関連する内容です。

9.1 ホームディレクトリ

- 一般ユーザー（cat）で LubanCat-RK のボードにログインすると、ホームディレクトリ/home/cat に入ります。

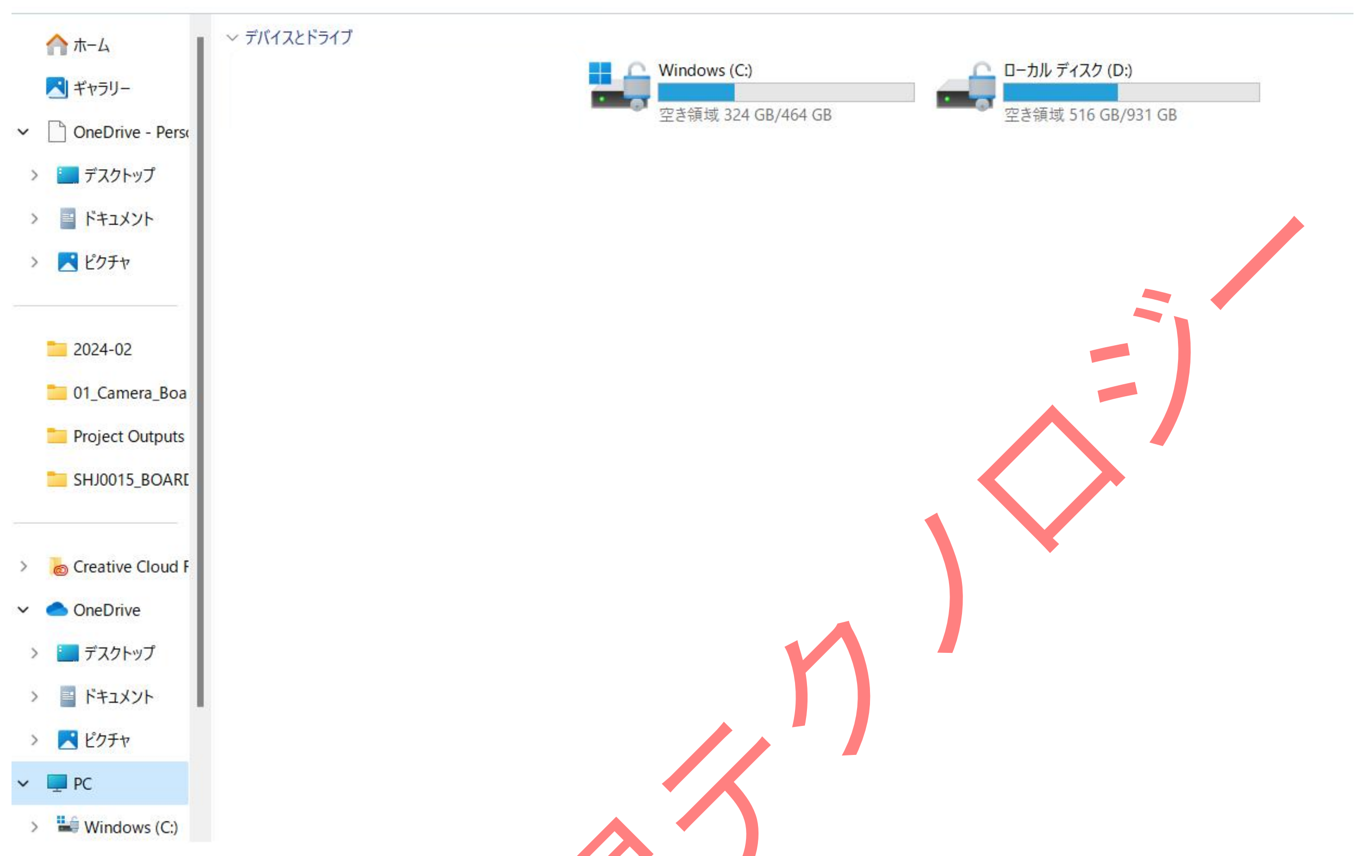
- root ユーザーでログインすると、/root ディレクトリに入ります。root ユーザーのホームディレクトリは/root です。

- cat ユーザーのホームディレクトリには、デスクトップシステムで使用されるフォルダーである Desktop のみが含まれます。

```
cat@lubancat:~$ pwd
/home/cat
cat@lubancat:~$ ls
Desktop
cat@lubancat:~$
```

株式会社日昇テクノロジー

windows のファイルストレージ



Windows と Debian（Ubuntu）では、ユーザーが個人ディレクトリにコンテンツを保存することを期待しています。これは、システムのコアファイルと分離するためです。Windows の使用習慣に従って、Debian で同様の習慣を試みると、多くのディスクがリストされず、すべてのコンテンツがルートディレクトリ下にあることがわかります。

しかし、Windows を使用する多くの人々の習慣に従えば、通常は異なる内容を保存するために追加のディスクを作成します。たとえば、ソフトウェア、作業資料、ゲーム、ビデオなどのための D、E、F、G ドライブです。しかし、Debian でこの習慣を試みると、多くのディスクがリストアップされていないことがわかります。そのすべての内容はルートディレクトリの下に位置しています。

9.2 ルートディレクトリ

ルートディレクトリは、Linux のすべてのファイルとディレクトリの起点であり、パスはスラッシュ「/」でリストされます。

株式会社日昇テクノロジー

以下のコマンドでルートディレクトリ下のファイルを確認できます。

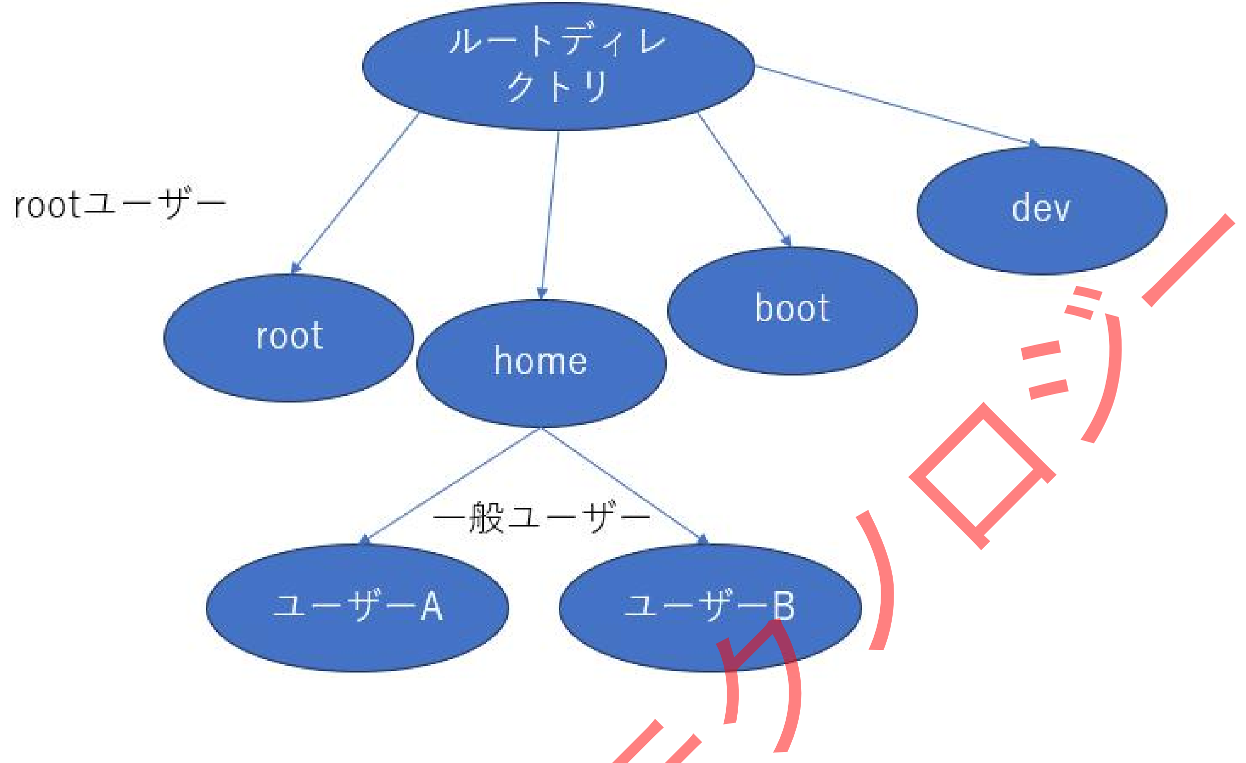
```
ls /
```

```
root@lubancat:/# ls /
bin   dev   lib         media  opt           root  sdcard  system  userdata
boot  etc   lost+found  mnt    proc          run   srv     tmp     usr
data  home  md5sum.txt  oem    rockchip-test sbin  sys     udisk   var
root@lubancat:/#
```

上記の画像では、これらのファイルやフォルダーがすべて「/」ディレクトリ下にあることがわかります。そして、それらのパスは「/bin、/boot、/dev、/opt、/root」などです。

特に、前に説明したホームディレクトリ（個人ディレクトリ）のパスは「/home/ユーザー名/」で、下の画像に示す例では、ディレクトリは/home/cat です。つまり、Debian システム下で、普通のユーザーの個人ディレクトリは「/home」下にあり、ユーザー名で命名されます。もしシステム下に複数のユーザー A、B、C がおり、それぞれが個人ディレクトリを持っている場合、それらの個人ディレクトリパスはデフォルトで「/home/A」、「/home/B」、「/home/C」になります。この点で、Windows システムと似ています。Windows システム下では、個人ディレクトリはデフォルトで「C:/Users/A」、「C:/Users/B」、「C:/Users/C」です。

株式会社日昇テクノロジー



Linux 開発者にとっては、ROOT ディレクトリの内容を理解することが必要です。これは多くの人が Linux を不親切だと感じる理由の一つでもあります。/home の個人ディレクトリを除外すると、実質的に Linux の ROOT ディレクトリは Windows システムの C:/Windows ディレクトリと機能が似ており、システムの核心ファイルが含まれています。ただし、Windows ユーザーは通常、システムファイルディレクトリを理解する必要はありません。Linux システムを普通のデスクトップアプリケーションにのみ使用する場合、実際には ROOT ディレクトリの内容を深く理解する必要はありません。

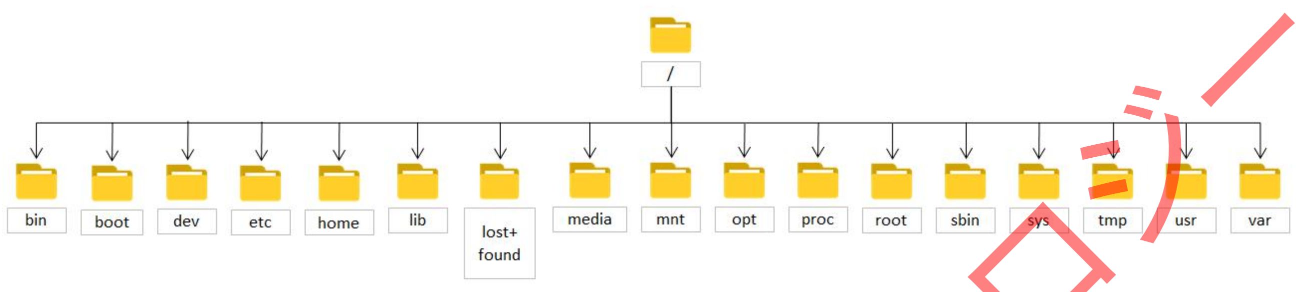
しかし、Linux システムを開発やサーバー用途で利用する際は、一般的にシステムに様々なソフトウェアツールをインストールしたり、システムやツールを設定する必要があります。しばしば、ROOT ディレクトリ下のファイル内容を変更することが避けられません。たとえば、ソフトウェアツールはデフォルトで /usr/bin ディレクトリにインストールされ、ソフトウェアツールの設定ファイルは/etc ディレクトリにあるかもしれません。また、U ディスクや CD を挿入した際に、Windows のように独立したドライブ文字でアクセスできるわけではなく、デフォルトで/media や/mnt ディレクトリ下にマウントされることがわ

株式会社日昇テクノロジー

かります。

9.2.1 ルートディレクトリ構造

Linux システムには、ルートディレクトリから始まる一つのディレクトリツリーのみが存在します。



複数のハードテスクがあるコンピューターでも、Windows の C、D、E ドライブの構造は表示されません。Windows の習慣を強引にコピーする場合、ルートディレクトリ下に C、D、E ディレクトリを作成し、それぞれのディスクを/C、/D、/E ディレクトリにマウントすることができますが、Linux では通常このようにはしません。

ディレクトリツリーはあくまで管理上の概念であり、ハードウェアに直接関連しているわけではありません。例えば、これらのファイルがディスクに保存されるのは、システムのインストール時にルートディレクトリ全体がディスクに「マウント」されるためです。マウントとは、ストレージのパーティションを特定のディレクトリに関連付けることを意味し、そのディレクトリへのアクセスは、関連付けられたストレージパーティションにマッピングされます。これにはディスク、CD、USB ドライブ、NAND FLASH などが含まれます。

異なるディストリビューションではルートディレクトリの解釈が異なる場合がありますが、ほとんどがファイルシステム階層標準（FHS、Filesystem Hierarchy Standard）に従っているため、ルートディレクトリ下の各サブディレクトリの内容と機能は概ね以下のリストの通りです。

株式会社日昇テクノロジー

ディレクトリ	ディレクトリに置かれた内容
bin	システムコマンドを格納するディレクトリ。例えば、cat、cp、mkdir などのコマンド。
boot	起動プロセスに必要な内容を格納するディレクトリ。例えば、ブートマネージャー grub2。
dev	すべてのデバイスファイルのディレクトリ（例: サウンドカード、ハードディスク、CD ドライブ）。
etc	システムの主要な設定ファイル。
home	ユーザーのホームディレクトリデータを格納するディレクトリ
lib	/sbin および/bin ディレクトリ下のコマンドに必要なライブラリファイルを格納するディレクトリ。
Lib32/lib64	32 ビット/64 ビットのバイナリ関数ライブラリを格納するディレクトリ。
Lost+found	EXT3/4 ファイルシステムでは、システムが予期せずクラッシュしたり、予期せずシャットダウンした場合に、このディレクトリ下に断片ファイルが生成されます。システム起動時に fsck ツールがこのディレクトリをチェックし、損傷したファイルを修復します。
media	CD、フロッピー、DVD などのデバイスをマウントするために使用されるディレクトリ。
mnt	/media と同様に、一時的なストレージデバイスをマウントするために使用されるディレクトリ。
opt	サードパーティのソフトウェアのインストールディレクトリ。
proc	プロセスおよびカーネル情報を格納するディレクトリ。ハードディスクスペースを占有しません。
root	root ユーザーのホームディレクトリ。
run	システムが起動してからの情報を格納する一時ファイルシステム。システムが再起動すると、このディレクトリ下のファイルは削除またはクリアされるべきです。
sbin	root ユーザーが使用するコマンドを格納するディレクトリ。
srv	一部のネットワークサービスが必要とするデータファイル。
sys	/proc ディレクトリと同様に、CPU やシステムのハードウェアに関連する情報を記録するために使用されるディレクトリ。
tmp	プログラム実行時に生成される一時ファイルを格納するディレクトリ。
usr	システムプログラムを格納するディレクトリ。Windows の「programefiles」フォルダに似ています。
var	内容が頻繁に変更されるファイル（例: システムログファイル）を格納するディレクトリ。

ここでは、ルートディレクトリ下の内容について詳しく説明します。これらの説明を読みながら、実際に

株式会社日昇テクノロジー

該当するディレクトリを開いて、初めての印象を得てください。

9.2.2 /bin ディレクトリ

/bin ディレクトリは/binary の略で、一般ユーザーが使用可能な多くのコマンドが含まれています。コマンドの本質は実行可能プログラム、つまり二進数の bin ファイルです。Linux コマンドに慣れた後、このディレクトリに戻ってみると、ls、cp、cat などのコマンドがここに見つかることがわかります。

```
root@lubancat:/# ls /bin
 2to3-2.7                              mkzftree
 GET                                   mmcli
 HEAD                                  monitor-sensor
 POST                                  more
 Thunar                                mount
 X                                     mountpoint
 X11                                   mousetweaks
 Xorg                                  mpi_dec_mt_test
'['                                    mpi_dec_multi_test
 aa-enabled                            mpi_dec_test
 aa-exec                               mpi_enc_mt_test
 aarch64-linux-gnu-cpp                 mpi_enc_test
 aarch64-linux-gnu-cpp-9               mpi_rc2_test
 aarch64-linux-gnu-pkg-config          mpp_info_test
 aarch64-linux-gnu-run                 mpv
 aarch64-unknown-linux-gnu-pkg-config  mscompress
 aconnect                              msexpand
 acpi_listen                           mtrace
 add-apt-repository                    mutter
 addpart                               mv
 alsabat                               namei
 alsaloop                              nawk
 alsamixer                             ncal
 alsatplg                              neqn
 alsaucm                               netstat
 amidi                                 networkctl
```

9.2.3 /sbin ディレクトリ

/sbin ディレクトリは/system bin の略で、/bin と機能的に似ていますが、ここには一般的にシステムプログラムが格納されており、管理者権限が必要な場合が多いです。たとえば、システムにロードされているモジュールを表示する lsmod コマンドや、ストレージをフォーマットする mkfs などが含まれます。

株式会社日昇テクノロジー

```
root@lubancat:/# ls /sbin
ModemManager          getpcaps               ntp-wait
NetworkManager        getty                  ntpd
aa-remove-unknown     getweb                 ntptime
aa-status             gnome-menus-blacklist  pam-auth-update
aa-teardown           groupadd               pam_extrausers_chkpwd
accessdb              groupdel               pam_extrausers_update
acpid                 groupmems              pam_getenv
add-shell             groupmod               pam_tally
addgnupghome          grpck                  pam_tally2
addgroup              grpconv                pam_timestamp_check
adduser               grpunconv              paperconfig
agetty                guest-account          parted
alsa                  halt                   partprobe
alsa-info             hwclock                pccardctl
alsabat-test          i2c-stub-from-dump     pivot_root
alsactl               i2cdetect              plipconfig
anacron               i2cdump                plymouthd
apparmor_parser       i2cget                 poweroff
apparmor_status       i2cset                 pppd
```

9.2.4 /etc ディレクトリ

/etc ディレクトリは/etcetera の略で、もともとは雑多な内容を格納する場所として設計されましたが、現在ではシステムの様々な設定ファイルが格納される非常に重要な場所になっています。たとえば、ユーザー情報ファイル/etc/passwd や、システム初期化ファイル/etc/rc などが含まれます。

9.2.5 /root ディレクトリ

/root はスーパーユーザーのホームディレクトリで、Ubuntu ではデフォルトで空です。

9.2.6 /home ディレクトリ

/home は前述のホームディレクトリです。

9.2.7 /lib ディレクトリ

/lib ディレクトリは/library の略で、ルートファイルシステム上のプログラムが必要とする共有ライブラリが格納されています。たとえば、C 言語の標準ライブラリなど、多くのプログラムで共有されるコードが含まれており、各プログラムが同じサブルーチンのコピーを持つ必要がないようにすることで、実行ファイルを小さくし、スペースを節約できます。もしコード A とコード B が C 言語の標準ライブラリである printf 関数と malloc 関数を使用している場合、彼らはファイルシステム上の C ライブラリの内容を共有することができます。そのため、コード A とコード B は自分たちが配布するプログラムパッケージ

株式会社日昇テクノロジー

に C ライブラリのこの部分を追加する必要はありません。この部分については、GCC の章で静的/動的コンパイルについて説明されます。

9.2.8 /dev ディレクトリ

/dev ディレクトリにはデバイスファイルが格納されており、これらのファイルを介してユーザーは外部デバイスにアクセスできます。例えば、/dev/mouse を通じてマウスの入力にアクセスすることができます。

9.2.9 /proc ディレクトリ

/proc ディレクトリには、内核がユーザースペースにエクスポートする内核情報が含まれており、このディレクトリ下のファイルにアクセスすることで、これらの情報を閲覧できます。

9.2.10 /sys ディレクトリ

/proc ディレクトリと同様に、/sys ディレクトリは内核のデバイスドライバをユーザースペースにエクスポートし、/sys ディレクトリおよびその下のファイルにアクセスすることで、内核の一部のドライバやデバイスを確認または制御できます。

9.2.11 /tmp ディレクトリ

/tmp ディレクトリには、プログラムが実行中に生成する一時情報やデータが格納されます。しかし、ブート後にプログラムが実行される場合、より大きなディスクスペースを持つ/var/tmp を/tmp の代わりに使用することが推奨されます。

9.2.12 /boot ディレクトリ

/boot ディレクトリには、ブートローダー（bootstrap loader）が使用するファイルが格納されています。たとえば lilo やカーネルイメージがここに配置されます。多くのカーネルイメージがある場合、このディレクトリが大きくなる可能性があるため、別のファイルシステムに配置する方が良い場合があります。

株式会社日昇テクノロジー

9.2.13 /mnt ディレクトリ

/mnt ディレクトリは、システム管理者がファイルシステムを一時的にマウントするためのマウントポイントです。/mnt の下には、たとえば/mnt/dosa が MS-DOS ファイルシステムのフロッピー、/mnt/exta が ext2 ファイルシステムのフロッピー、/mnt/cdrom が CD-ROM ドライブなど、様々なサブディレクトリがあります。

9.2.14 /media ディレクトリ

/media ディレクトリは、CD や USB ドライブなどの自動マウントされるデバイスのためのディレクトリです。一部のシステムでは、これらのデバイスが自動的にこのディレクトリにマウントされ、対応するデバイスにアクセスできるようになります。

9.2.15 /usr ディレクトリ

かつての/usr はユーザーのホームディレクトリであり、様々なユーザーファイルが保存されていました——現在は/home に置き換えられています（例えば、/usr/someone は/home/someone に変更されました）。現代の/usr は、様々なプログラムとデータを専門に保存するためのもので、ユーザーディレクトリは移動しました。/usr の名前は変わっていませんが、その意味は「ユーザーディレクトリ」から「unix system resource」（Unix システムリソース）に変わりました。注目すべきは、一部の Unix システム上では、/usr/someone をユーザーのホームディレクトリとして依然として使用していることです、例えば Minix がそうです。

- /usr/bin: すべての実行可能ファイル（/sbin および/bin に含まれないもの）があります。例えば、gcc、firefox など；
- /usr/include: 様々なヘッダーファイルがあり、コンパイル時に使用されます；
- /usr/include/'package-name': プログラム特有のヘッダーファイル；

株式会社日昇テクノロジー

- /usr/lib: 実行可能ファイルが必要とするすべてのライブラリファイル；

- /usr/src: ソースコードがあり、Linux カーネルのソースコードはこのディレクトリにあります。

第 10 章 テキストエディタ

10.1 Vi/Vim エディタ

ほとんどの Linux システムには、ターミナル上でファイルを編集するための Vi エディタが標準装備されています。Vim は Vi のアップグレード版で、Vi の基本機能に加えて多くの改良と機能追加が行われており、より強力でカスタマイズ可能なテキストエディタです。例えば、コードの折り畳み、プラグイン、多言語サポート、垂直分割ウィンドウ、スペルチェック、コンテキストに応じた補完、タブでの編集などがサポートされています。Vim を愛用するユーザーは、それを IDE のように使いこなすこともできます。

Linux サーバーの運用管理や組み込み開発ボードの制御など、ほとんどの場合ターミナルのみを使用するため、Vi/Vim エディタの使用はほぼ最良の選択です。そのため、初心者でも Vi/Vim エディタを使ってファイルの基本的な読み書き変更ができるようになる必要があります。

10.1.1 Vim の使用デモ

10.1.1.1 Vim のインストール

Ubuntu では、apt を使用して Vim エディタをインストールできます。

```
1. sudo apt install vim
```

10.1.1.2 Vi/Vim を開く

vi または vim コマンドで開けます。どちらも基本的には同じ使い方です。

```
1. vi # システムに vim がインストールされていれば、このコマンドで vim が開きます。
2. vim # vim ソフトウェアを開きます。
```

また、次のコマンドでファイルを作成または開くことができます：

株式会社日昇テクノロジー

1. vi ファイル名 # ファイルが存在すれば開き、存在しなければ作成します。

2. vim ファイル名

10.1.1.3 Vim を終了する

Vim はターミナルで使用するため、グラフィカルインターフェイスがなく、すべての操作をコマンドで行います。そのため、Vim を終了する方法もコマンドで実現します。初心者にとっては、Vim からの退出方法が一つの問題です。

Vim を終了する最も単純な方法は、「Ctrl」+「z」を使用して強制終了することです。

ここでは、Vim の「ノーマルモード」と「コマンドラインモード」を使って Vim を終了する手順を紹介します。

Vim を開いた後、以下の手順で操作します。入力法が有効になっている場合は、英語に設定してください：

1.「Esc」キーを押して「ノーマルモード」に入ります。

2. 英語のコロン「:」を入力して「コマンドラインモード」に入ります。

3. 強制終了コマンド「q!」を入力します（「q」と英語の感嘆符「!」）。

4. Enter キーを押してコマンドを実行し、Vim を終了してターミナルに戻ります。

実行時に端末が表示する内容を下図に示します。

1. 終了キー「Esc」を押す
2. コロン「:」を入力する
3. 強制終了コマンド「q!」、つまり文字「q」と英語の感嘆符「!」を入力する
4. Enter キーを押してコマンドを実行する

:q!

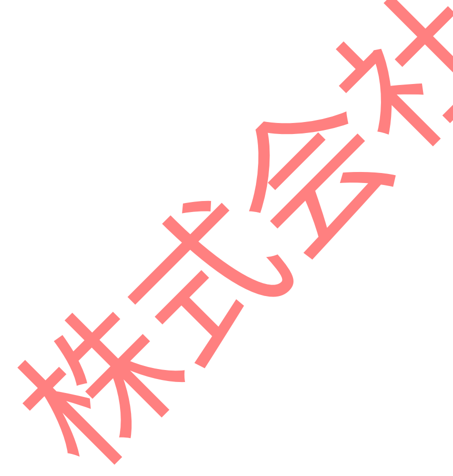
株式会社日昇テクノロジー

10.1.1.4 内容を入力する

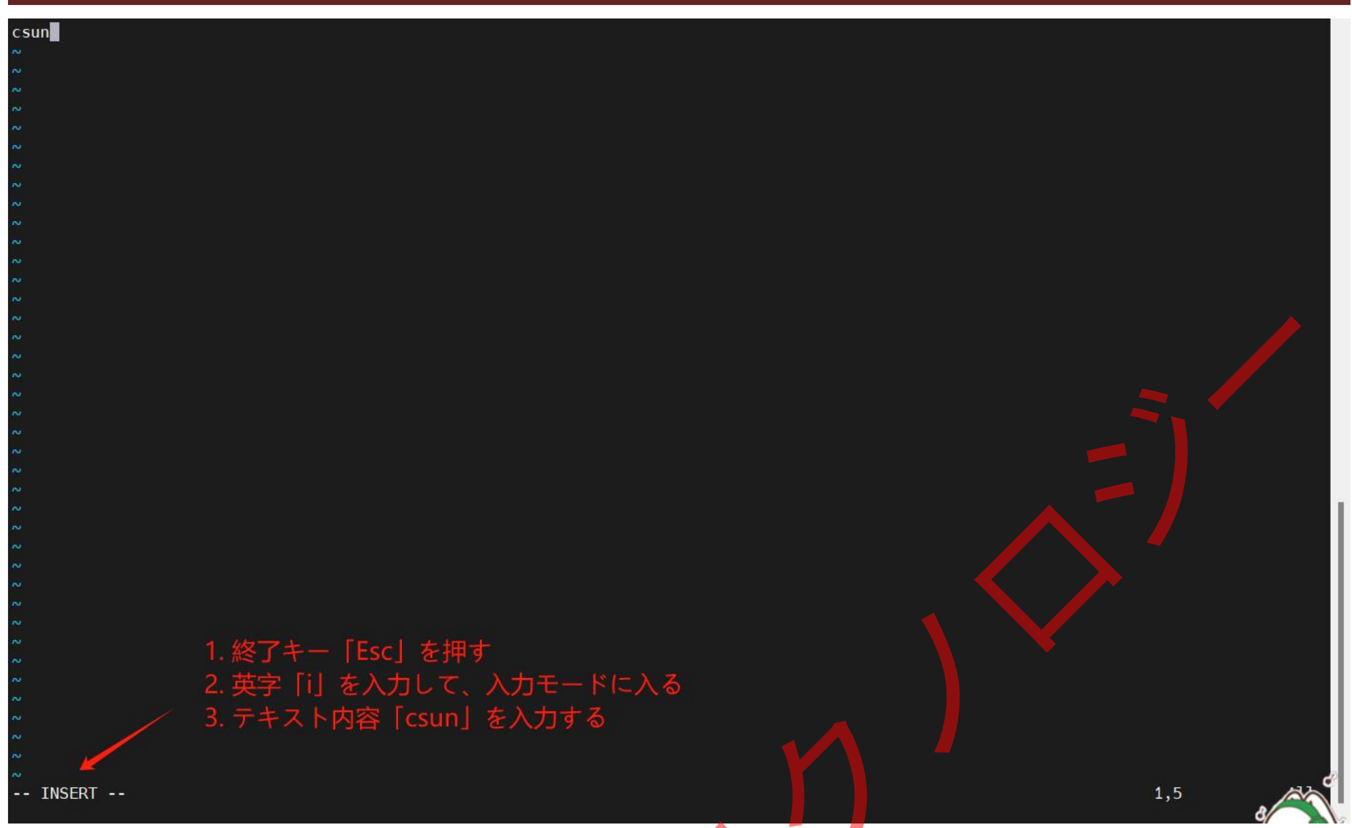
Vim を初めて使用する際、エディタに内容を入力することも簡単ではありません。Vim は開いた直後は「ノーマルモード」であり、キーボード入力はすべて通常のコマンドとして扱われ、テキストの内容としては認識されません。たまたま「インサートモード」に入るショートカットキーを押すと、その後の入力がテキストとして認識されます。

「インサートモード」に入って内容を入力し、ファイルを保存する手順は以下の通りです：

1. 「Esc」キーを押して「ノーマルモード」に入ります。
2. 「i」というノーマルコマンドを入力し、「インサートモード」に入ります。
3. 任意の内容を入力します。
4. 再び「Esc」キーを押して「ノーマルモード」に戻ります。
5. 英語のコロン「:」を入力して「コマンドラインモード」に入ります。
6. 保存して終了するコマンド「wq」を入力します。
7. Enter キーを押してコマンドを実行し、Vim を終了してターミナルに戻ります。



株式会社日昇テクノロジー



10.1.2 Vim3 つの作業モード

以下の 3 つの作業モードがあります：

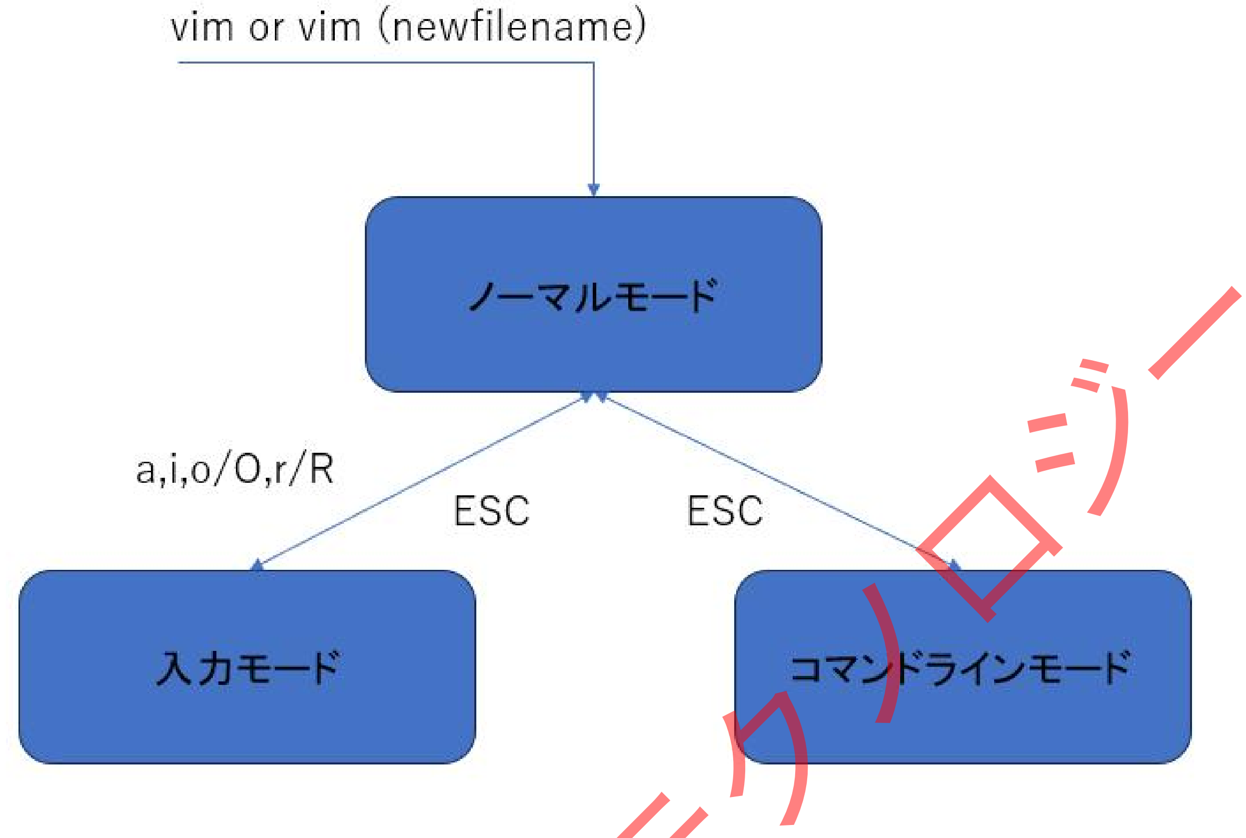
- ノーマルモード（normal mode）：テキストの閲覧、検索ができますが、編集はできません。このモードでのキーボード入力はショートカットキーとして扱われ、例えばコピー＆ペーストなどが行えます。Vim を開いた時、デフォルトでこのモードになります。

- 入力モード（insert mode）：通常のエディターのように機能し、このモードでのキーボード入力はテキストとして扱われます。

- コマンドラインモード（command-line mode）：保存、終了、置換などのコマンド、および Vim の高度な機能をサポートします。

Vim を使用する際、これら 3 つのモード間を頻繁に切り替えることになります。切り替え方法は以下のとおりです：

株式会社日昇テクノロジー



- 任意のモードから「Esc」キーを押すことでノーマルモードに入ります。

- ノーマルモードから「a」「i」「o」「O」「r」「R」などのキーを押すと挿入モードに入ります。

- ノーマルモードから「:」キーを押すとコマンドラインモードに入ります。

10.1.3 挿入モード

Vim は、ノーマルモードから入力モードへの複数のショートカットキーを提供しています。入力モードに入ると、通常のテキスト編集が可能になります。カーソルを動かすには矢印キーを使用し、エンターキーで改行します。操作は Windows のメモ帳と大差ありません。

入力モードに入る/退出するショートカットキー

ショートカットキー	機能の説明
i	現在のカーソル位置にテキストを挿入します。
a	現在のカーソル位置の次の文字にテキストを挿入します。
o	カーソルの現在位置の後に新しい行を挿入します。
r	現在のカーソル位置の文字を置換します。
R	現在のカーソル位置以降の文字を置換できます。「Esc」を押すと退出しま

株式会社日昇テクノロジー

	す。
ESC	挿入モードを退出します。

10.1.4 ノーマルモード

任意のモードから「Esc」キーを押すとノーマルモードに入ります。以下はノーマルモードでよく使用されるショートカットキーの一覧です。ノーマルモードでは、コピー、ペースト、削除、キーワードの検索置換などが行えます。

	ショートカットキー	機能説明
カーソル移動	k / ↑	カーソルを上に移動
	j / ↓	カーソルを下に移動
	h / ←	カーソルを左に移動
	l / →	カーソルを右に移動
	PageUp	上にページをめくる
	PageDown	下にページをめくる
	nG	第 n 行にジャンプ
文字検索および置換	/word	ファイル内でキーワード word を検索
	n	次のキーワードを検索
	N	前のキーワードを検索
	1,$s/word1/word2/gc	ファイル内のすべてのキーワード word1 を word2 に置換、ユーザーの確認が必要。（:1,$s/word1/word2/g を使用すると直接全てを置換）。これはコマンドモードで実行されます。
	u	直前の操作を元に戻す、Windows の Ctrl+Z に相当
	Ctrl+r	直前の操作をやり直す。
削除、カット、コピー、ペースト	d	カーソルで選択した内容を削除
	dd	現在の行を削除
	ndd	カーソルの後ろの n 行を削除
	x	カーソルで選択した文字をカット
	y	カーソルで選択した内容をコピー
	yy	現在の行をコピー
	nyy	現在の行から後ろの n 行をコピー
	p	コピーしたデータを現在の行の下にペ

株式会社日昇テクノロジー

		ースト
	P	コピーしたデータを現在の行の上にペースト
ブロック操作	v	複数の文字を選択
	V	複数行を選択
	ctrl+v	複数列を選択

10.1.5 コマンドラインモード

ノーマルモードでコロン「:」キーを押すとコマンドラインモードに入り、続けて実行したいコマンドを入力してエンターキーを押すと実行されます。

コマンドラインモードのショートカットキー

ショートカットキー	機能説明
w	ドキュメントを保存します。
w <filename>	<filename>というファイル名でドキュメントを別名保存します。
r <filename>	filename というファイル名のドキュメントを読み込みます。
q	ドキュメントが何も変更されていない場合、ソフトウェアを直接終了します。
q!	変更を保存せずにソフトウェアを直接終了します。
wq	ドキュメントを保存し、ソフトウェアを終了します。
set nu	行の先頭に行番号を追加します。
set nonu	行番号の表示を無効にします。
set hlsearch	検索結果をハイライト表示します。
! command	ターミナルウィンドウに戻り、command コマンドを実行します。Enter キーを押すと vim に戻ります。

10.1.6 Vi/Vim で簡単な sh スクリプトを作成

これまでの Vim の基本操作を紹介しましたが、最後に Vim を使って「hello world」を出力するスクリプトを作成してみましょう。

ターミナルで以下のコマンドを実行します：

```
vim hello_world.sh
```

株式会社日昇テクノロジー

次に、挿入モードに入り（ノーマルモードで「i」キーを押す）、以下のコードを入力した後、ファイルを保存して終了します（挿入モードを終了し、コマンドラインモードに入ってから「wq」を入力し、エンターキーを押します）。以下の内容は全て英数字で入力してください。

```
echo "helloworld, this is a script test!"
```

このスクリプトファイルの中の echo コマンドは、その後のテキスト内容を直接ターミナルに出力するために使用されます。

```
echo "helloworld,this is a script test!"
~
~
~
:wq
```

Vim を終了してターミナルに戻った後、`source`コマンドを使って、作成したスクリプトファイルを実行します：

```
# 実行権限の付与

sudo chmod u+x hello_world.sh
```

ターミナルに「helloworld,this is a script test!」と表示されます。

```
cat@lubancat:~$ vim hello_world.sh
cat@lubancat:~$ chmod u+x hello_world.sh
cat@lubancat:~$ ./hello_world.sh
helloworld,this is a script test!
cat@lubancat:~$
```

第 11 章 この章のコードと記事の取得

株式会社日昇テクノロジー

11.1 Git とは

Git は、様々なプロジェクトのバージョン管理を効率的に行うための非常に強力な分散型バージョン管理システムです。

Git の誕生は、Linux コミュニティの Andrew（Samba の父）が BitKeeper リポジトリに接続するプラグインを作成したことが起源です。これにより BitMover 社の怒りを買い、Linux コミュニティへの BitKeeper の無料ライセンスが取り消されました。その結果、Linux の父、Torvalds は、Linux コミュニティのために新しいバージョン管理ツールである Git を開発しました。

11.2 Git の特徴

他のバージョン管理ツールと比較して、Git には以下のような利点があります：

- スナップショット記録。Git と他のバージョン管理ツールの大きな違いは、データの保存方法です。他のツールは通常、原始データとファイルの変更記録として情報を保存しますが、Git は全ファイルのスナップショットを生成して記録します（スナップショットは単なるファイルコピーではなく、全ファイルへのインデックスです）。

- 分散型。プロジェクトに関わる各開発者のコンピュータには、プロジェクトの完全なバックアップがあります。これにより、データ損失のリスクを避けることができます。また、ローカルのプロジェクトバックアップにより、すべてのバージョン管理操作がミリ秒単位で完了します。これは、中央サーバーの応答を待つ必要がある SVN や CVS などの集中型バージョンツールと比べて、非常に優れたユーザー体験を提供します。

- オープンソース。Linux カーネルと同様に、オープンソースであることが Git の信頼性とセキュリティを保証し、Git の機能が強化されることにもつながります。

11.3 Git と GitHu

GitHub は、Git ベースのオンラインプロジェクトホスティングプラットフォームです。Web インターフェースを提供し、ユーザーはリモートリポジトリを作成して自分のプロジェクトを保存できます。すべ

株式会社日昇テクノロジー

ての開発者がこのリモートリポジトリを基に協力してプロジェクトを維持し、改善することができます。

11.4 Git を使用してプロジェクト資料をダウンロードする

11.4.1 Linux システム

11.4.1.1 Git のインストール

異なるディストリビューションに応じて、関連するパッケージ管理ツールを使用してインストールできます。

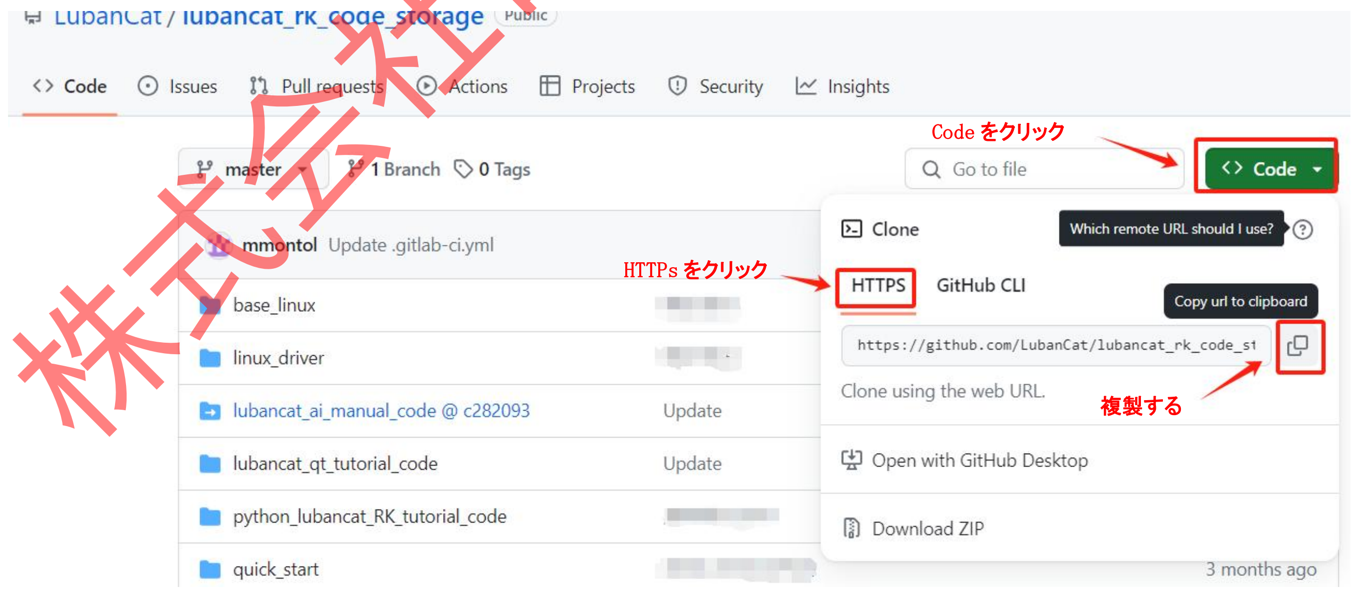
例えば Ubuntu と Debian では：

```
# ターミナルで実行

sudo apt-get install git -y
```

11.4.1.2 プロジェクトリポジトリアドレスの取得

GitHub または Gitee のウェブサイトで、ダウンロードしたいプロジェクトを見つけ、そのプロジェクトのクローンリンクをコピーします。コードを例にとると以下のようになります：

- GitHub リポジトリアドレス



株式会社日昇テクノロジー

11.4.1.3 プロジェクトのダウンロード

Ubuntu でプロジェクトをダウンロードするには、リポジトリを保存したいフォルダに移動し、次に「git clone + リポジトリアドレス」コマンドを実行してプロジェクトをダウンロードします。

```
git clone + リポジトリアドレス
```

11.4.2 Windows システム

11.4.2.1 Git のインストール

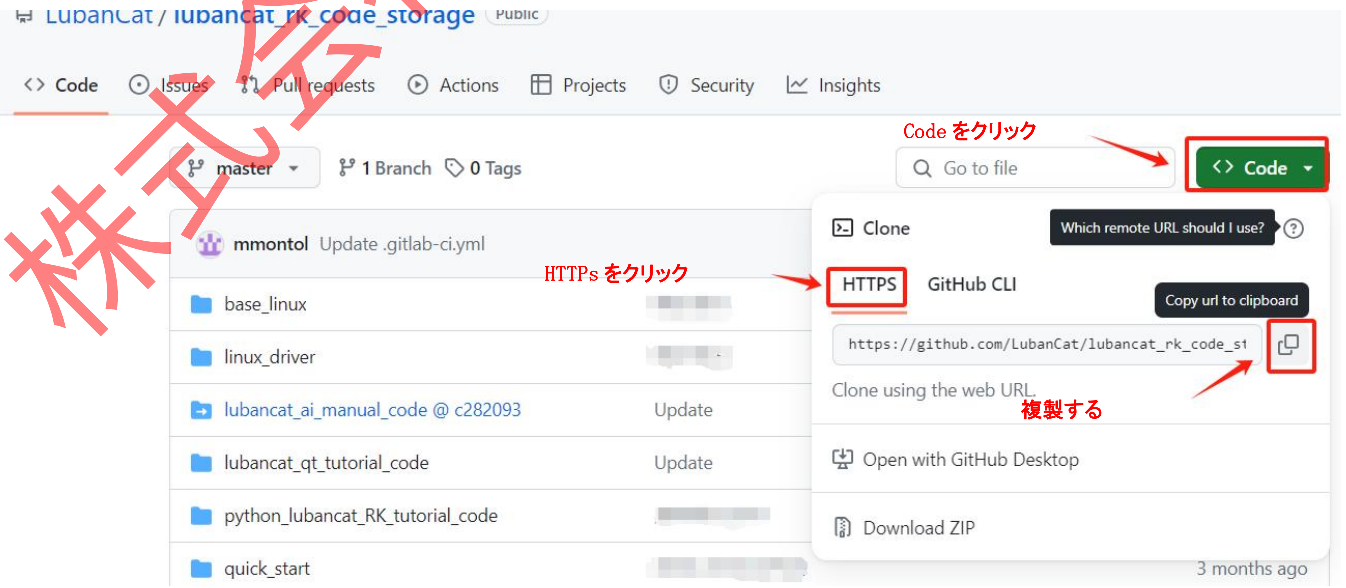
Git の公式ウェブサイトからインストールパッケージをダウンロードし、直接インストールします。

https://gitforwindows.org

11.4.2.2 プロジェクトリポジトリアドレスの取得

GitHub や Gitee のウェブサイトで、ダウンロードしたいプロジェクトを探し、そのプロジェクトのクローンリンクをコピーします。以下は参考のための画像です：

- GitHub リポジトリアドレス

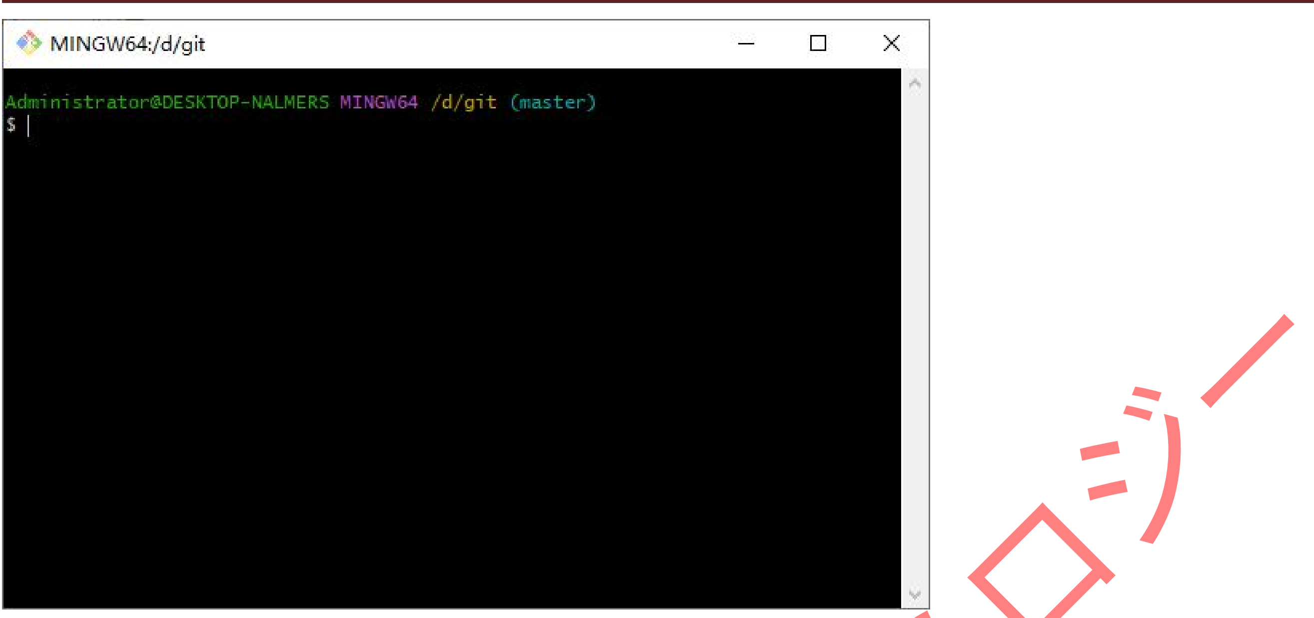


株式会社日昇テクノロジー

11.4.2.3 プロジェクトのダウンロード

Windows に Git をインストールした後、プロジェクトをダウンロードしたいディレクトリで右クリックして「Git Bash Here」を選択します。以下は参考のための画像です：

新しいウィンドウが開きます。



Git Bash コマンドツールで「git clone + "リポジトリアドレス"」コマンドを実行して、プロジェクトをダウンロードします。したがって、実行するコマンドは次のとおりです：

```
git clone + リポジトリアドレス
```

この時点でリポジトリのダウンロードが完了します。

第 12 章 プログラムのコンパイル

第 12 章では、Linux で GCC を使用して Hello World プログラムをコンパイルする過程と概念について説明します。これは、コンピュータ上の Linux システムや LubanCat-RK ボードで C 言語プログラムをコンパイルすることができます。

Windows で C プログラムを開発する際には Visual Studio を使用し、MCU プログラムを開発する際には Keil や IAR などの IDE を使用します。Linux では、eclipse や Clion などの類似の IDE があります。これらの環境では、通常、プロジェクトテンプレートを作成し、具体的なコードを記述した後、対応するコンパイル・実行ボタンをクリックするだけで操作が完了します。

大規模なアプリケーションの開発やデバッグ時には、IDE を使用することが非常に有効です。IDE の特徴は、さまざまな一般的な操作をグラフィカルインターフェースに包括してユーザーが使用できるようにすることですが、シェルコマンドラインの学習と同じように、グラフィカルインターフェースの下にはさ

株式会社日昇テクノロジー

らに多くの機能が隠されています。Linux の日常開発では、コマンドラインを直接使用することが多く、他のコマンドラインツールと組み合わせると、コンパイルプロセスが簡単で迅速になり、コンパイルの原理を理解するのに役立ちます。

本章では、hello world プログラムの Linux でのコンパイルと実行プロセスを分解し、GCC、readelf、ldd ツールの基本的な使用法をマスターすることで、開発プロセスを理解し、将来的にコンパイルツールチェーンを構築する際に何をすべきかを理解することが目的です。各コンパイルステップとその生成ファイルを理解することは、Makefile の作成や他のツールの使用に非常に役立ちます。リンクプロセスを理解することで、なぜあるプログラムが特定のファイルに依存しているのかを理解し、Linux ファイルシステムを特別にカスタマイズするのに役立ちます。

本章のサンプルコードディレクトリは：base_linux/hello

12.1 GCC コンパイルツールチェーン

GCC コンパイルツールチェーン（toolchain）は、ソースコードを実行可能なアプリケーションに変換するために、GCC コンパイラを中心とした一連のツールを指します。主に以下の三つの部分から構成されます：

- gcc-core：GCC コンパイラ自体で、プリプロセスとコンパイルプロセスを完了し、たとえば C コードをアセンブリコードに変換します。
- Binutils：GCC コンパイラ以外の一連の小ツールには、リンカー ld、アセンブラ as、オブジェクトファイルフォーマットビューア readelf などが含まれます。
- glibc：C 言語の主要な標準関数ライブラリで、C 言語で頻繁に使用される printf 関数や malloc 関数が含まれます。

多くの場合、GCC コンパイラを GCC コンパイルツールチェーン全体を指すために使用します。

株式会社日昇テクノロジー

12.1.1 GCC コンパイラ

GCC（GNU Compiler Collection）は、GNU が開発したプログラミング言語のコンパイラです。GCC は元々「GNU C Compiler」の略で、C 言語のみをサポートしていました。後に、C++、Fortran、Java など他の多くのプログラミング言語をサポートするように拡張されました。そのため、「GNU Compiler Collection」の略として再定義され、歴史上最も優れたコンパイラの一つになりました。その実行効率は、一般的なコンパイラと比較して平均で 20%～30%高いです。

LubanCat4 ボードには gcc コンパイラがプリインストールされています。システムに gcc がない場合は、以下の方法でインターネット経由でインストールできます：

```
sudo apt install gcc
```

gcc のバージョンとインストール場所を確認するには：

ボード上で以下のコマンドを実行します

```
gcc -v # gcc コンパイラのバージョンを確認

which gcc # gcc のインストールパスを確認
```

```
cat@lubancat:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/aarch64-linux-gnu/9/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.4.0-1ubuntu1~20
.04.1' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-languages=
c,ada,c++,go,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-major-version-o
nly --program-suffix=-9 --program-prefix=aarch64-linux-gnu- --enable-shared --en
able-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-t
hreads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu --enable-libstd
cxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-
gnu-unique-object --disable-libquadmath --disable-libquadmath-support --enable-p
lugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --e
nable-objc-gc=auto --enable-multiarch --enable-fix-cortex-a53-843419 --disable-w
error --enable-checking=release --build=aarch64-linux-gnu --host=aarch64-linux-g
nu --target=aarch64-linux-gnu
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)
cat@lubancat:~$ which gcc
/usr/bin/gcc
cat@lubancat:~$
```

画像内の 2 つの情報は以下の通りです：

- 「Target: aarch64-linux-gnu」は、その GCC のターゲットプラットフォームが ARM64 ビットアーキテクチャであることを示しており、それによってコンパイルされたアプリケーションは ARM ボー

ドプラットフォームにのみ適用され、x86 アーキテクチャには適用されません。

- 「gcc version 9.4.0」は、その GCC のバージョンが 9.4.0 であることを示しています。一部のプログラムはコンパイラのバージョンに要求があるかもしれませんが、デモで使用するアプリケーションは比較的シンプルで互換性が高いため、初めはこれについて心配する必要はありません。ただし、特定のバージョンの uboot や Linux カーネルをコンパイルする際には、GCC のバージョンに要求があるかもしれません。

12.1.2 Binutils ツールセット

Binutils（binary utilities の略）は GNU バイナリツールセットで、通常 GCC コンパイラと一緒にシステムにインストールされます。公式ウェブサイトは https://www.gnu.org/software/binutils/ です。

プログラム開発時にこれらのツールを直接呼び出すことはほとんどありませんが、GCC コンパイルコマンドを使用する際に GCC コンパイラによって間接的に呼び出されます。以下は、よく使用されるいくつかのツールです：

- as：アセンブラ。アセンブリ言語コードを機械語コード（オブジェクトファイル）に変換します。
- ld：リンカ。複数のオブジェクトファイルを最終的な実行可能プログラムファイルに組み立てます。
- readelf：オブジェクトファイルや実行可能プログラムファイルの情報を表示するために使用されます。
- nm：オブジェクトファイル内のシンボルを表示します。
- objcopy：オブジェクトファイルの形式変換に使用されます（例：.bin を.elf に変換、.elf を.bin に変換など）。
- objdump：オブジェクトファイルの情報を表示し、主に逆アセンブルに使用されます。
- size：オブジェクトファイルの異なるセクションのサイズと総サイズを表示します（例：コードセクションのサイズ、データセクションのサイズ、静的メモリの使用量、総サイズなど）。

システムのデフォルトの Binutils ツールセットは/usr/bin ディレクトリ下にあり、以下のコマンドを使

株式会社日昇テクノロジー

用してシステム内の存在する Binutils ツールセットを確認できます：

```
1 # Ubuntu 上で以下のコマンドを実行

2 ls /usr/bin/ | grep linux-gnu-
```

```
cat@lubancat:~$ ls /usr/bin/ | grep linux-gnu-
aarch64-linux-gnu-addr2line
aarch64-linux-gnu-ar
aarch64-linux-gnu-as
aarch64-linux-gnu-c++filt
aarch64-linux-gnu-cpp
aarch64-linux-gnu-cpp-9
aarch64-linux-gnu-dwp
aarch64-linux-gnu-elfedit
aarch64-linux-gnu-gcc
aarch64-linux-gnu-gcc-9
aarch64-linux-gnu-gcc-ar
aarch64-linux-gnu-gcc-ar-9
aarch64-linux-gnu-gcc-nm
aarch64-linux-gnu-gcc-nm-9
aarch64-linux-gnu-gcc-ranlib
aarch64-linux-gnu-gcc-ranlib-9
aarch64-linux-gnu-gcov
aarch64-linux-gnu-gcov-9
aarch64-linux-gnu-gcov-dump
aarch64-linux-gnu-gcov-dump-9
aarch64-linux-gnu-gcov-tool
aarch64-linux-gnu-gcov-tool-9
aarch64-linux-gnu-gold
aarch64-linux-gnu-gprof
aarch64-linux-gnu-ld
aarch64-linux-gnu-ld.bfd
aarch64-linux-gnu-ld.gold
aarch64-linux-gnu-nm
aarch64-linux-gnu-objcopy
aarch64-linux-gnu-objdump
aarch64-linux-gnu-pkg-config
aarch64-linux-gnu-ranlib
aarch64-linux-gnu-readelf
aarch64-linux-gnu-run
aarch64-linux-gnu-size
aarch64-linux-gnu-strings
aarch64-linux-gnu-strip
aarch64-unknown-linux-gnu-pkg-config
```

画像には Binutils ツールの完全な名前がリストされており、ターミナルで使用する際は通常、それらのエイリアスを直接使用します。後ほど、readelf ツールの使用方法について説明します。

12.1.3 glibc ライブラリ

glibc ライブラリは、GNU システムおよび Linux システムのために GNU が作成した C 言語の標準ライブラリです。ほとんどの C プログラムがこの関数ライブラリに依存しており、read、write、open など

株式会社日昇テクノロジー

の一般的なファイル操作関数、printf、malloc などの関数が含まれています。

Ubuntu システムでは、libc.so.6 が glibc のライブラリファイルで、このライブラリファイルを実行することでバージョンを確認できます。コマンドは以下のとおりです：

```
# Ubuntu 上で以下のコマンドを実行
# 以下は Ubuntu 64 ビットマシンの glibc ライブラリファイルのパスで、直接実行可能です
/lib/aarch64-linux-gnu/libc.so.6
```

```
cat@lubancat:~$
cat@lubancat:~$ /lib/aarch64-linux-gnu/libc.so.6
GNU C Library (Ubuntu GLIBC 2.31-0ubuntu9.9) stable release version 2.31.
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 9.4.0.
libc ABIs: UNIQUE ABSOLUTE
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
cat@lubancat:~$
```

このコマンドは、システムが使用している glibc がバージョン 2.31 であり、GCC 9.4.0 バージョンのコンパイラでコンパイルされたことを示しています。

C 言語を学ぶ際に、printf や malloc といった関数がどのように実装されているかに特に興味を持つ学生もいるかもしれませんが、Windows 下の C ライブラリはオープンソースではなく、確認することはできません。しかし、Linux では、glibc のソースコードを直接研究したり、開発コミュニティに参加して自分のコードを貢献することが可能です。glibc の公式ウェブサイトは https://www.gnu.org/software/libc/ で、このウェブサイトからソースコードをダウンロードして学習することができます。

GCC コンパイルツールをより直感的に体験するために、以下のステップに従って新しい世界の扉を開いてみましょう。

12.2 HelloWorld

12.2.1 作業ディレクトリの作成

現在のユーザーの下に、この章で使用する作業ディレクトリ test を作成します。

株式会社日昇テクノロジー

```
# Debian 上で以下のコマンドを実行

mkdir test # test ディレクトリを作成

cd test # ディレクトリを切り替え
```

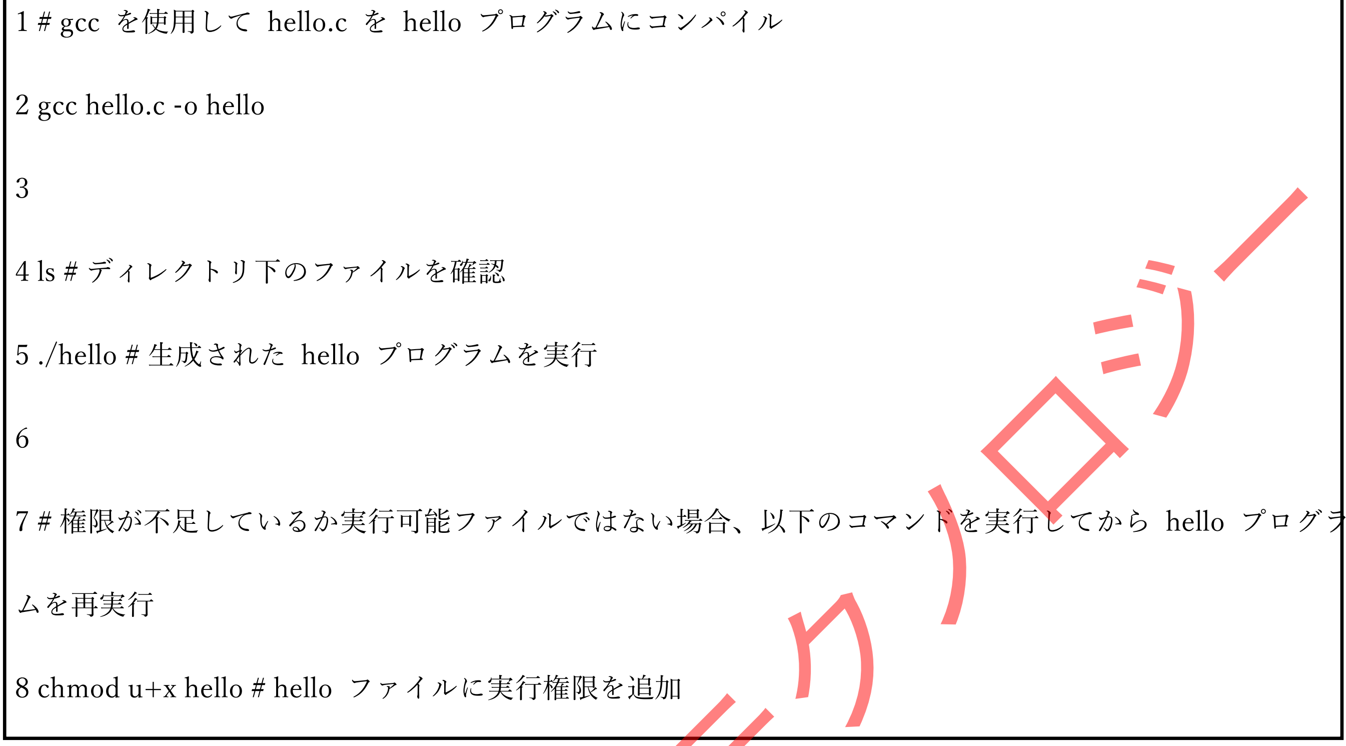
12.2.2 コードファイルの作成

エディターを使用して hello.c という名前のファイルを新規作成し、以下のサンプルコードを入力して hello_c ディレクトリに保存します。

```
1 #include <stdio.h>

2

3 int main()

4 {

5 int i;

6 printf("hello, world! This is a C program.¥n");

7 for(int i=0;i<10;i++ ){

8 printf("output i=%d¥n",i);

9 }

10

11 return 0;

12 }
```

株式会社日昇テクノロジー

12.2.3 コンパイルと実行



1 # gcc を使用して hello.c を hello プログラムにコンパイル

2 gcc hello.c -o hello

3

4 ls # ディレクトリ下のファイルを確認

5 ./hello # 生成された hello プログラムを実行

6

7 # 権限が不足しているか実行可能ファイルではない場合、以下のコマンドを実行してから hello プログラムを再実行

8 chmod u+x hello # hello ファイルに実行権限を追加

以下のように:

```
cat@lubancat:~/test$
cat@lubancat:~/test$ ./hello
hello, world! This is a C program.
output i=0
output i=1
output i=2
output i=3
output i=4
output i=5
output i=6
output i=7
output i=8
output i=9
cat@lubancat:~/test$
```

これは、Linux で GCC を使用して簡単な C アプリケーションを開発して実行する基本的なプロセスです。次に、GCC のコンパイルプロセスについて説明します。

株式会社日昇テクノロジー

12.3 GCC コンパイルプロセス

12.3.1 基本構文

GCC で使用されるコマンド構文は以下の通りです：

gcc [オプション] 入力ファイル名

よく使われるオプション：

-o：出力される実行ファイルの名前を指定します。指定しない場合、生成される実行ファイル名は a.out になります。

-E：プリプロセスのみを行い、コンパイルやアセンブルは行いません。

-S：コンパイルのみを行い、アセンブルは行いません。

-c：コンパイルとアセンブルを行いますが、リンクは行いません。

-g：デバッグ情報を含む実行ファイルを生成し、gdb でのデバッグを容易にします。

-Ox：プログラムの最適化レベルを設定します。例えば、-O0、-O1、-O2、-O3 で、数字が大きいほどコードの最適化レベルが高くなりますが、プログラムが正常に動作しない可能性があります。

12.3.1.1 コンパイルプロセス

プログラムのコンパイルプロセスを理解していないと、GCC のコンパイルオプションが理解しにくいかもしれません。ここでは、X86_64 プラットフォームの Ubuntu でのコンパイルプロセスを例にして説明します。ARM プラットフォームの Debian でのコンパイルプロセスも同様です。

GCC のコンパイルオプションは、-g と-Ox を除いて、実際にはコンパイルの各ステップを指すものです。

以下のコマンドは、直接実行可能なファイルをコンパイルする方法です：

株式会社日昇テクノロジー

```
1 # 直接コンパイルして実行ファイルを生成
2 gcc hello.c -o hello
3
4 # 上記のコマンドは以下の全操作に相当します
5 # プリプロセス、ヘッダーファイルのコードを C コードに集約し、*.c から*.i ファイルに変換することを意味します
6 gcc -E hello.c -o hello.i
7
8 # コンパイル、C コードをアセンブリコードに変換し、*.i から*.s ファイルに変換することを意味します
9 gcc -S hello.i -o hello.s
10
11 # アセンブル、アセンブリコードを機械コードに変換し、*.s から*.o、つまりオブジェクトファイルに変換することを意味します
12 gcc -c hello.s -o hello.o
13
14 # リンク、異なるファイル間の呼び出し関係をリンクし、一つまたは複数の*.o を最終的な実行ファイルに変換します
15 gcc hello.o -o hello
```

MCU 開発経験がある読者には、野火の「STM32 ライブラリ開発実戦ガイド」や「i.MX RT ライブラリ開発実戦ガイド」内の「MDK のコンパイルプロセス及びファイル詳細」章を学習することをお勧めします。これは MCU の観点から上記のコンパイルプロセスを非常に詳細に説明しています。GCC のコンパイルプロセスも同様で、Linux プラットフォームでこのプロセスを分解することはより直感的ですが、本

株式会社日昇テクノロジー

章は入門章として、コンパイル原理の概要をリスト面的に構築するに留め、詳細な紹介は行いません。

GCC コンパイルツールチェーンが C ソースファイルをコンパイルする際には、以下の 4 ステップが必要です：

1. プリプロセス、ソースコードファイルのファイルインクルード、プリコンパイルステートメント（マクロ定義など）を展開し、.i ファイルを生成します。これは、ヘッダーファイルのコードやマクロなどの内容をより純粋な C コードに変換することを意味し、生成されたファイルは.i という拡張子を持ちます。

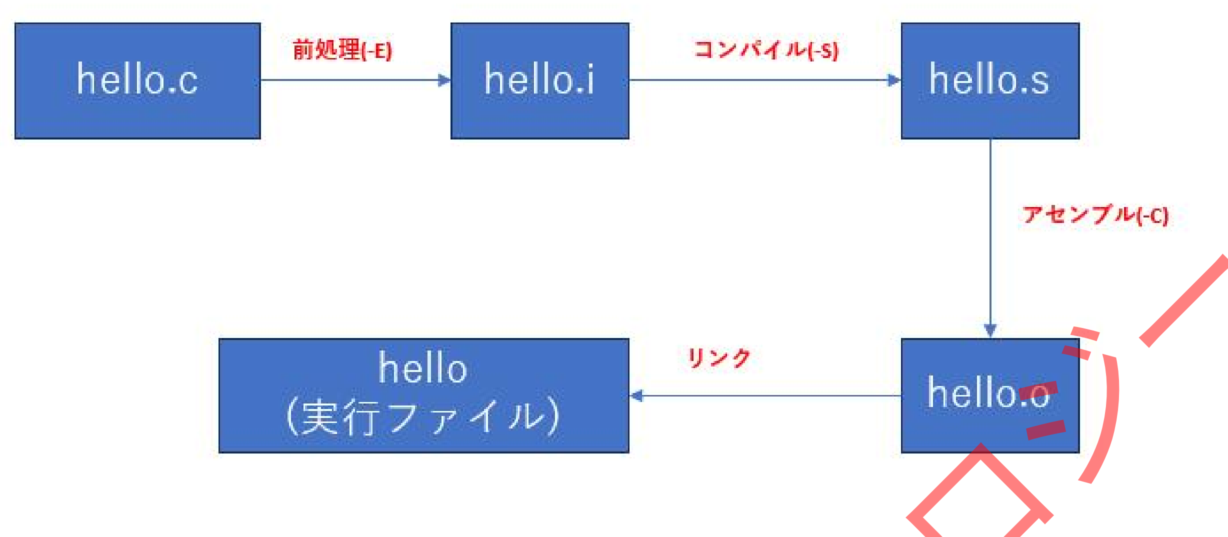
2. コンパイル、プリプロセス後の.i ファイルをアセンブリ言語にコンパイルし、.s ファイルを生成します。これは、コードを C 言語からアセンブリ言語に変換する作業で、GCC コンパイラが行います。

3. アセンブル、アセンブリ言語ファイルをアセンブルしてオブジェクトファイル.o を生成します。各ソースファイルには対応するオブジェクトファイルがあります。これは、アセンブリ言語のコードを機械コードに変換する作業で、as アセンブラが行います。

4. リンク、最後に各ソースファイルに対応する.o ファイルをリンクして実行可能なプログラムファイルを生成します。これはリンカ ld が行う作業です。

上記の hello.c を例に、括弧内が gcc のパラメータを示しており、ステップバイステップのコンパイルプロセスが以下の図に示されます。

株式会社日昇テクノロジー



コンパイル原理については、専門の書籍を読んでさらに理解を深めることができます。これはプログラム開発に大きな利益をもたらします。以下では、各段階で生成されるファイルについて概観します。

12.3.2 プリプロセス段階

GCC の-E オプションを使用すると、コンパイラは.i ファイルを生成します。-o オプションを使用して出力ファイル名を指定できます。実行コマンドは以下の通りです：

```
gcc -E hello.c -o hello.i
```

生成された`hello.i`をエディタで開くと、以下のような内容が表示されます。

リスト 2: hello.i ファイル

```
1 # 1 "hello.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
```

株式会社日昇テクノロジー

7 # 1 "hello.c"

8 # 1 "/usr/include/stdio.h" 1 3 4

9 # 27 "/usr/include/stdio.h" 3 4

10 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4

11 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4

12 # 1 "/usr/include/features.h" 1 3 4

13 # 424 "/usr/include/features.h" 3 4

14 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4

15 # 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4

16 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4

17 # 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4

18 # 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4

19 # 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4

20 # 425 "/usr/include/features.h" 2 3 4

21 # 448 "/usr/include/features.h" 3 4

22 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4

23 # 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4

24 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4

25 # 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4

26 # 449 "/usr/include/features.h" 2 3 4

27 # 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4

28 # 28 "/usr/include/stdio.h" 2 3 4

29 # 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4

株式会社日昇テクノロジー

30 # 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

31 # 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

32 typedef long unsigned int size_t;

33 # 34 "/usr/include/stdio.h" 2 3 4

34 # 1 "/usr/include/x86_64-linux-gnu/bits/types.h" 1 3 4

35 # 27 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4

```
36 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4

37 # 28 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4

38 typedef unsigned char __u_char;

39 typedef unsigned short int __u_short;

40

41 ......

42

43 int main(void)

44 {

45 printf("hello, world! This is a C program.¥n");

46 for(int i = 0; i < 10; i++)

47 {

48 printf("output i = %d¥n", i );

49 }

50 return 0;

51 }
```

株式会社日昇テクノロジー

このプリプロセス段階では、ヘッダーファイルのコードやマクロなどが展開され、より純粋な C コードに変換されますが、生成されたファイルは`.i`という拡張子を持ちます。

12.3.3 コンパイル段階

GCC は`-S`オプションを使用して、ソースファイルをアセンブリ言語のコードファイル（`.s`拡張子）にコンパイルすることができます。このプロセスでは、GCC は各ソースファイルの文法をチェックしますが、定義されていない関数を呼び出してもエラーにはなりません。

```
# C コードをアセンブリコードにコンパイルし、*.i から*.s ファイルを生成
gcc -S hello.i -o hello.s
# C ファイルを直接入力としてコンパイルすることもでき、上記のコマンドと同等
gcc -S hello.c -o hello.s
```

コンパイルで生成された hello.s ファイルの内容は以下の通りです：

```
1 .arch armv8-a
2 .file "hello.c"
3 .text
4 .section .rodata
5 .align 3
6 .LC0:
7 .string "hello, world! This is a C program."
8 .align 3
9 .LC1:
10 .string "output i=%d¥n"
11 .text
```

株式会社日昇テクノロジー

```
12 .align 2
13 .global main
14 .type main, %function
15 main:
16 .LFB0:
17 .cfi_startproc
```

```
18 stp x29, x30, [sp, -32]!
19 .cfi_def_cfa_offset 32
20 .cfi_offset 29, -32
21 .cfi_offset 30, -24
22 mov x29, sp
23 adrp x0, .LC0
24 add x0, x0, :lo12:.LC0
25 bl puts
26 str wzr, [sp, 28]
27 b .L2
28 .L3:
29 ldr w1, [sp, 28]
30 adrp x0, .LC1
31 add x0, x0, :lo12:.LC1
32 bl printf
33 ldr w0, [sp, 28]
```

株式会社日昇テクノロジー

34 add w0, w0, 1

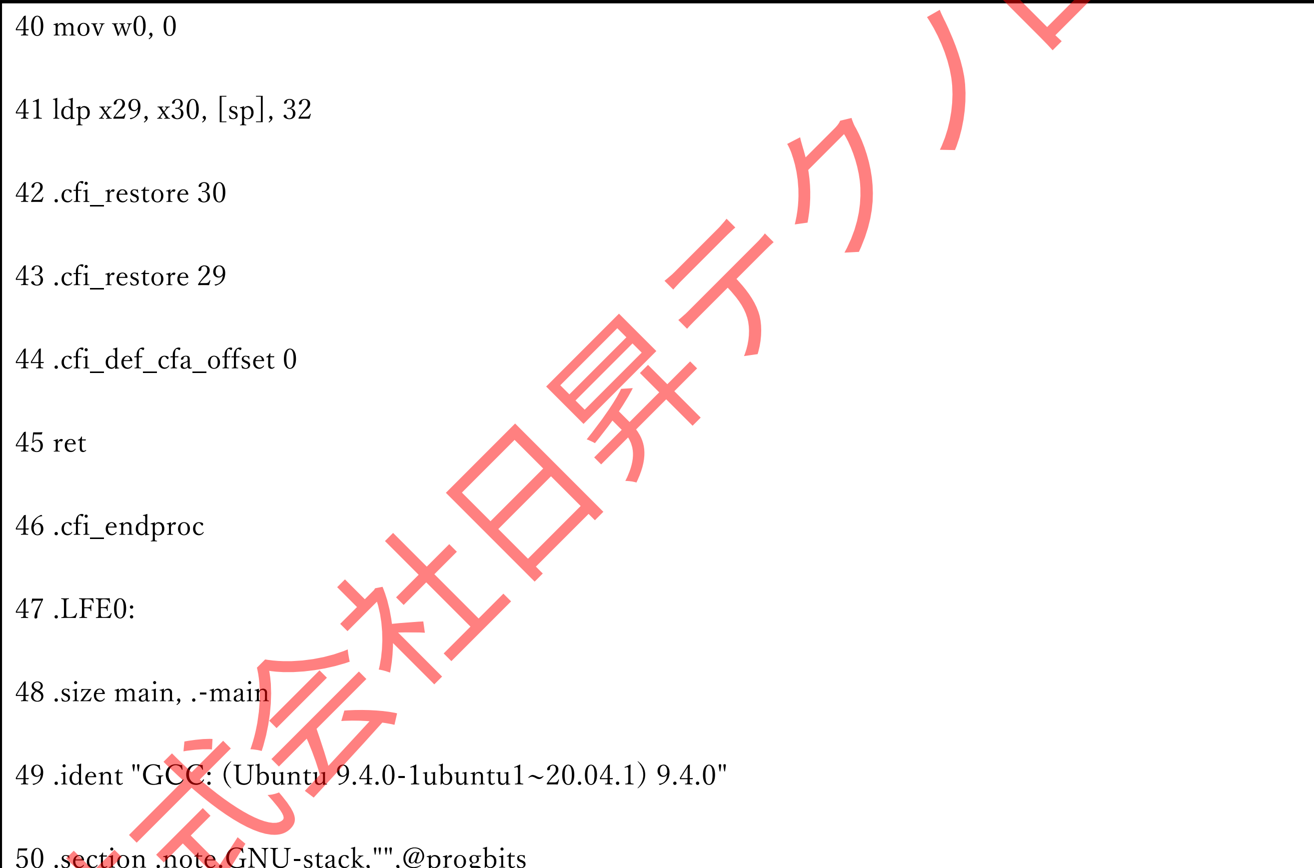
35 str w0, [sp, 28]

36 .L2:

37 ldr w0, [sp, 28]

38 cmp w0, 9

39 ble .L3



```
40 mov w0, 0
41 ldp x29, x30, [sp], 32
42 .cfi_restore 30
43 .cfi_restore 29
44 .cfi_def_cfa_offset 0
45 ret
46 .cfi_endproc
47 .LFE0:
48 .size main, .-main
49 .ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
50 .section .note.GNU-stack,"",@progbits
```

株式会社日昇テクノロジー

12.3.4 アセンブル段階

GCC の-c オプションは、ソースファイルをコンパイル（compile）のみ行い、リンクは行わずに、ソースプログラムをオブジェクトファイル（.o 拡張子）にします。コンピュータは 0 または 1 のみを理解し、C 言語やアセンブリ言語を理解しません。コンパイルとアセンブルの後、生成されたオブジェクトファイ

ルは機械コードを含み、これはコンピュータで直接実行できます。通常、-c オプションを使用して、上述の 2 つのプロセスをスキップできます。

```
# アセンブルして、アセンブリコードを機械コードに変換し、*.s を*.o に変換します
gcc -c hello.s -o hello.o
# C ファイルを直接入力としてアセンブルすることもでき、上記のコマンドと同等
gcc -c hello.c -o hello.o
```

.o ファイルはコンピュータが読むためのものであり、直接エディタで開くと乱数として表示されます。

hello.o ×
home > embedfire > workdir > example > hello_c > hello.o
このファイルはバイナリファイルであるか、サポートされていないテキストエンコーディングを使用しているため、エディタで表示できません。それでも開きますか?

Linux で生成される.o オブジェクトファイル、.so ダイナミックライブラリファイル、および次のセクションで生成される最終実行ファイルはすべて ELF 形式で、readelf ツールを使用して内容を表示できます。以下のコマンドを実際に実行してみてください：

```
# hello.o があるディレクトリで以下のコマンドを実行します
readelf -a hello.o
```

株式会社日昇テクノロジー

```
cat@lubancat:~/test$ readelf -a hello.o
  ELF Header:
    Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
    Class:                             ELF64
    Data:                              2's complement, little endian
    Version:                           1 (current)
    OS/ABI:                            UNIX - System V
    ABI Version:                       0
    Type:                              REL (Relocatable file)
    Machine:                           AArch64
    Version:                           0x1
    Entry point address:               0x0
    Start of program headers:          0 (bytes into file)
    Start of section headers:          968 (bytes into file)
    Flags:                             0x0
    Size of this header:               64 (bytes)
    Size of program headers:           0 (bytes)
    Number of program headers:         0
    Size of section headers:           64 (bytes)
    Number of section headers:         13
    Section header string table index: 12

  Section Headers:
    [Nr] Name              Type             Address           Offset
         Size              EntSize          Flags  Link  Info  Align
    [ 0]                   NULL             0000000000000000  00000000
         0000000000000000  0000000000000000           0     0     0
    [ 1] .text             PROGBITS         0000000000000000  00000040
         0000000000000050  0000000000000000  AX       0     0     4
    [ 2] .rela.text        RELA             0000000000000000  000002b8
         0000000000000090  0000000000000018   I      10     1     8
    [ 3] .data             PROGBITS         0000000000000000  00000090
         0000000000000000  0000000000000000  WA       0     0     1
    [ 4] .bss              NOBITS           0000000000000000  00000090
         0000000000000000  0000000000000000  WA       0     0     1
    [ 5] .rodata           PROGBITS         0000000000000000  00000090
         0000000000000035  0000000000000000   A       0     0     8
    [ 6] .comment          PROGBITS         0000000000000000  000000c5
         000000000000002c  0000000000000001  MS       0     0     1
    [ 7] .note.GNU-stack   PROGBITS         0000000000000000  000000f1
```

readelf ツールの出力情報から、ターゲットファイルが ELF ヘッダー、プログラムヘッダー、セクションなどを含んでいることがわかります。*.o ターゲットファイルや*.so ライブラリファイルについて、コンパイラはリンク段階でこれらの情報を利用して複数のファイルを組織します。実行可能ファイルについては、システムがこれらの情報に基づいてプログラムをロードして実行します。

12.3.5 リンク段階

リンクプロセスは、アセンブルプロセスで生成された全てのオブジェクトファイルをリンクし、実行可能ファイルを生成します。

例えば、プロジェクトに A と B の 2 つのコードファイルが含まれており、コンパイル後にそれぞれの A.o と B.o オブジェクトファイルが生成された場合、A のコードで B のある関数 fun を呼び出したとしても、A のコードに fun の関数宣言が含まれていれば、コンパイルは通過しますが、B で fun 関数が実際に定義されていない場合でもエラーは発生しません（もちろん、関数宣言がなければコンパイルエラーになります）。これは、A.o と B.o オブジェクトファイルがコンパイル段階で独立しているためで、リンク段階では、A と B の間の関数呼び出し関係を整理し、A が fun 関数をどこで呼び出せるかのマッピング関係を確立する必要があります。リンクプロセスで fun 関数の具体的な定義が見つからない場合、リン

株式会社日昇テクノロジー

クエラーが発生します。

この例では、単一の hello.c ファイルのみがありますが、C 標準ライブラリの printf 関数を呼び出しているため、リンカーはそれを printf 関数とリンクし、最終的な実行可能ファイルを生成します。

リンクには 2 種類あります：

- 動的リンク：GCC のコンパイル時のデフォルトオプションです。動的とは、アプリケーションの実行時に外部のコードライブラリをロードすることを意味します。例えば、printf 関数の C 標準ライブラリ*.so ファイルは Linux システムの特定の場所に保存されており、hello プログラムの実行時に*.so ファイルの内容を呼び出します。これにより、異なるプログラムがコードライブラリを共有できるため、動的リンクで生成されたプログラムは比較的小さく、メモリ使用量も少なくなります。

- 静的リンク：--static オプションを使用してリンクする場合です。静的リンクは、コンパイル段階で使用される全てのライブラリを自身の実行可能プログラムにパッケージングします。したがって、静的リンクの利点は、良好な互換性を持ち、外部環境に依存しないことですが、生成されるプログラムは比較的大きくなります。

以下のコマンドを実行して、静的リンクと動的リンクの違いを体験してみてください：

```
# hello.o が存在するディレクトリで以下のコマンドを実行
# 動的リンクで hello という名前の実行可能ファイルを生成
gcc hello.o -o hello
#C ファイルを直接使用して一歩で生成することもできます。上記のコマンドと同等です。
gcc hello.c -o hello
# 静的リンクで hello_static という名前の実行可能ファイルを生成
gcc hello.o -o hello_static –static
C ファイルを直接使用して一歩で生成することもできます。上記のコマンドと同等です。
gcc hello.c -o hello_static –static
```

株式会社日昇テクノロジー

```
cat@lubancat:~/test$ ls -lh
total 668K
-rwxrwxr-x 1 cat cat  18K Sep  6 15:03 drm
-rw-rw-r-- 1 cat cat 3.0K Sep  6 14:55 drm.c
-rw-rw-r-- 1 cat cat 1.7K Sep  6 15:38 framebuffer.c
-rwxrwxr-x 1 cat cat 9.1K Sep  7 08:56 hello
-rw-rw-r-- 1 cat cat  187 Apr 21 20:55 hello.c
-rw-rw-r-- 1 cat cat  17K Sep  6 17:54 hello.i
-rw-rw-r-- 1 cat cat 1.8K Sep  6 17:58 hello.o
-rw-rw-r-- 1 cat cat  819 Sep  6 17:54 hello.s
-rwxrwxr-x 1 cat cat 595K Sep  7 08:56 hello_static
cat@lubancat:~/test$
```

図からわかるように、動的リンクで生成された hello プログラムはわずか 9.1KB で、静的リンクで生成された hello_static プログラムは 595KB にもなります。

Ubuntu では、ldd ツールを使用して動的ファイルのライブラリ依存関係を確認できます。以下のコマンドを試してみてください：

```
# hello が存在するディレクトリで以下のコマンドを実行

ldd hello

ldd hello_static
```

```
cat@lubancat:~/test$ ldd hello
        linux-vdso.so.1 (0x0000007fafa94000)
        libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6 (0x0000007faf8cc000)
        /lib/ld-linux-aarch64.so.1 (0x0000007fafa64000)
cat@lubancat:~/test$ ldd hello_static
        not a dynamic executable
cat@lubancat:~/test$
```

動的リンクで生成された hello プログラムは、ライブラリファイル linux-vdso.so.1、libc.so.6、および ld-linux-aarch64.so.1 に依存しています。ここでの libc.so.6 は、プログラムで呼び出した printf ライブラリ関数が含まれている C 標準ライブラリです。

静的リンクで生成された hello_static は外部ライブラリファイルに依存していません。

12.4 HelloWorld アドバンス版-1

```
1 #C 言語側
2 int main(int xxx, char **xxxx)
3
4 # コマンドラインから実行
5 ./hello aaa bbb ccc
```

main 関数では、引数を渡すこともできます。渡されたすべての引数は保存されます。

- int xxx: 引数の数を保存します。上の例では xxx = 4 です。

株式会社日昇テクノロジー

- char **xxxx: [渡された引数を指す文字列配列へのポインタを保存します。]

1. argv[0]="./hello"
2. argv[1]="aaa"
3. argv[2]="bbb"
4. argv[3]="ccc"

LubanCat ボード上でエディタを使用して hello_arg.c という名前のファイルを新規作成し、以下のサンプルコードを入力して hello_arg.c ディレクトリに保存します。

リスト 4: base_linux/hello/hello_arg/hello_arg.c

```
1 #include <stdio.h>
2 /* 実行コマンド: ./hello lubancat
3 * argc = 2
4 * argv[0] = ./hello
5 * argv[1] = lubancat
6 */
7 int main(int argc, char **argv)
8 {
9 int i;
10 if(argc >= 2){
11 printf("you enter: ¥"");
12 // 各引数をプリント
13 for(i = 0; i<argc; i++)
14 printf("%s ", argv[i]);
```

株式会社日昇テクノロジー

15 printf("¥"¥n");

16 // 引数の数を印刷

17 printf("you enter %d strings¥n", argc);

18 }

19 else

20 printf("hello, world! This is a C program.¥n");

21 return 0;

22 }

12.4.1 コンパイルして実行

```
1 # gcc を使用して hello_arg.c を hello プログラムにコンパイル

2 gcc hello_arg.c -o hello

3

4 # 異なる値で試してみましょう

5

6 ./hello

7 ./hello lubancat

8 ./hello I am pc!

9 ./hello I delete king of honor
```

以下のように:

株式会社日昇テクノロジー

```
root@lubancat:/home/cat/all_test/hello# ^C
root@lubancat:/home/cat/all_test/hello# ./hello
hello, world! This is a C program.
root@lubancat:/home/cat/all_test/hello# ./hello lubancat
you enter: "./hello lubancat "
you enter 2 strings
root@lubancat:/home/cat/all_test/hello# ./hello I am pc!
you enter: "./hello I am pc! "
you enter 4 strings
root@lubancat:/home/cat/all_test/hello# ./hello I delete king of honor
you enter: "./hello I delete king of honor "
you enter 6 strings
root@lubancat:/home/cat/all_test/hello#
```

注意: 入力された引数はすべて文字列形式で保存されます。数字を入力する場合は、数字を文字列に変換してから使用する必要があります。

12.5 HelloWorld の進化版-2

コマンド`ls`のように、異なるオプションを後ろに追加することで異なる機能を実現できるように、オプション設定を追加しました。エディタを使用して`hello_opt.c`という名前のファイルを新規作成し、以下のサンプルコードを入力して`hello_opt.c`ディレクトリに保存します。このプログラムは主に`<getopt.h>`ライブラリと`getopt`関数に依存しています。

リスト 5: base_linux/hello/hello_opt/hello_opt.c

```
1 #include <stdio.h>
2 #include <getopt.h>
3 #include <stdlib.h>
4 #include <string.h>
5 // ヘルプ情報を表示
6 void usage(const char *argv_0)
7 {
8 printf("¥nUsage %s: [-option] ¥n", argv_0);
9 printf("[-a] hello!¥n");
10 printf("[-b] i am lubancat¥n");
```

株式会社日昇テクノロジー

```
11 printf("[-c<str>] str¥n");

12 printf("[-d<num>] '*'の数を表示 (num<100)¥n");

13 printf("[-h] ヘルプを表示¥n");

14 exit(1);

15 }

18 // n 個の'*'を表示

19 void d_option(char *num_str)

20 {

21 int num,i;

22 int 10,1;

23 10 = (char)num_str[0] - 48 ;

24 1 = (char)num_str[1] - 48 ;

25 num = 10*10+1;

26 for(i = 0 ;i <num;i++ )

27 printf("*");

28 printf("*¥n");

29 }

32 int main(int argc,char **argv)

33 {

34 int i;

35 int opt;
```

株式会社日昇テクノロジー

```
36 // オプションをループで取得
37 while((opt = getopt(argc, argv, "c:d:abh")) != -1) { // コロンが付いた文字は引数が必要
38 switch (opt) {
39 case 'a':
40 printf("hello!¥n");
41 break;
43 case 'b':
44 printf("i am lubancat¥n");
45 break;
47 case 'c':
48 if(optarg)
49 if(optarg[0] == '-')
50 usage(argv[0]);
51 else
52 printf("%s¥n",optarg);
53 else
54 usage(argv[0]);
55 break;
57 case 'd':
58 if(optarg) // →引数がない場合
59 if(optarg[0] == '-') // →'-'を引数に取れない
60 usage(argv[0]);
61 else
```

株式会社日昇テクノロジー

```
62 if(strlen(optarg) < 3) // 100 未満であること
63 d_option(optarg);
64 else
65 usage(argv[0]);
66 else
67 usage(argv[0]);
68 break;
70 default:
71 usage(argv[0]);
72 break;
73 }
74 }
76 return 0;
77 }
```

このプログラムは<getopt.h>ライブラリに主に依存しており、getopt 関数を使用しています。

株式会社日昇テクノロジー

12.5.1 コンパイルと実行

```
1 # gcc を使用して hello_arg.c を hello プログラムにコンパイル

2 gcc hello_opt.c -o hello

3

4 # 異なる値を試してみましょう

5

6 ./hello

7 ./hello -a

8 ./hello -b

9 ./hello -c xxxx

10 ./hello -d num

11 ./hello -abcd

12 … … … … … … … …
```

以下のように:

```
root@lubancat:/home/cat/all_test/hello#
root@lubancat:/home/cat/all_test/hello# gcc hello_opt.c -o hello
root@lubancat:/home/cat/all_test/hello# ./hello -a
hello!
root@lubancat:/home/cat/all_test/hello# ./hello -b
i am lubancat
root@lubancat:/home/cat/all_test/hello# ./hello -c Saul-Goodman
Saul-Goodman
root@lubancat:/home/cat/all_test/hello# ./hello -d 50
**************************************************
root@lubancat:/home/cat/all_test/hello# ./hello -abc abcdefg
hello!
i am lubancat
abcdefg
root@lubancat:/home/cat/all_test/hello# ./hello -abcd 12
hello!
i am lubancat
d
root@lubancat:/home/cat/all_test/hello# ./hello -d25
*************************
root@lubancat:/home/cat/all_test/hello# ./hello -cSaul_Goodman
Saul_Goodman
root@lubancat:/home/cat/all_test/hello#
```

株式会社日昇テクノロジー

第 13 章 Linux システム下のプログラム

前章では GCC コンパイルツールチェーンを紹介するために、Hello World プログラムの核心に触れませんでした。ここで疑問に思うかもしれません：Hello World プログラムに何の核心があるのでしょうか？しかし、Hello World プログラムを軽視すると、大きな間違いを犯すことになります。

このようにシンプルなプログラムが、数え切れないほど多くの人々をプログラミングの世界に導いてきました。しかし、シンプルな事柄の背後にはしばしば複雑なメカニズムが隠されています。Hello World プログラムを深く考えると、多くの問題がリスト面上はシンプルに見えるかもしれませんが、実際には非常に明確な思考がないことに気づきます。特に、マイクロコントローラから Linux 開発を学び始めた人たちにとって、マイクロコントローラ下の Hello World プログラムに関する底辺の知識があるため、Linux システム開発を学ぶ際に、しばしば無意識のうちにその知識を Linux システムの Hello World プログラムに適用し、2 つのプログラム間の運行メカニズムが全く異なることを知らずにいます。

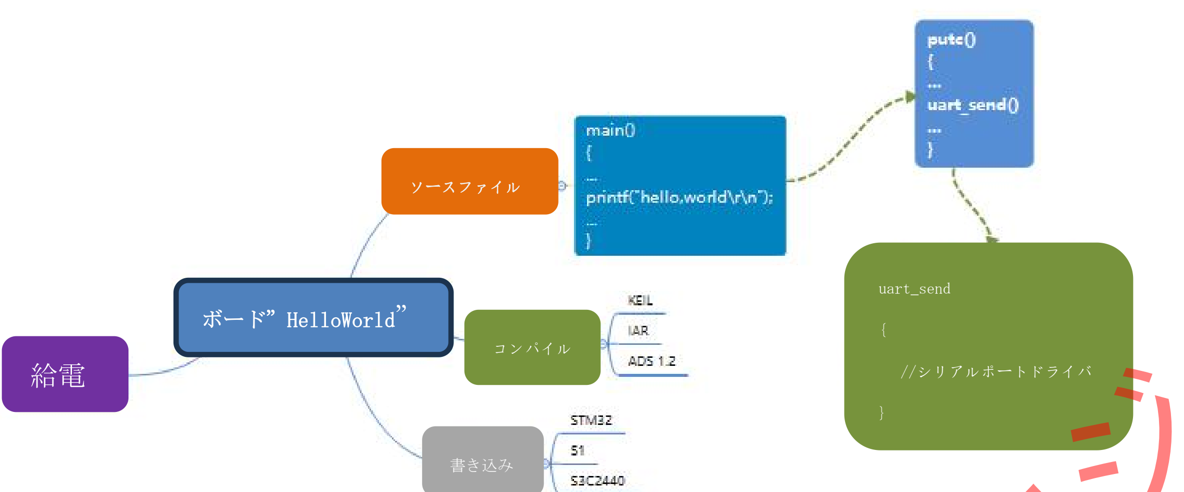
Hello World プログラムは典型的な例です。Linux システム下での運行メカニズムを理解することは、今後他のプログラムを開発する上で非常に良い参考になります。特に、プログラムのコンパイルやリンクでエラーが発生したときに、問題がどの段階で発生しているかを明確に判断できます。

以下では、"裸のマイクロコントローラ下の Hello World"と比較して、"Linux システム下の Hello World"を詳しく説明します。これからの分析内容では、全体の実行プロセスをしっかりと把握してください。Linux システムに深く入り込む前に、多くの細部に陥ることは避けてください。学習の初期段階で、大局的な思考を身につけることが重要です。

13.1 マイクロコントローラでの Hello World

マイクロコントローラ内で Hello World プログラムを実装する手順は複雑ではありません。以下の画像は開発プロセスを包括的に示しており、非常に明確です：

株式会社日昇テクノロジー



・第一歩目：ソースコードの作成であり、printf 関数の実装が重要なポイントです。これはマイクロコントローラのシリアルポートドライバーに依存します。

・第二歩目：いくつかの統合開発環境を利用してプログラムをコンパイルします。ワンクリックでコンパイルし、コンパイルやリンクに関する知識を学ぶ必要はありません。

・第三歩目：書き込みツールを使って具体的なチップに書き込みます。ワンクリックで書き込み、チップのフラッシュや様々な起動方式を学ぶ必要はありません。

・第四歩目：電源を入れてボードを起動し、シリアルポートから「Hello World」という文字列を出力します。

13.2 Linux システムでの Hello World

上述したマイクロコントローラでの Hello World 開発には、多くのコンパイル、リンク、書き込みの知識が関係していますが、この章の主題ではないため、ここで詳しく説明するのは適切ではありません。一方、Linux システムでの Hello World には、参考になる意義が非常に大きく、今後の他のコースの学習に役立つ良い指導が期待できるため、ここではその全体的な作業原理を可能な限り詳細に分析します。

まず、Linux システムでの Hello World プログラムの流れを見てみましょう。以下のように、

株式会社日昇テクノロジー

hello.c ① hello.i ② hello.s ③ hello.o ④ hello应用程序
GCC のコンパイルプロセス
Shell
⑤ fork() ⑥ execev() ⑦ sys_execve() ⑧ do_execve() ⑨ load_elf_binary()
Linux Kernel
ld-linux-xx.so
glibc ライブラリに関連
_start 12 _libc_start_main()
printf("hello,world\r\n!") main()
_exit()
glibc ライブラリに関連

プログラムのコンパイルと実行プロセスは、上から下の順序に従って、4 つの主要な部分に分けることができます：

・第一部分、上の浅緑色の外枠はプログラムのコンパイルをリストしています。

・第二部分、浅黄色の外枠は Linux カーネルが提供するサービスをリストしています。

・第三部分、オレンジ色の外枠は glibc ライブラリが提供するサービスをリストしています。

・第四部分、浅灰色の外枠はユーザープログラムをリストしています。

以下では、各サブステップについてさらに説明します：

1. hello.c の前処理、主にプログラム内のファイルインクルード、マクロ定義の処理、条件コンパイルを処理します。

株式会社日昇テクノロジー

2. C ファイルをアセンブリファイル(.s)にコンパイルし、ここでは語彙分析、構文分析、意味分析、中間コードの生成、コードの最適化などの作業が行われます。

3. アセンブリファイル(.s)を再配置可能ファイル(.o)にコンパイルします。

4. 再配置可能ファイル(.o)を実行可能ファイルにリンクします。リンクには静的リンクと動的リンクがあります。

・静的リンク：コンパイル段階で使用されるすべてのライブラリを実行可能プログラムにパッケージ化します。互換性が高く外部環境に依存しないが、生成されるプログラムが大きくなります。

・動的リンク：アプリケーション実行時に、リンカーが外部の共有ライブラリをロードし、共有ライブラリと動的コンパイルプログラム間のリンクを完了します。異なるプログラムがコードライブラリを共有し、メモリスペースを節約できます。

5. コンソールで./hello コマンドを入力すると、Shell はこのプログラムを実行するために新しいプロセスを作成します。fork()関数は新しいプロセスを作成するために使用されます。ここでのプロセスは、プログラムのコンテナとして簡単に理解できます。

6. exeve()関数は、前のステップで新しく作成されたプロセスに実行可能プログラム(hello)を充填すると理解できます。

7. sys_execve()関数は Linux のシステムコールで、exeve()関数によって呼び出されます。ここでのシステムコールは、オペレーションシステムがユーザーに提供する最下層のインターフェースと理解できます。

8. do_execve()関数は sys_execve()関数の核心です。

9. load_elf_binary()関数は、hello プログラムをファイルシステムからメモリに読み込み、それが動的リンクの実行可能プログラムかどうかを判断します。そうでない場合は、静的リンクのファイルかどうかをさらに判断します。

10. ld-linux-xx.so は glibc ライブラリの動的リンカです。hello プログラムが動的リンクプログラムで

株式会社日昇テクノロジー

ある場合、この動的リンカは共有ライブラリをロードし、共有ライブラリとプログラムのリンク作業を完了し、hello プログラムの実行準備を始めます。

11. 逆に、hello プログラムが静的にコンパイルされたプログラムである場合、共有ライブラリのロードやリンクは必要なく、直接 hello プログラムの実行準備を始めます。

・10 と 11 のステップがそれぞれ実行された後、hello プログラムの実行が始まります。_start はプログラムの実際の入口であり、このシンボルは glibc にあります。つまり、プログラムの実際の入口は glibc にあります。

12. __libc_start_main()も glibc の関数で、ユーザープログラムの実行前にいくつかの初期化作業を行います。

13. ユーザープログラムの main()関数を呼び出し、printf 印刷関数の実行を開始します。

14. プログラムの実行が完了した後、glibc ライブラリの_exit()関数を呼び出して、現在のプロセスを終了します。

このプロセスを大まかに分析し終えたところで、マイクロコントローラと Linux システムでの「Hello World」を比較すると、オペレーションシステムがために大量の作業を行っていることが明らかになります。さらに、メモリスペースを節約するために、プログラムのリンクという非常に基本的な作業を glibc の動的リンカに委ねるなど、マイクロコントローラ開発では実現できない効率的な開発方法を採用しています。

ここに記載されている内容について多くの人が知らないことや疑問を持っているかもしれませんが、これらの内容を詳しく説明することはできません。各サブステップの背後には巨大な内容があります。ここで多くの時間を費やすと、学習の進行が大幅に遅れる可能性があります。高度な内容に深く入るためには、段階を追って学ぶ必要があります。

しかし、一方で、プログラム実行の全体的な概要を頭に入れておくと、今後の学習やプログラミングに向けて指針と進む方向が得られます。プログラム開発プロセスの各ステップを理解することができます。これがこの章の意義です。

株式会社日昇テクノロジー

第 14 章 vscode での便利なデバッグ開発

直接ボード上でデバッグやコンパイルを行うだけでなく、ここではもっと便利な開発デバッグ方法を提供します。それは vscode を使用して SSH で LubanCat にリモートログインし、魯班猫に対して二次開発を行う方法です。

注意: この機能を使用するには、コンピュータとボードが SSH でログインできる状態である必要があります。

SSH ログインが設定されていない場合は、以下の章「SSH 端末ログイン」で設定を参照してください。

14.1 環境設定

まず、環境を設定する必要があります。環境設定で静的 IP およびパスワードなしでのログインを設定した場合、後の開発で素早く接続し、迅速に開発することができ、開発効率が大幅に向上します。

14.1.1 vscode の使用

vscode ダウンロード公式サイト「vscode 公式サイト」

vscode 初心者向けチュートリアル：https://blog.csdn.net/weixin_52212950/article/details/124693906

14.1.2 LubanCat の IP アドレスの設定確認

```
1 # コマンドを実行して IP を確認
2 ifconfig
```

以下の画像のように

株式会社日昇テクノロジー

```
cat@lubancat:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.103.156  netmask 255.255.255.0  broadcast 192.168.103.255
        inet6 fe80::a093:53ff:fe6d:3c69  prefixlen 64  scopeid 0x20<link>
        ether a2:93:53:6d:3c:69  txqueuelen 1000  (Ethernet)
        RX packets 818406  bytes 133075412 (133.0 MB)
        RX errors 0  dropped 13533  overruns 0  frame 0
        TX packets 996749  bytes 1086875595 (1.0 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 39

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 179775  bytes 6457137183 (6.4 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 179775  bytes 6457137183 (6.4 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

cat@lubancat:~$
```

192.168.103.156 は LubanCat の IP アドレスです

14.1.3 vscode でボードに接続

ボードに接続する前に、ボードとホスト間のネットワークが正常に通信できることを確認する必要があります。PC 上でコマンドプロンプトまたは PowerShell を開きます

```
1 # 実行
2 ping 192.168.xxx.xxx
3 #IP アドレスにはボードの IP アドレスを入力
```

以下の画像のように

```
C:\Users\zhang>ping 192.168.1.5

192.168.1.5 に ping を送信しています 32 バイトのデータ:
192.168.1.5 からの応答: バイト数 =32 時間 =1ms TTL=64
192.168.1.5 からの応答: バイト数 =32 時間 <1ms TTL=64
192.168.1.5 からの応答: バイト数 =32 時間 <1ms TTL=64
192.168.1.5 からの応答: バイト数 =32 時間 <1ms TTL=64

192.168.1.5 の ping 統計:
    パケット数: 送信 = 4、受信 = 4、損失 = 0 (0% の損失)、
ラウンド トリップの概算時間 (ミリ秒):
    最小 = 0ms、最大 = 1ms、平均 = 0ms
```

ボードとホスト間で正常にネットワーク通信ができることを確認したら、インストール済みの vscode を

株式会社日昇テクノロジー

開いて基本操作を設定した後、左側の拡張機能をクリックして Remote-SSH プラグインをダウンロードします。

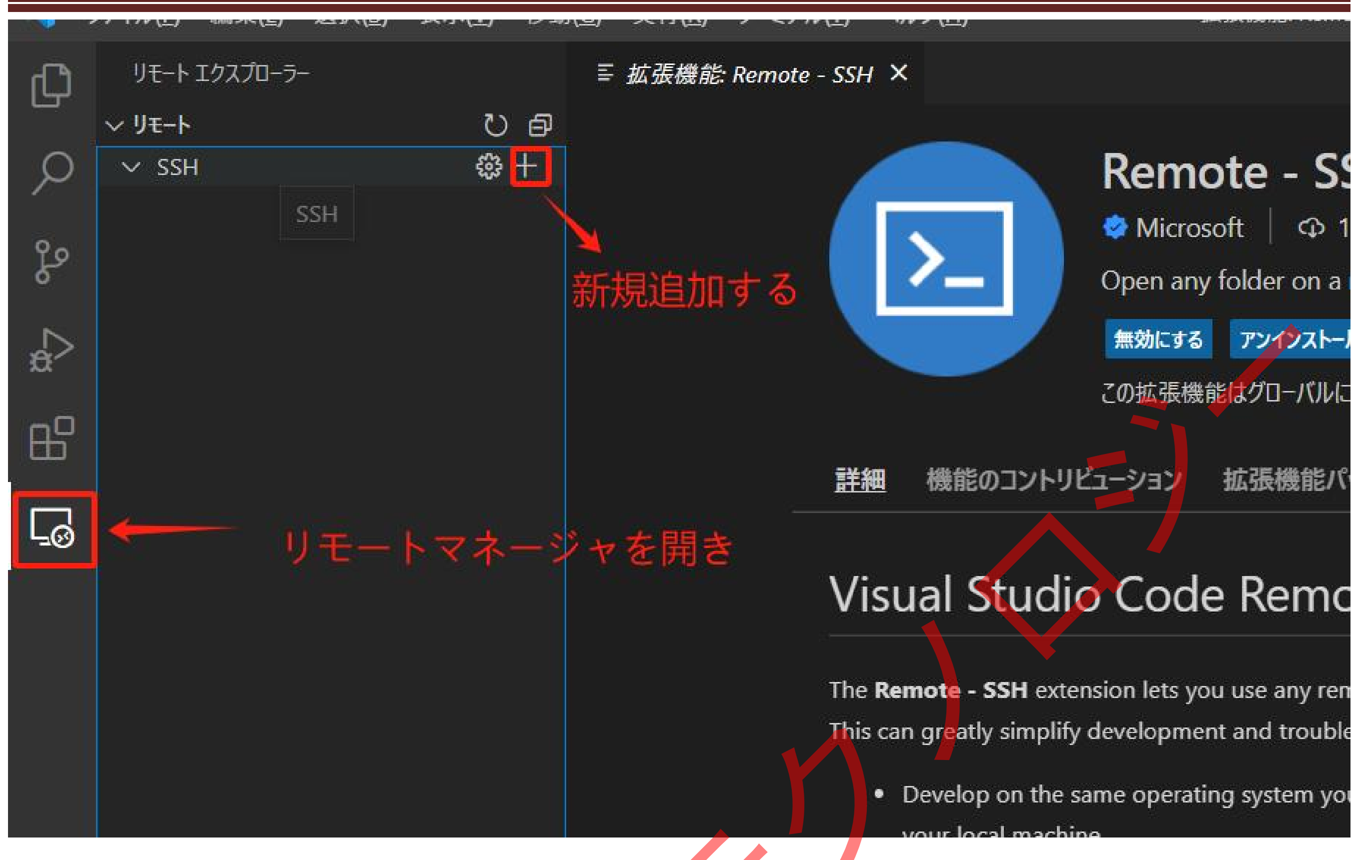
以下の画像のように：

インストールが完了すると、SSH でリモート接続してボードに接続できるようになります。

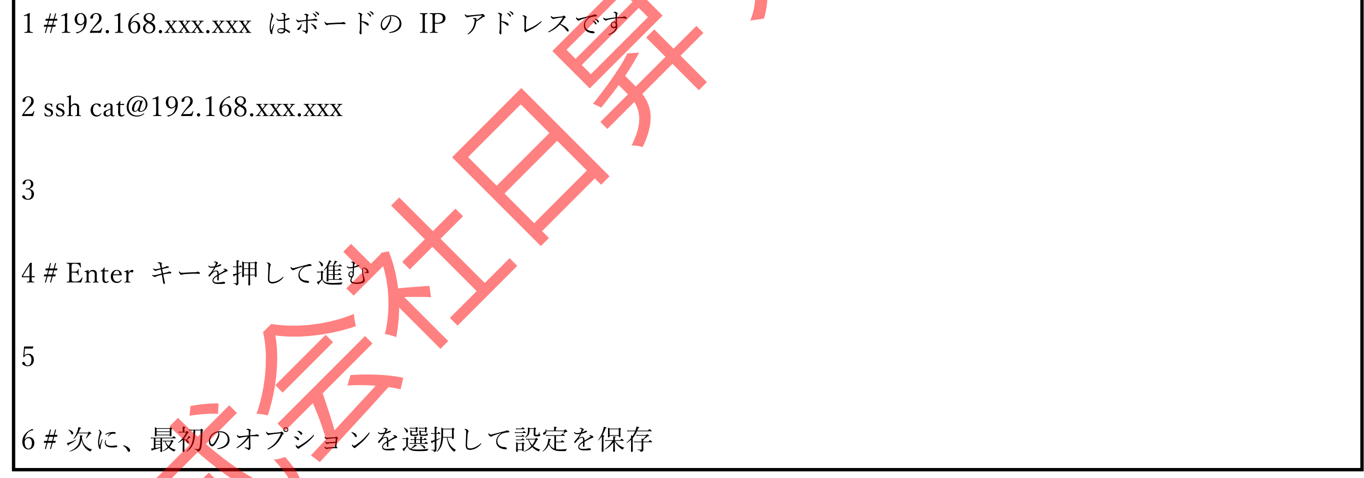
以下の画像のように：

そして、上部に表示される内容を入力します。以下を入力します。

株式会社日昇テクノロジー



その後、上部に表示される入力欄があり、以下のように入力



1 #192.168.xxx.xxx はボードの IP アドレスです

2 ssh cat@192.168.xxx.xxx

3

4 # Enter キーを押して進む

5

6 # 次に、最初のオプションを選択して設定を保存

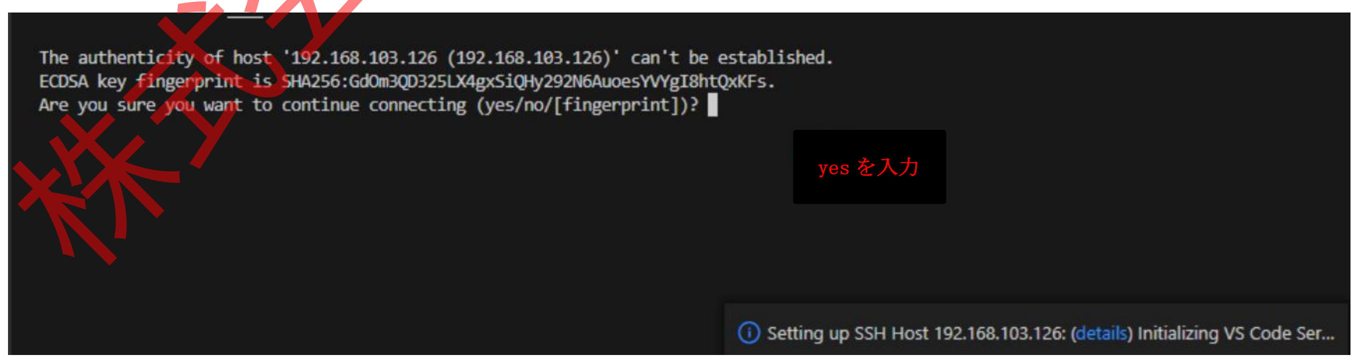
ボードが開きます、以下の画像のように

株式会社日昇テクノロジー

その後、vscode は Linux、Windows、mac から選択するように求められますが、ここでは Linux を選択します。

パスワードの入力が求められた場合は、パスワードを入力します。

初めてログインする場合、vscode はログインを求め、ターミナルに「yes」と入力します。

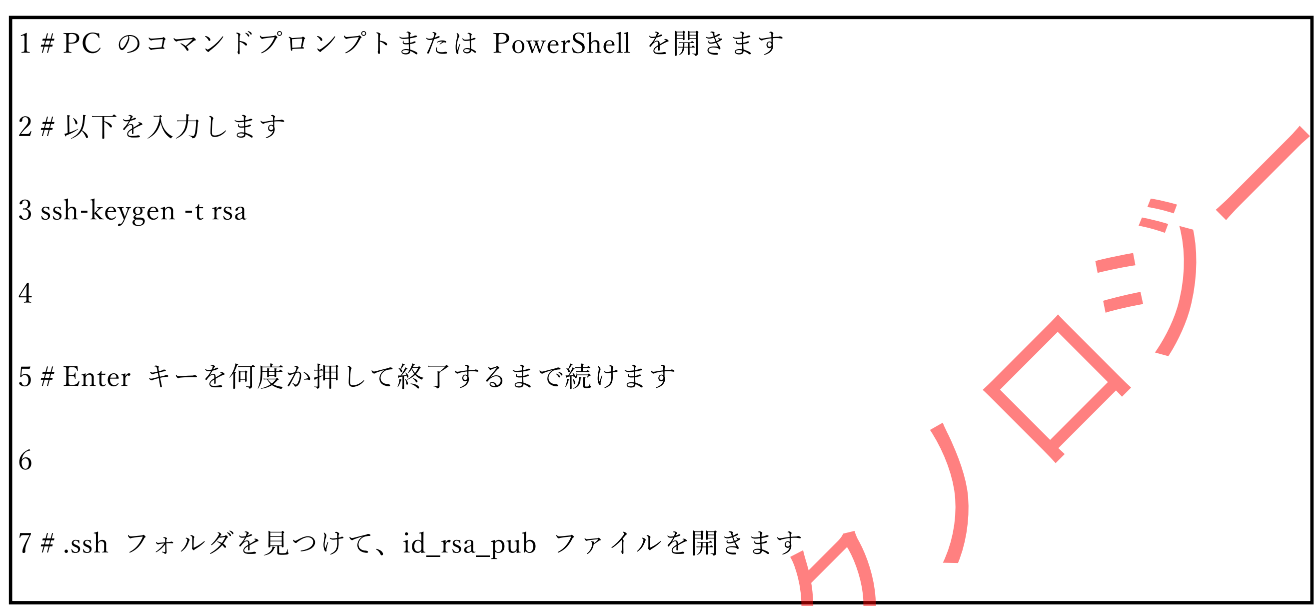


その後、フォルダを開くことで開きたいプロジェクトやその他を選択することができます。

株式会社日昇テクノロジー

14.1.4 パスワードなしでログイン

vscode を使用する際に毎回ログインするのが面倒な場合は、以下の方法で解決できます。



1 # PC のコマンドプロンプトまたは PowerShell を開きます

2 # 以下を入力します

3 ssh-keygen -t rsa

4

5 # Enter キーを何度か押して終了するまで続けます

6

7 # .ssh フォルダを見つけて、id_rsa_pub ファイルを開きます

以下の画像のように

株式会社日昇テクノロジー

メモ帳などで開き、内容をコピーします。

```
1 # ボードに接続します

2 # cat ユーザーでログインします

# コマンドラインに以下を入力します

4

5 ssh-keygen -t rsa

6

7 # Enter キーを何度か押して終了するまで続けます

8 # /home/cat ディレクトリに.ssh フォルダが生成されていることが確認できます
```

```
cat@lubancat:~$ ls -ag
total 56
drwxr-xr-x 6 cat  4096 Aug 19 13:41  .
drwxr-xr-x 3 root 4096 Aug 10 14:16  ..
-rw------- 1 cat    54 Apr 21 20:55  .Xauthority
-rw-r--r-- 1 cat  1600 Apr  9  2020  .Xdefaults
-rw-r--r-- 1 cat   220 Feb 25  2020  .bash_logout
-rw-r--r-- 1 cat  3771 Feb 25  2020  .bashrc
drwx------ 2 cat  4096 Apr 21 20:55  .cache
drwxr-xr-x 5 cat  4096 Apr 21 20:55  .config
-rw-r--r-- 1 cat   807 Feb 25  2020  .profile
drwx------ 2 cat  4096 Aug 19 13:41  .ssh
-rw------- 1 cat   870 Aug 19 13:41  .viminfo
drwxrwxr-x 5 cat  4096 Aug 19 12:00  .vscode-server
-rw-rw-r-- 1 cat   183 Aug 19 12:00  .wget-hsts
-rw-r--r-- 1 cat    14 Apr  9  2020  .xscreensaver
-rw-rw-r-- 1 cat     0 Aug 18 15:48 '01'$'\344\270\203\351\207\214\351\246\231''.mp3'
-rw-rw-r-- 1 cat     0 Aug 18 15:48 '02'$'\344\270\203\351\207\214\351\246\231''.wav'
-rw-rw-r-- 1 cat     0 Aug 18 15:48 '03'$'\344\270\203\351\207\214\351\246\231''.flac'
cat@lubancat:~$
```

そのフォルダに移動すると、生成された公開鍵と秘密鍵を確認できます

```
cat@lubancat:~/.ssh$ ls -ag
total 16
drwx------ 2 cat 4096 Aug 19 13:58 .
drwxr-xr-x 6 cat 4096 Aug 19 13:58 ..
-rw------- 1 cat 2602 Aug 19 13:40 id_rsa
-rw-r--r-- 1 cat  566 Aug 19 13:40 id_rsa.pub
```

```
1 # その後、authorized_keys ファイルを作成します

2 vi authorized_keys

3

4 # 先ほどメモ帳からコピーした内容を貼り付けて、保存して終了します
```

株式会社日昇テクノロジー

これでパスワードなしでのログイン操作が完了しました。

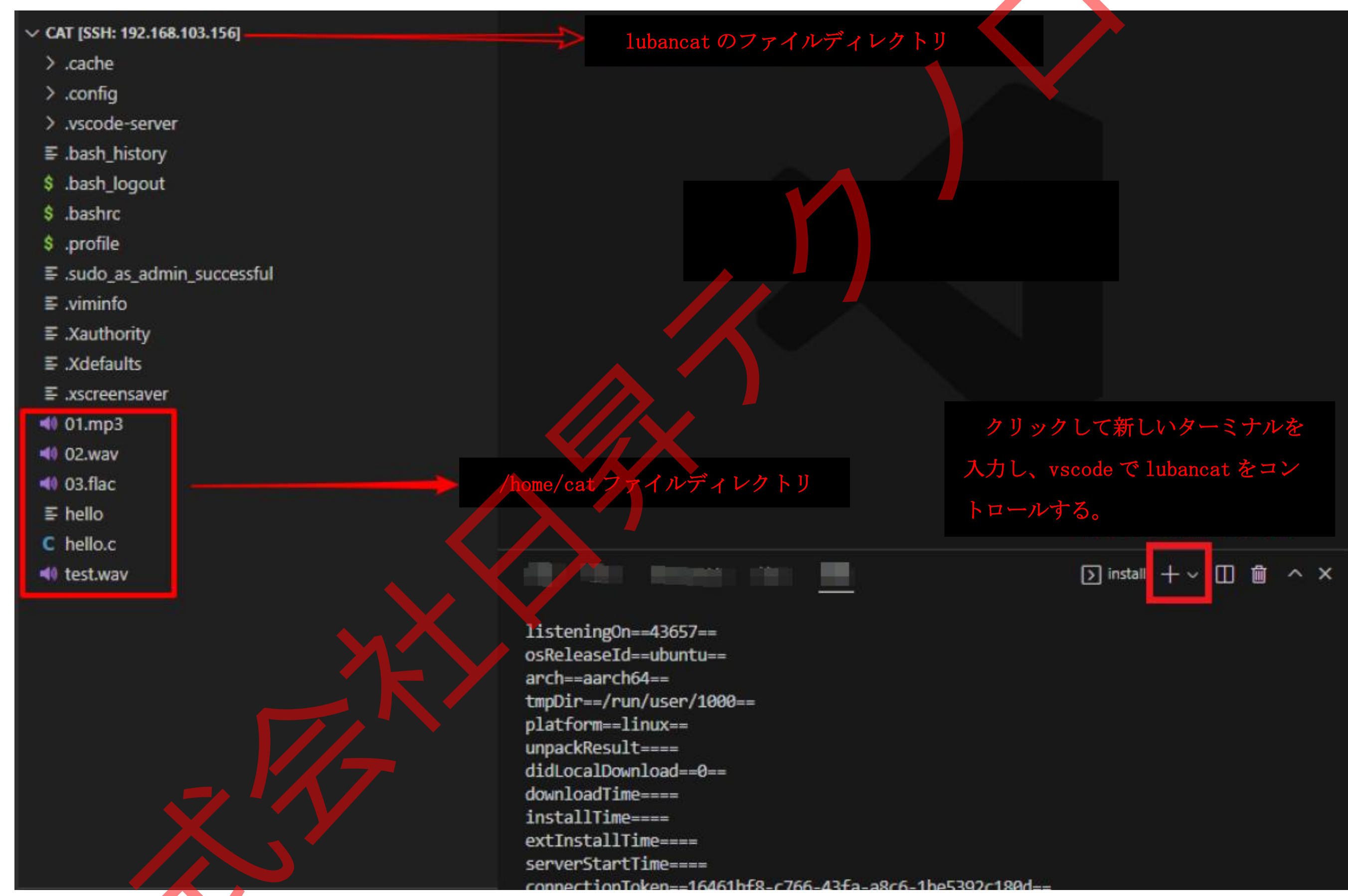
この記事の一部はこちらを参考にしています：

https://blog.csdn.net/qq_44571245/article/details/123031276

この記事の操作に疑問を感じた場合は、上記の記事を参照してください。

14.1.5 ボードへの接続

lubancat のファイルディレクトリ lubancat のファイルディレクトリ lubancat のファイルディレクトリ



機能紹介は以下の画像のように

株式会社日昇テクノロジー

ファイルディレクトリ

ファイル編集

拡張プラグイン：git、makefile、gcc、python など。

lubancat の操作，方法はコマンドラインと同様です。

```c
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    printf("meow!\n");
    return 0;
}
```

コンパイルと実行操作を実行します

以上が vscode を使用した便利な開発のチュートリアルです。このような操作は C 言語に限らず、C++、Python、Java、Go などでも vscode で対応するプラグインを見つけて開発をサポートできます。

また、vscode のプラグインを利用してデバッグも可能です。

株式会社日昇テクノロジー

vscode にはまだ発見されていない多くの機能があります。

第 15 章 Makefile の紹介

15.1 Makefile とは何か

これまでのいくつかの章で使用した例のコードは hello.c ファイルのみで非常にシンプルだったため、以下のコマンドを直接実行してコンパイルすることも非常に便利でした。

ファイルをコンパイルする

```
1 gcc hello.c -o hello
```

しかし、コースが徐々に深まるにつれて、プロジェクトエンジニアリングにはますます多くの C ファイルと H ヘッダファイルが存在することになります。多くの C ソースファイルと H ヘッダファイルがプロジェクトに存在する場合、コンパイラのコマンドを直接使用するのは非常に面倒です。ファイル名を入力するだけで気分が悪くなることもあります。例えば、以下の例のように:

複数のファイルをコンパイルする

```
1 gcc hello.c aaa.c bbb.c -o hello
```

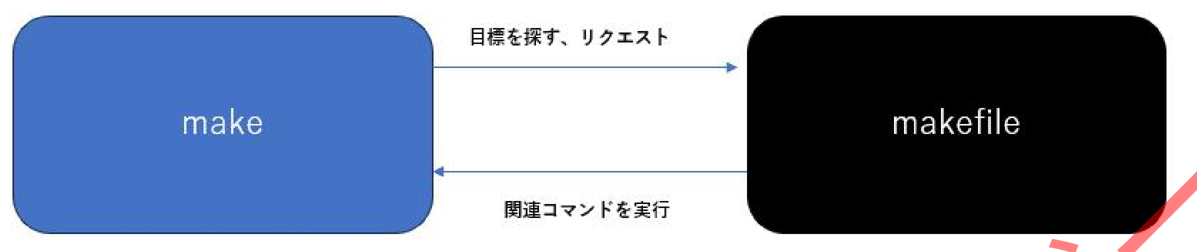
さらに、ファイルを 1 つだけ変更した場合でも、全てのファイルを再コンパイルする必要があり、多くの開発時間が無駄になります。この問題を解決する最良の方法は、プロジェクトのコンパイルルールを記述し、コンパイラに自動的にこれらのルールに従ってコンパイルさせることです。解決策は make と Makefile の使用です。これら 2 つのツールは連携して使用されます。以下に紹介します:

・make ツール：プロジェクト内で変更されたファイルを見つけ出し、依存関係に基づいて修正の影響を受ける他の関連ファイルを見つけ出し、これらのファイルをルールに従って個別にコンパイルすることができます。これにより、プロジェクトの全ファイルを再コンパイルする必要がなくなります。

・Makefile ファイル：上記のルールや依存関係は主にこの Makefile ファイル内で定義されています。ファイルの依存関係を適切に定義した後、make ツールは正確にコンパイル作業を行うことができます。

株式会社日昇テクノロジー

それらの関係は以下の図に示されています:



上記の紹介から、プロジェクトエンジニアリングを管理することは、実質的にはプロジェクトファイル間の依存関係を管理することであると知ることができます。そのため、Makefile を学習して使用する際には、その依存指向の考え方をしっかりと掴む必要があります。Makefile 内の全ての複雑で難解な構文は、依存問題をより良く解決するために存在していることを心に留めておく必要があります。その本質的な目的を理解した後、様々な構文を暗記する必要はなくなります。この本質的な目的を思い出すだけで、全てが自然と理解できるようになります。

Makefile を真にマスターしたかどうかの指標は、目標と依存関係の関係を頭の中で明確に理解しているかどうかにあります。あなたの頭が make ツールのように正確に Makefile を解釈し実行できる時、あなたは Makefile の達人です。その目標を目指しています。

さらに、プロジェクトの複雑さがさらに一段階上がると、手作業で Makefile を作成することも面倒になります。その時は、CMake、autotools などのツールを使用して Makefile を生成することができます。実際には、Windows システム下の多くの IDE ツールも、内部的には Makefile のような方法でプロジェクトファイルを組織していますが、グラフィカルインターフェースに包まれており、ユーザーには見えません。

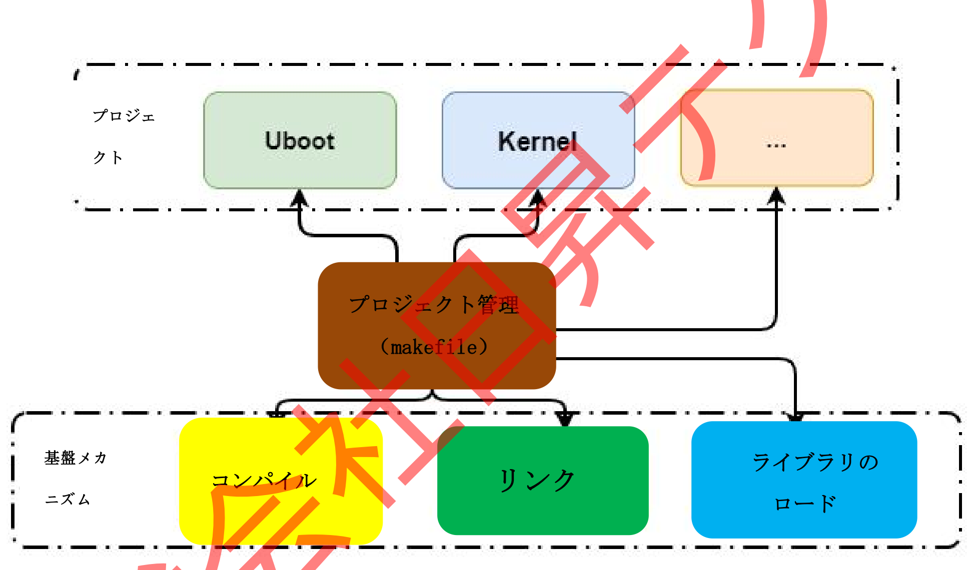
15.2 Makefile 概要

Makefile は、組み込み Linux 開発者にとって、その重要性をどれだけ強調しても過言ではありません。それはプログラマーのプログラミング能力と直接関連しているわけではありませんが、それが示すのは一種のエンジニアリング能力であり、このエンジニアリング能力こそが、プロフェッショナルプログラマー

株式会社日昇テクノロジー

とアマチュアプログラマーを分けるものです。なぜなら、Linux 開発環境下では、あまりにも多くの統合開発環境 IDE がないため、プログラムの全ての制御権が開発者にあり、ソフトウェアのコンパイル、リンク、ロードに関して深い理解を持っていなければ、本当の意味でプログラムをコントロールすることはできません。前の章で底層に関連する内容を簡単に紹介しましたが、それだけでは十分ではありません。今後のコースでは、さらに深く掘り下げていきます。

また、今後のコースでは、Uboot の移植開発、Linux カーネルの移植開発、およびその他のオープンソースプロジェクトについて順を追って説明していきます。Makefile の基礎がしっかりしていない場合、プロジェクトの分析方法がわからなかったり、プログラムのアーキテクチャが整理できなかったり、プログラム機能の修正ができなかったりする可能性があります。以下の図のように:



次に、Makefile の関連する文法全体について理解を深めましょう。ここで事前に明確にしておく必要がありますが、Makefile は多年にわたる発展を経て、非常に強力な機能を持っていますが、重い歴史的な負担も残しています。そのため、make の各詳細な知識を詳細に紹介することはできません（する必要もありません）。過去の開発経験を参考にして、Makefile の 80%の常用知識点をまとめました。これは、将来的に Uboot、Linux kernel、およびその他のオープンソースプロジェクトを研究するための堅固な基礎を築くことを目的としています。この目標はまだかなり難しいものであり、学ぶべき知識点も少なくありませんが、

株式会社日昇テクノロジー

一連の段階的に難易度が増す小さな実験を通じて、皆さんが Makefile を苦労せずに学べるように支援します。

知識を学ぶ前に、最初に頭の中に知識点の全体的な枠組みを構築することで、さらなる学習を指導するべきです。次に、Makefile の学習ポイントを全体的に見てみましょう。図は以下の通りです：

図には多くの知識点がありますが、ここでは具体的な文法を深く学習するのではなく、各文法が何の問題を解決するために存在するかを皆さんに伝えます。左上隅から始めて説明します：

1. 基本文法 - 目標と依存関係を記述する特定の形式で、Makefile の核心です。

2. 変数 - 特定の情報を記録し、原始情報を繰り返し入力するのを避けます。特に原始情報の手動入力が長い場合に非常に便利です。

3. 分岐判断 - 複数の異なるコンパイルプロセスを柔軟に制御し、異なる属性を容易に対応させます。

4. ヘッダーファイルの依存関係 - ヘッダーファイルの変更を監視します。ヘッダーファイルもプログラムの重要な内容です。

株式会社日昇テクノロジー

5. 暗黙のルール - Makefile のいくつかのデフォルトルールを利用することで、Makefile の作業量を減らすことができます。

6. 自動変数 - Makefile のデフォルトの自動変数を利用することで、Makefile の作業量を減らすことができます。

7. パターンルール - 正規リスト現を柔軟に使用することで、Makefile の作業量を減らすことができます。

8. 関数 - Makefile のさまざまな関数を使用することで、Makefile の機能をより便利に実装することができます。

Makefile の知識点を理解した上で、上記の分析からわかるように、Makefile の核心は基本文法であり、目標と依存関係の関係を記述することです。他の文法の目的は、Makefile の作業量を減らし、よりエレガントで、よりシンプルで、よりメンテナンスしやすい方法で Makefile の機能を実装することです。これはプログラム開発に非常に似ており、機能を実装するだけでなく、プログラムの可読性、拡張性、メンテナンス性なども考慮する必要があります。

次の章では、小さな実験を通じてさらに Makefile を学習します。

第 16 章 Makefile を使ったコンパイル制御

Makefile の詳細な使用方法については、GNU 公式の make マニュアル（https://www.gnu.org/software/make/manual）を参照してください。この章では、Makefile の基本文法について例を挙げて説明します。

LubanCat-RK ボードは出荷時に Makefile ツールが搭載されていないため、自分でダウンロードしてインストールする必要があります。

```
1 sudo apt install make
```

この章のサンプルコードのディレクトリは以下の通りです：base_linux/makefile/

株式会社日昇テクノロジー

コードの場所：

16.1 Makefile の小実験

Makefile の機能を直感的に示すために、例を使って説明します。まず、エディタを使用して「Makefile」という名前のファイルを作成し、以下のコードを入力して保存します。"#"で始まる行はコメントであり、実験をする際には入力する必要はありません。また、「ls -lh」、「touch test.txt」などのコマンドの前には Tab キーを使用し、スペースを使用しないように注意してください。

リスト 1: base_linux/makefile/test1/Makefile

```
1 #Makefile 形式
2 #目標: 依存ファイルまたは他の目標
3 #Tab コマンド 1
4 #Tab コマンド 2
5 #最初の目標、make のデフォルト目標
6 #目標 a、targetc と targetb に依存
7 #実行する shell コマンド ls -lh、ディレクトリ内の内容をリスト表示
8 targeta: targetc targetb
9 ls -lh
10 #目標 b、依存なし
11 #実行する shell コマンド、touch で test.txt ファイルを作成
12 targetb:
13 touch test.txt
14 #目標 c、依存なし
15 #実行する shell コマンド、pwd で現在のパスを表示
16 targetc:
```

株式会社日昇テクノロジー

```
17 pwd
18 #目標 d、依存なし
19 #targetd は targeta、b、c に依存しないため、直接 make を実行しても実行されない
20 #make targetd コマンドで実行可能
21 targetd:
22 rm -f test.txt
```

この Makefile ファイルは、主に 4 つの目標操作を定義しています。それぞれの関係を大まかに理解しましょう：

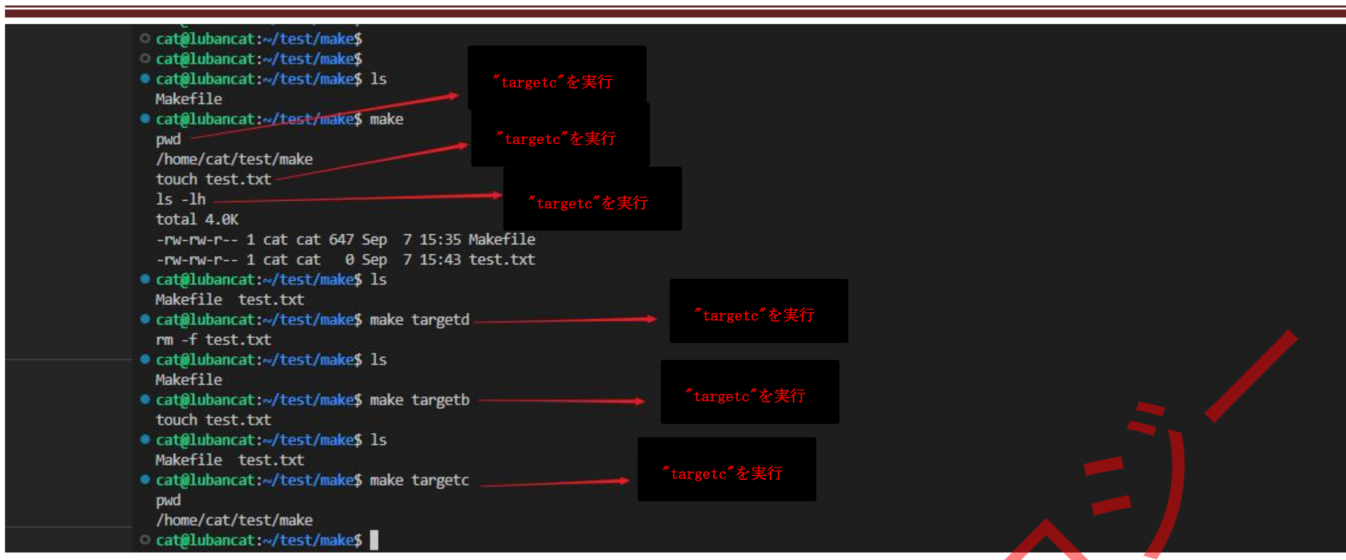
- targeta：Makefile の最初の目標であり、":"の後に記述された内容は、targetc と targetb に依存していることを意味します。そのコマンドは「ls -lh」で、現在のディレクトリの内容をリスト表示します。
- targetb：この目標は他に依存していません。実行するコマンドは「touch test.txt」で、test.txt ファイルを作成します。
- targetc：この目標も他に依存していません。実行するコマンドは「pwd」で、単純に現在のパスを表示します。
- targetd：この目標は他に依存しておらず、実行するコマンドは「rm -f test.txt」で、ディレクトリから test.txt ファイルを削除します。targetb、c とは異なり、他の目標に依存せず、デフォルトの目標でもありません。

次に、この Makefile を使用して様々な make コマンドを実行し、異なる make コマンドの出力を比較することで、Makefile の仕組みを明確に理解できます。Makefile があるディレクトリで以下のコマンドを実行します：

株式会社日昇テクノロジー

```
1 #ホスト上で Makefile があるディレクトリで以下のコマンドを実行

2 #現在のディレクトリの内容を表示

3 ls

4

5 #make コマンドを実行、現在のディレクトリで「Makefile」または「makefile」を検索し、実行

6 make

7

8 #make コマンド実行後の出力を見ると、Makefile に書かれたコマンドが実行されている

9

10 #make コマンド実行後のディレクトリ内容を確認、test.txt ファイルが追加されている

11 ls

12

13 #Makefile の targetd 目標を実行し、確認すると test.txt ファイルがなくなっている

14 make targetd

15 ls

16

17 #Makefile の targetb 目標を実行し、確認すると再び test.txt ファイルが生成されている

18 make targetb

19 ls

20

21 #Makefile の targetc 目標を実行

22 make target
```

株式会社日昇テクノロジー



上図に含まれる原理説明は以下の通りです：

make コマンド：

- ターミナルで make コマンドを実行すると、make は現在のディレクトリ下で「Makefile」または「makefile」という名前のファイルを探し、そのファイルのルールに従って解析し実行します。他のファイルを入力ルールとして指定する場合は、「-f」パラメータを使用して入力ファイルを指定できます。例：「make -f ファイル名」。

- ここで make コマンドが Makefile ファイルを読み込むと、targeta が Makefile の最初のターゲットであることがわかり、デフォルトのターゲットとして実行されます。

- targeta は targetc と targetb に依存しているため、targeta 自体のコマンドを実行する前に、先に targetc と targetb を完了させます。

- targetc のコマンドは pwd で、現在のパスを表示します。

- targetb のコマンドは touch test.txt で、test.txt ファイルを作成します。

- 最後に targeta 自体のコマンド ls -lh を実行し、現在のディレクトリの内容をリスト表示します。その結果、test.txt ファイルが追加されていることが確認できます。

make targetd、make targetb、make targetc コマンド：

- targetd はデフォルトのターゲットではなく、他のどのターゲットにも依存していないため、直接

株式会社日昇テクノロジー

make を実行した場合には targetd は実行されません。Makefile の中の特定のターゲットを個別に実行するには、「make ターゲット名」という構文を使用します。例えば、上の図では「make targetd」、「make targetb」、「make targetc」という指令がそれぞれ実行されています。「make targetd」ターゲットを実行するとき、そのコマンド rm -f test.txt が実行され、test.txt ファイルが削除されます。このプロセスから、make プログラムが Makefile に記述されたターゲットと依存関係に基づいて、目標を達成するために必要な shell コマンドを実行することがわかります。簡単に言うと、Makefile は make プログラムに何をどのように行うかを指示するリストです。

16.2 プログラムの Makefile を使用したコンパイル

16.2.1 複数のファイルを GCC でコンパイルする

次に、Makefile を使用してプログラムのコンパイルを制御します。説明を簡単にするために、前の章での hello.c プログラムを 3 つのファイルに分割して書きます。それぞれ hello_main.c のメインファイル、hello_func.c の関数ファイル、hello_func.h のヘッダファイルで、以下のコードのようになります。

2: base_linux/makefile/test2/hello_main.c

```
1 #include "hello_func.h"

2

3 int main()

4 {

5 hello_func();

6 return 0;

7 }
```

株式会社日昇テクノロジー

3: base_linux/makefile/test2/hello_func.c

```
1 #include <stdio.h>

2 #include "hello_func.h"

3

4 void hello_func(void)

5 {

6 printf("hello, world! This is a C program.¥n");

7 for (int i=0; i<10; i++ ) {

8 printf("output i=%d¥n",i);
```

4: base_linux/makefile/test2/hello_func.h

```
1 void hello_func(void);
```

つまり、hello_main.c の main 関数は hello_func.c ファイルの印刷関数を呼び出し、その印刷関数は hello_func.h ファイルで宣言されています。これは複雑なプロジェクトで一般的なプログラム構造です。

直接 GCC を使用してコンパイルする場合、以下のコマンドを使用する必要があります：

```
1 # ホスト上のサンプルコードディレクトリで以下のコマンドを実行

2 # 最後の"-I ."は現在のディレクトリを含むことを意味します

3 gcc -o hello_main hello_main.c hello_func.c -I .

4

5 # 生成された hello_main プログラムを実行

6 ./hello_main
```

株式会社日昇テクノロジー

```
cat@lubancat:~/test/make/test2$ ls
Makefile  hello_func.c  hello_func.h  hello_main.c
cat@lubancat:~/test/make/test2$ gcc -o hello_main hello_main.c hello_func.c -I .
cat@lubancat:~/test/make/test2$ ls
Makefile  hello_func.c  hello_func.h  hello_main  hello_main.c
cat@lubancat:~/test/make/test2$ ./hello_main
hello, world! This is a C program.
output i=0
output i=1
output i=2
output i=3
output i=4
output i=5
output i=6
output i=7
output i=8
output i=9
cat@lubancat:~/test/make/test2$
```

基本の hello.c のコンパイルコマンドに比べて、ここでは入力ファイルの数を増やした点が主な違いです。例えば、「hello_main.c」、「hello_func.c」があります。さらに、新たに追加された「-I .」はコンパイラにヘッダーファイルのパスを伝え、コンパイル時に「.」（現在のディレクトリ）でヘッダーファイルを探せるようにするものです。実際には、「-I .」オプションを追加しなくても正常にコンパイルできますが、ここでは後で Makefile の関連変数を示すためにのみ追加されています。

16.2.2 Makefile を使用したコンパイル

gcc のコンパイルコマンドを Makefile に書式に従って記述するだけで、make を使用してコンパイルできることが想像できます。そのため、毎回 gcc コンパイルコマンドを手動で直接打つ必要はありません。操作は以下の通り、hello_main.c が存在するディレクトリでエディタを使用して「Makefile」という名前のファイルを新規作成し、以下の内容を入力して保存します。

株式会社日昇テクノロジー

```
1 #Makefile の形式

2 #ターゲット: 依存関係

3 #タブ コマンド 1

4 #タブ コマンド 2

5 #デフォルトターゲット

6 #hello_main は hello_main.c と hello_func.c に依存しています

7 hello_main: hello_main.c hello_func.c

8 gcc -o hello_main hello_main.c hello_func.c -I .

9

10

11 #clean ターゲットは、コンパイルで生成されたファイルを削除するために使用されます

12 clean:

13 rm -f *.o hello_main
```

このファイルは、プログラムをコンパイルするためのデフォルト目標 hello_main と、コンパイルで生成されたファイルを削除するための clean 目標を定義しています。特に、hello_main 目標名は gcc でコンパイルされたファイル名「gcc -o hello_main」と一致しています。つまり、この Makefile においては、hello_main は既に目標ファイル hello_main と見なされています。

これの利点は、make が実行されるたびに、hello_main ファイルと依存ファイル hello_main.c、hello_func.c の修正日をチェックし、依存ファイルの修正日が hello_main ファイルの日付より新しい場合、make は目標の下にある Shell コマンドを実行して hello_main ファイルを更新します。それ以外の場合は実行されません。

以下のコマンドを実行して実験してください：

株式会社日昇テクノロジー

```
1 # Makefile があるディレクトリで以下のコマンドを実行
2 # 以前に hello_main プログラムをコンパイルしていた場合、先に削除
3
4 rm hello_main
5 ls
6
7 # Makefile に従ってプログラムをコンパイルするために make を使用
8 make
9 ls
10
11 # 生成された hello_main プログラムを実行
12 ./hello_main
13
14 # 再び make を実行すると、hello_main ファイルは最新であると表示
15 make
16
17 # touch コマンドで hello_func.c のタイムスタンプを更新
18 touch hello_func.c
19
20 # hello_func.c が hello_main より新しいため、再コンパイルが実行される
21 make
22 ls
```

株式会社日昇テクノロジー

```
cat@lubancat:~/test/make/test2$ rm hello_main
cat@lubancat:~/test/make/test2$ ls
Makefile  hello_func.c  hello_func.h  hello_main.c
cat@lubancat:~/test/make/test2$ make
gcc -o hello_main hello_main.c hello_func.c -I .
cat@lubancat:~/test/make/test2$ ls
Makefile  hello_func.c  hello_func.h  hello_main  hello_main.c
cat@lubancat:~/test/make/test2$ ./hello_main
hello, world! This is a C program.
output i=0
output i=1
output i=2
output i=3
output i=4
output i=5
output i=6
output i=7
output i=8
output i=9
cat@lubancat:~/test/make/test2$ make
make: 'hello_main' is up to date.
cat@lubancat:~/test/make/test2$ touch hello_func.c
cat@lubancat:~/test/make/test2$ make
gcc -o hello_main hello_main.c hello_func.c -I .
cat@lubancat:~/test/make/test2$ ls
Makefile  hello_func.c  hello_func.h  hello_main  hello_main.c
cat@lubancat:~/test/make/test2$
```

上記の説明の通り、Makefile があれば、make コマンドを実行するだけで、全コンパイルプロセスを完了できます。

また、make は目標ファイルと依存ファイルの更新をチェックし、依存ファイルに変更がある場合にのみ、コマンドを再実行して目標ファイルを更新します。

16.3 ターゲットと依存関係

次に、Makefile の目標に関連する文法を再度まとめてみましょう：

[目標 1]:[依存関係]

[コマンド 1]

[コマンド 2]

[目標 2]:[依存関係]

[コマンド 1]

[コマンド 2]

- 目標：make が行うべきことを指し、簡単な識別子や目標ファイルであることができます。空白や Tab で始めることはできません。Makefile には複数の目標があり、最初に記述された目標が「デフォ

株式会社日昇テクノロジー

ルト目標」として設定されます。例えば、先に述べた targeta や hello_main がそれに該当します。

- 依存関係：目標を達成するために必要なファイルや他の目標。例えば、targeta は targetb と targetc に依存しています。また、コンパイルの例では、hello_main は hello_main.c と hello_func.c のソースファイルに依存しており、これらのファイルが更新された場合に再コンパイルされます。

- コマンド 1、コマンド 2...コマンド n：目標を達成するために必要なコマンド。目標ファイルが存在しないか、依存ファイルの修正日時が目標ファイルより新しい場合にのみ実行されます。コマンドの先頭は「Tab」キーを使用する必要があり、空白で置き換えることはできません。一部のエディタでは Tab が自動的に空白に変換されることがあり、これが原因でエラーが発生する場合があるので、エディタの設定を確認してください。

16.4 偽目標

Makefile で以前に記述した目標は、make にとって実際には目標ファイルです。例えば、make が実行される際に、ディレクトリ内に targeta ファイルが見つからないため、make targeta が実行されるたびに targeta のコマンドが実行され、targeta という名前のファイルが生成されることを期待しています。ディレクトリ下に実際に targeta、targetb、targetc のファイルが存在し、それらのファイルが最新である場合、make targeta は正常に実行されないことがあります。これを避けるために、Makefile では「.PHONY」プレフィックスを使用して目標識別子と目標ファイルを区別し、「偽目標」と呼ばれます。つまり、目標ファイルの生成を期待しない場合は、それを偽目標として定義すべきです。以下はその例です。

リスト 6: test1 の Makefile の修正

```
1 # .PHONY を使用して targeta を偽目標として宣言
2 .PHONY:targeta
3 # 目標 a は targetc と targetb に依存
4 # 実行する shell コマンド ls -lh でディレクトリ内の内容をリスト表示
```

株式会社日昇テクノロジー

```
5 targeta: targetc targetb
6 ls -lh
7
8 # .PHONY を使用して targetb を偽目標として宣言
9 .PHONY:targetb
10 # 目標 b は依存しない
11 # 実行する shell コマンドで test.txt ファイルを作成
する
12 targetb:
13 touch test.txt
14
15 # .PHONY を使用して targetc を偽目標として宣言
16 .PHONY:targetc
17 # 目標 c は依存しない
18 # 実行する shell コマンドで現在のパスを表示する
19 targetc:
20 pwd
21
22 # .PHONY を使用して targetd を偽目標として宣言
23 .PHONY:targetd
24 # 目標 d は依存しない
```

株式会社日昇テクノロジー

```
25 # abc の目標は targetd に依存していないため、直接 make を実行しても targetd は実行されない
26 # make targetd コマンドで実行可能
27 targetd:
28 rm -f test.txt
```

test2 の Makefile に.PHONY を追加する

リスト 7: base_linux/makefile/test3/Makefile

```
1 # デフォルトターゲット
2 #hello_main は hello_main.c と hello_func.c に依存しています
3 hello_main: hello_main.c hello_func.c
4 gcc -o hello_main hello_main.c hello_func.c -I .
5 #clean 疑似ターゲットは、コンパイルで生成されたファイルを削除するために使用されます
6 .PHONY: clean
7 clean:
8 rm -f *.o hello_main
```

GNU 組織がリリースするソフトウェアエンジニアリングのコードの Makefile には、上記のコードで定義された clean などの偽目標がよくあります。これは、コンパイルの出力ファイルをクリアするために使用されます。他にも「all」、「install」、「print」、「tar」などがあり、それぞれすべてをコンパイルする、コンパイル済みのプログラムをインストールする、変更されたファイルをリストアップする、tar ファイルにパッケージするために使用されます。これらの名前を使用する必要はありませんが、これらの慣習に従って自分の Makefile を記述することができます。

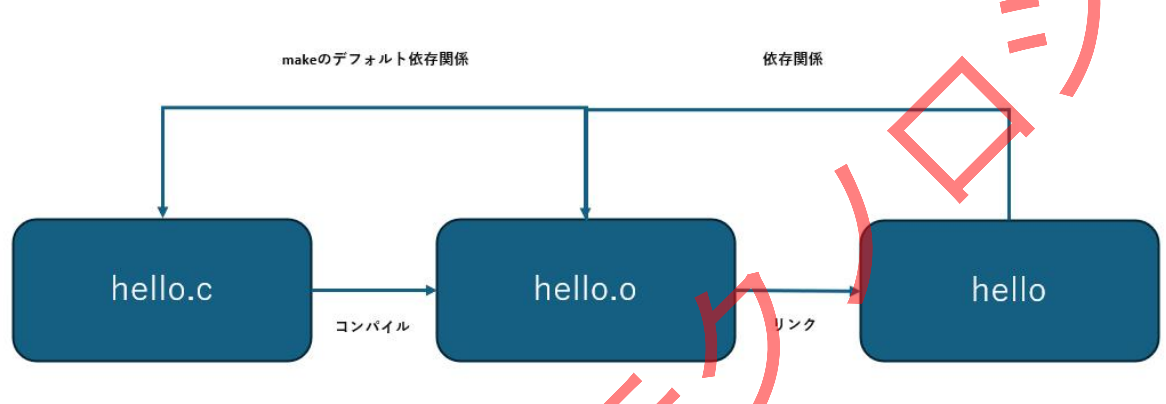
このコードに「.PHONY:clean」という文を書かずに、ディレクトリに「clean」という名前のファイルを作成した場合、「make clean」を実行しても clean のコマンドは実行されないことを、興味がある方は実

株式会社日昇テクノロジー

際に試してみてください。

16.5 デフォルトルール

前章「GCC コンパイルプロセス」で述べたコンパイルプロセスには以下のステップが含まれており、make も同じ流れで実行されます。しかし、Makefile の実際の使用では、コンパイルと最終リンクプロセスを分けることが一般的です。



つまり、hello_main ターゲットファイルは本質的には hello_main.c と hello_func.c ファイルに依存しているわけではなく、hello_main.o と hello_func.o に依存しており、これら二つのファイルをリンクすることで最終的に求める hello_main ターゲットファイルが得られます。さらに、make にはデフォルトルールがあり、xxx.o ファイルが見つからない場合には、ディレクトリ内の同名の xxx.c ファイルを探してコンパイルします。このルールに基づいて、Makefile を以下のように変更できます。

リスト 8: base_linux/makefile/test4/Makefile

```
1 #Makefile 形式
2 # ターゲットファイル: 依存するファイル
3 #Tab コマンド 1
4 #Tab コマンド 2
5 hello_main: hello_main.o hello_func.o
```

株式会社日昇テクノロジー

```
6 gcc -o hello_main hello_main.o hello_func.o
7 # 以下は make のデフォルトルールで、次の 2 行は書かなくても良い
8 #hello_main.o: hello_main.c
9 # gcc -c hello_main.c
10
11 # 以下は make のデフォルトルールで、次の 2 行は書かなくても良い
12 #hello_func.o: hello_func.c
13 # gcc -c hello_func.c
```

上記のコードの 5～6 行目では、依存ファイルを C ファイルから.o ファイルに変更し、gcc コンパイルコマンドもそれに応じて変更されています。8～13 行目は、hello_main.o ファイルと hello_func.o ファイルの依存関係とコンパイルコマンドですが、C を同名の.o ファイルにコンパイルするのが make のデフォルトルールであるため、この部分の内容は通常書かれません。

修正後の Makefile を使用したコンパイル結果は下図の通りです。

```
cat@lubancat:~/test/make/test3$ ls
Makefile  hello_func.c  hello_func.h  hello_main.c
cat@lubancat:~/test/make/test3$ make
cc    -c -o hello_main.o hello_main.c
cc    -c -o hello_func.o hello_func.c
gcc -o hello_main hello_main.o hello_func.o
cat@lubancat:~/test/make/test3$ ls
Makefile  hello_func.c  hello_func.h  hello_func.o  hello_main  hello_main.c  hello_main.o
cat@lubancat:~/test/make/test3$ ./hello_main
hello, world! This is a C program.
output i=0
output i=1
output i=2
output i=3
output i=4
output i=5
output i=6
output i=7
output i=8
output i=9
cat@lubancat:~/test/make/test3$
```

変更後の Makefile を使用したコンパイル結果から、make が追加の「cc」コンパイルコマンドを 2 つ実行したことがわかります。これは make のデフォルトルールによるもので、C コードを同名の.o ファイル

株式会社日昇テクノロジー

にコンパイルし、その後 Makefile のコマンドに従ってこれら二つのファイルをリンクして最終的なターゲットファイル hello_main を生成します。

16.6 変数の使用

C を自動的に*.o にコンパイルするデフォルトルールには欠点があり、*.o が.h ヘッダーファイルに依存していることを明示的に示していないため、ヘッダーファイルの内容を変更しても*.o が更新されないことがあります。これは受け入れがたいです。さらに、デフォルトルールでは固定の「cc」を使用してコンパイルするため、ARM-GCC を使用してクロスコンパイルしたい場合、システムのデフォルトの「cc」ではコンパイルエラーが発生します。

これらの問題を解決し、Makefile をより汎用的にするためには、変数と分岐を導入して処理する必要があります。

16.6.1 基本構文

Makefile 内の変数は、C 言語のマクロ定義のようなもので、変数を使用する場所で変数値に置き換えられます。変数の命名には文字、数字、アンダースコアを含めることができ、大文字と小文字は区別されます。変数を定義する方法には以下の四つがあります：

- 「=」：遅延割り当て、この変数は呼び出された時にのみ値が割り当てられます
- 「:=」：直接割り当て、遅延割り当てとは逆に、この方法を使用すると、変数の値は定義時にすでに決定されています。
- 「?=」：変数の値が空の場合に割り当てを行います、通常はデフォルト値を設定するのに使用されます。
- 「+=」：追加割り当て、変数の後ろに新しい内容を追加することができます。

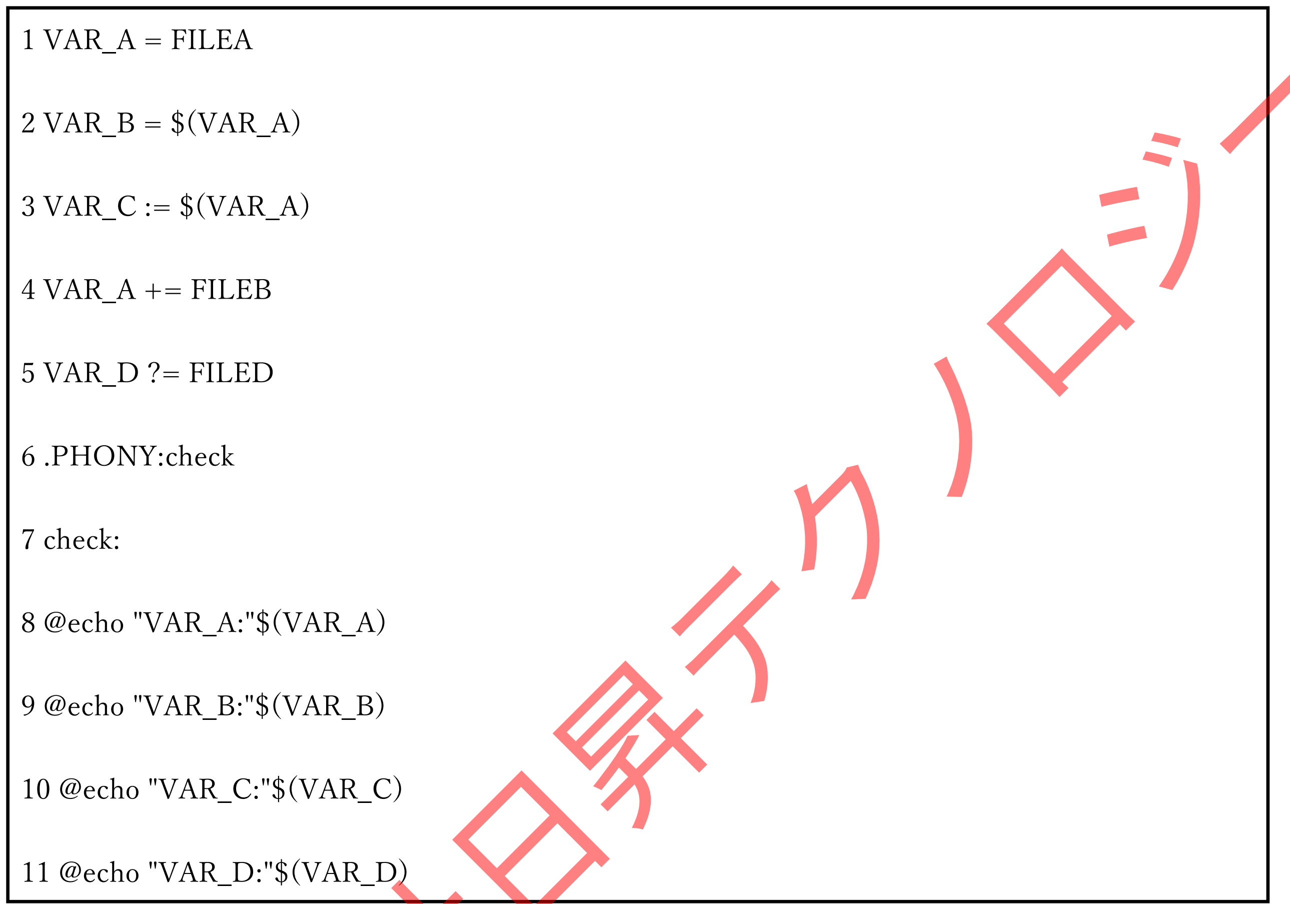
変数を使用したい場合の構文は以下の通りです：

```
1 $(変数名)
```

株式会社日昇テクノロジー

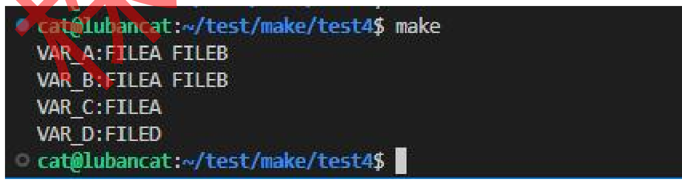
これらの四つの定義方法について、特に遅延割り当てと直接割り当ての違いを考えながら、以下の実験コードを通じて説明します。

リスト 9: base_linux/makefile/test5/Makefile



```
1 VAR_A = FILEA
2 VAR_B = $(VAR_A)
3 VAR_C := $(VAR_A)
4 VAR_A += FILEB
5 VAR_D ?= FILED
6 .PHONY:check
7 check:
8 @echo "VAR_A:"$(VAR_A)
9 @echo "VAR_B:"$(VAR_B)
10 @echo "VAR_C:"$(VAR_C)
11 @echo "VAR_D:"$(VAR_D)
```

ここでは VAR_B と VAR_C の代入方法に主に注目します。実験結果は下図のようになります。make コマンドを実行した後、VAR_C のみが FILEA である。これは、VAR_B が遅延代入を使用しており、呼び出された時にのみ代入が行われるためです。VAR_B を呼び出す時には、VAR_A の値が既に FILEA FILEB に変更されているため、VAR_B の変数値も FILEA FILEB と等しくなります。



16.6.2 デフォルトルールの改造

次に、以前の hello_main の Makefile を変数を使用して大きく改造します。以下のように示します。

株式会社日昇テクノロジー

リスト 10: base_linux/makefile/test6/Makefile

```
1 # 変数の定義
2 CC=gcc
3 CFLAGS=-I.
4 DEPS = hello_func.h
5
6 # ターゲットファイル
7 hello_main: hello_main.o hello_func.o
8 $(CC) -o hello_main hello_main.o hello_func.o
9
10 # *.o ファイルの生成ルール
11 %.o: %.c $(DEPS)
12 $(CC) -c -o $@ $< $(CFLAGS)
13
14 # 疑似ターゲット
15 .PHONY: clean
16 clean:
17 rm -f *.o hello_main
```

株式会社日昇テクノロジー

• コードの 1～4 行目：CC、CFLAGS、DEPS 変数をそれぞれ定義し、定義された変数は「$(変数名)」の形式で参照できます。例えば、後の「$(CC)」、「$(CFLAGS)」、「$(DEPS)」は、定義時に割り当てられた変数値「gcc」、「-I.」、「hello_func.h」と等価です。

• コードの 8 行目：$(CC)を gcc に代えて使用し、このように Makefile を記述することで、異なるコンパイラを簡単に切り替えることができます。例えば、クロスコンパイルを行う場合、冒頭のコン

パイラ名を変更するだけで済みます。

• コードの 11 行目：「%」はワイルドカードで、「*」に似た機能を持ちます。例えば、「%.o」とは、「.o」で終わるすべてのファイルを意味します。したがって、「%.o:%.c」はこの例では「hello_main.o: hello_main.c」、「hello_func.o: hello_func.c」と等価で、o ファイルが c ファイルに依存するデフォルトルールと等価です。しかし、この行の後の「$(DEPS)」は、c ファイルに加えて、変数「$(DEPS)」がリストすヘッダーファイルにも依存することを意味します。したがって、ヘッダーファイルが変更された場合、o ファイルも再コンパイルされます。

• コードの 12 行目：特別な変数「$@」、「$<」が登場します。これらは Makefile で予約されたキーワードであり、システム予約の自動変数です。「$@」はターゲットファイルを、「$<」は最初の依存ファイルをリストします。つまり、「$@」は「%.o」を、「$<」は「%.c」をリストし、11 行目の「%」が「hello_func」と一致した場合、12 行目のコードは以下と等価です：

```
1 # "%"が"hello_func"に一致した場合：
2 $(CC) -c -o $@ $< $(CFLAGS)
3 # 以下と等価：
4 gcc -c -o hello_func.o hello_func.c -I.
```

つまり、Makefile は変数と自動変数を利用して、.o ファイルのデフォルト生成ルールを再記述し、ヘッダーファイルへの依存性を追加できます。

16.6.3 リンクルールの改造

*.o ファイルのデフォルトルールと同様に、最終ターゲットファイルの生成ルールも変数を使用して修正できます。具体的には以下のコードを参照してください。

株式会社日昇テクノロジー

リスト 11: base_linux/makefile/test7/Makefile

```
1 # 変数の定義
2 TARGET = hello_main
3 CC = gcc
4 CFLAGS = -I.
5 DEPS = hello_func.h
6 OBJS = hello_main.o hello_func.o
7
8 # ターゲットファイル
9 $(TARGET): $(OBJS)
10 $(CC) -o $@ $^ $(CFLAGS)
11
12 # *.o ファイルの生成ルール
13 %.o: %.c $(DEPS)
14 $(CC) -c -o $@ $< $(CFLAGS)
15
16 # 疑似ターゲット
17 .PHONY: clean
18 clean:
19 rm -f *.o $(TARGET)
```

この部分の説明は以下の通りです：

- コードの 2 行目：TARGET 変数を定義し、その値はターゲットファイル名 hello_main です。
- コードの 6 行目：OBJS 変数を定義し、その値は依存する各 o ファイル、例えば hello_main.o、

株式会社日昇テクノロジー

hello_func.o ファイルです。

• コードの 9 行目：TARGET と OBJS 変数を使用して、元々固定された内容を置き換えます。

• コードの 10 行目：自動変数「$@」を使用してターゲットファイル「$(TARGET)」をリストし、「$^」を使用してすべての依存ファイル、即ち「$(OBJS)」をリストします。

つまり、上記のコードによる Makefile は、コンパイルおよびリンクのプロセスを変数でリスト現しており、非常に汎用的です。このような Makefile を使用すると、異なるプロジェクトに対して変数の内容を直接変更するだけで使用できます。

16.6.4 その他の自動変数

Makefile には他にも自動変数がありますが、ここでは将来の参照のために一部を列挙します。以下のリストを参照してください。

リスト自動変数

記号	意味
$@	ターゲットファイルにマッチします
$%	$@ に似ていますが、$% は「ライブラリ」タイプのターゲットファイルにのみマッチします
$<	依存関係の中の最初のターゲットファイルです
$^	全ての依存ターゲットで、依存関係に重複がある場合は一つだけを保持します
$+	全ての依存ターゲットで、依存関係に重複があってもそのまま保持します
$?	ターゲットより新しい全ての依存ターゲットです

16.7 関数の使用

より複雑なプロジェクトでは、ヘッダーファイルやソースファイルがサブディレクトリに配置されることがあります。また、*.o ファイルや実行可能ファイルを整理しやすいように専用のビルド出力ディレクトリに配置することもあります。下図に示すように、例では*.h ヘッダーファイルを includes ディレクトリに、*.c ファイルを sources ディレクトリに配置し、ビルド出力は build ディレクトリに保存されます。

これらの複雑な操作を実現するためには、通常、Makefile の関数を使用する必要があります。

株式会社日昇テクノロジー

```
cat@lubancat:~/test/make/test5$ tree
.
|-- Makefile
|-- build
|   |-- hello_func.o
|   |-- hello_main
|   `-- hello_main.o
|-- includes
|   `-- hello_func.h
`-- sources
    |-- hello_func.c
    `-- hello_main.c

3 directories, 7 files
cat@lubancat:~/test/make/test5$
```

16.7.1 関数の形式と例

Makefile 内で関数を呼び出す方法は、変数の使用と似ており、「$()」または「${}」記号で関数名と引数を囲みます。具体的な文法は以下の通りです：

```
1 $(関数名 引数)
2 # または波括弧を使用
3 ${関数名 引数}
```

ここでは、よく使用される notdir、patsubst、wildcard 関数を例に説明します。これらの例は後に Makefile で使用されます。

16.7.1.1 notdir 関数

notdir 関数は、ファイルパスからディレクトリ部分を取り除くために使用されます。その形式は以下の通りです：

```
1 $(notdir ファイル名)
```

例えば、引数に「./sources/hello_func.c」を入力すると、関数の実行結果は「hello_func.c」になります。つまり、入力から「./sources/」パス部分を取り除き、ファイル名のみを保持します。使用例は以下の通りです：

株式会社日昇テクノロジー

```
1 # 以下は使用例
2 $(notdir ./sources/hello_func.c)
# 上記の関数を実行すると、「./sources/」部分が取り除かれ、「hello_func.c」と出力されます。
```

16.7.1.2 wildcard 関数

wildcard 関数は、ファイルリストを取得し、空白で区切って返すために使用されます。その形式は以下の通りです：

$(wildcard マッチングパターン)

例えば、関数呼び出し$(wildcard *.c)を実行すると、現在のディレクトリの全ての c ファイルがリストアップされます。Makefile ディレクトリでこの関数を実行した場合の使用例は以下の通りです：

```
# sources ディレクトリには hello_func.c、hello_main.c、test.c ファイルが存在
# 次の関数を実行
$(wildcard sources/*.c)
# 関数の出力は以下の通り：
sources/hello_func.c sources/hello_main.c sources/test.c
```

16.7.1.3 patsubst 関数

patsubst 関数は、パターン文字列の置換を行います。その形式は以下の通りです：

```
1 $(patsubst マッチングパターン, 置換ルール, 入力文字列)
```

入力文字列がマッチングパターンに一致する場合、その文字列は置換ルールに従って置換されます。マッチングパターンに「%」が含まれる場合、置換ルールでも「%」を使用して「%」にマッチした内容を最終的な置換文字列に取り込むことができます。具体例を以下に示します：

次の関数を実行

株式会社日昇テクノロジー

```
1 $(patsubst %.c, build_dir/%.o, hello_main.c )
2 # 関数の出力は
3 build_dir/hello_main.o
4 # 次の関数を実行
5 $(patsubst %.c, build_dir/%.o, hello_main.xxx )
6 # hello_main.xxx はマッチングパターン「%.c」に一致しないため、関数に出力はありません
```

最初の関数呼び出しでは、「hello_main.c」が「%.c」のマッチングパターンに一致し（%は Makefile でのワイルドカードに似ています）、そして「%」は「hello_main.c」から「hello_main」という文字を抽出し、置換ルール「build_dir/%.o」の「%」に挿入したため、最終出力は「build_dir/hello_main.o」になります。

二番目の関数呼び出しでは、「hello_main.xxx」が「%.c」のマッチングパターンに一致しないため（「.xxx」と「.c」が一致しないため）、置換は行われず、関数は空の内容を返します。

16.7.2 多層構造プロジェクトの Makefile

次に、上記の 3 つの関数を使用して、複数のディレクトリを含むプロジェクトに適した Makefile を修正します。修正後の内容は以下の通りです。

リスト 12: base_linux/makefile/test8/Makefile

```
1 # 変数の定義
2 TARGET = hello_main
3 # 中間ファイルを保存するパス
4 BUILD_DIR = build
5 # ソースファイルを保存するフォルダ
6 SRC_DIR = sources
```

株式会社日昇テクノロジー

7 # ヘッダーファイルを保存するフォルダ

```
8 INC_DIR = includes

9 # ソースファイル

10 SRCS = $(wildcard $(SRC_DIR)/*.c)

11 # 目的ファイル（*.o）

12 OBJS = $(patsubst %.c, $(BUILD_DIR)/%.o, $(notdir $(SRCS)))

13 # ヘッダーファイル

14 DEPS = $(wildcard $(INC_DIR)/*.h)

15 # ヘッダーファイルのパスを指定

16 CFLAGS = $(patsubst %, -I%, $(INC_DIR))

17

18 # 目的ファイル

19 $(BUILD_DIR)/$(TARGET): $(OBJS)

20 $(CC) -o $@ $^ $(CFLAGS)

21

22 # *.o ファイルの生成ルール

23 $(BUILD_DIR)/%.o: $(SRC_DIR)/%.c $(DEPS)

24

25 # コンパイルディレクトリを作成し、途中ファイルを保存

26 # コマンドの前に「@」がある場合、ターミナルに出力されない

27 @mkdir -p $(BUILD_DIR)

28 $(CC) -c -o $@ $< $(CFLAGS)

29
```

株式会社日昇テクノロジー

```
30 # 疑似ターゲット
31 .PHONY: clean cleanall
32
33 # 出力フォルダを削除
34 clean:
35 rm -rf $(BUILD_DIR)
36
37 # 全て削除
38 cleanall:
39 rm -rf $(BUILD_DIR)
```

この Makefile は、先に示したプロジェクト構造と連携している必要があります。そうでなければ、Makefile が正しく記述されていても、コンパイルエラーが発生します。具体的には、サンプルコード「step5」の内容を直接参照できます。修正後の Makefile の分析は以下の通りです：

- コードの 4～8 行：変数 BUILD_DIR、SRC_DIR、INC_DIR をそれぞれ、プロジェクトのビルド出力パス build、ソースファイルパス sources、ヘッダーファイルパス includes および現在のディレクトリ「.」として定義しています。
- コードの 10 行目：変数 SRCS を定義し、コンパイルする必要がある全てのソースファイルを格納しています。その値は wildcard 関数の出力で、この例では「sources/hello_func.c sources/hello_main.c sources/test.c」となります。
- コードの 12 行目：変数 OBJS を定義し、生成される全ての.o ファイルを格納しています。その値は patsubst 関数の出力で、この例では全ての c ファイル名を同名の.o ファイルに置き換え、build ディレクトリを追加しています。つまり、関数の出力は「build/hello_func.o build/hello_main.o

株式会社日昇テクノロジー

build/test.o」となります。

• コードの 14 行目：SRCS 変数と同様に、依存する全てのヘッダーファイルを格納する DEPS 変数を定義しています。その値は wildcard 関数の出力で、この例では「includes/hello_func.h」となります。

• コードの 16 行目：CFLAGS 変数を定義し、インクルードするヘッダーファイルのパスを格納しています。その値は patsubst 関数の出力で、この例では includes ディレクトリを「-I」の後に追加しています。関数の出力は「-Iincludes」となります。

• コードの 19 行目：以前の Makefile に比べて、$(TARGET)の前に$(BUILD_DIR)パスを追加し、最終的な実行可能プログラムを build ディレクトリに配置しています。

- 23 行目のコード：上記のように、.o オブジェクトファイルに $(BUILD_DIR) パスを追加します。

- 27 行目のコード：コンパイルを実行する前に build ディレクトリを作成し、後で .o ファイルを保存します。コマンドの前にある「@」は、このコマンドを実行する際に端末に出力しないことを意味します。

- 34 行目のコード：rm 削除コマンドも、コンパイルディレクトリ $(BUILD_DIR) を直接削除するように変更されました。

- 38～39 行目のコード：すべてのアーキテクチャコンパイルディレクトリを削除する疑似ターゲット cleanall を追加しました。

この Makefile を使用するには、Makefile のディレクトリで直接 make を実行します。

```
1 # ディレクトリ構造を確認するために tree コマンドを使用
2 # コマンドが見つからない場合は、sudo apt install tree でインストール
3 tree
4
5 # コンパイル
6 make
```

株式会社日昇テクノロジー

以下のように表示されます:

```
cat@lubancat:~/test/make/test5$ tree
.
|-- Makefile
|-- includes
|   `-- hello_func.h
`-- sources
    |-- hello_func.c
    `-- hello_main.c

2 directories, 4 files
cat@lubancat:~/test/make/test5$ make
cc -c -o build/hello_func.o sources/hello_func.c  -Iincludes  -I.
cc -c -o build/hello_main.o sources/hello_main.c  -Iincludes  -I.
cc -o build/hello_main build/hello_func.o build/hello_main.o  -Iincludes  -I.
cat@lubancat:~/test/make/test5$ tree
.
|-- Makefile
|-- build
|   |-- hello_func.o
|   |-- hello_main
|   `-- hello_main.o
|-- includes
|   `-- hello_func.h
`-- sources
    |-- hello_func.c
    `-- hello_main.c

3 directories, 7 files
cat@lubancat:~/test/make/test5$
```

16.8 ブランチの使用

PC に Linux がない場合は、このセクションをスキップするか、ブランチの構文について学びます。

GCC コンパイラを直接切り替えることができるように、条件分岐を使用してコンパイラの切り替え機能を追加することもできます。Makefile 内の条件分岐の構文は以下の通りです：

```
1 ifeq(arg1, arg2)
2 ブランチ 1
3 else
4 ブランチ 2
5 endif
```

括弧内の引数「arg1」と「arg2」の値が同じ場合は真となり、ブランチ 1 の内容を実行します。そうでない場合は、ブランチ 2 の内容を実行します。arg1 と arg2 は変数でも定数でもかまいません。

GCC コンパイラを切り替える Makefile の使用例は以下の通りです。(PC で使用)

株式会社日昇テクノロジー

リスト 13: base_linux/makefile/test9/Makefile

```
1 # 変数の定義
2 #ARCH はデフォルトで x86、gcc コンパイラを使用、
3 # そうでなければ arm コンパイラを使用
4 ARCH ?= x86
5 TARGET = hello_main
6
7 # 中間ファイルのパス
8 BUILD_DIR = build_$(ARCH)
9 # ソースファイルのフォルダ
10 SRC_DIR = sources
11 # ヘッダーファイルのフォルダ
12 INC_DIR = includes .
13
14 # ソースファイル
15 SRCS = $(wildcard $(SRC_DIR)/*.c)
# オブジェクトファイル（*.o）
17 OBJS = $(patsubst %.c, $(BUILD_DIR)/%.o, $(notdir $(SRCS)))
18 # ヘッダーファイル
19 DEPS = $(wildcard $(INC_DIR)/*.h)
20
21 # ヘッダーファイルのパスを指定
```

株式会社日昇テクノロジー

22 CFLAGS = $(patsubst %, -I%, $(INC_DIR))

```
23
24 # 入力された ARCH 変数に基づいてコンパイラを選択
25 #ARCH=x86 の場合は gcc を使用
26 #ARCH=arm の場合は arm-gcc を使用
27 ifeq ($(ARCH),x86)
28 CC = gcc
29 else
30 CC = arm-linux-gnueabihf-gcc
31 endif
32
33 # オブジェクトファイル
34 $(BUILD_DIR)/$(TARGET): $(OBJS)
35 $(CC) -o $@ $^ $(CFLAGS)
36
37 #*.o ファイルの生成規則
38 $(BUILD_DIR)/%.o: $(SRC_DIR)/%.c $(DEPS)
39 # 過程ファイルを格納するためのコンパイルディレクトリを作成
40 # コマンドの前に「@」を付けると、端末に出力されない
41 @mkdir -p $(BUILD_DIR)
42 $(CC) -c -o $@ $< $(CFLAGS)
43
44 # 疑似ターゲット
```

株式会社日昇テクノロジー

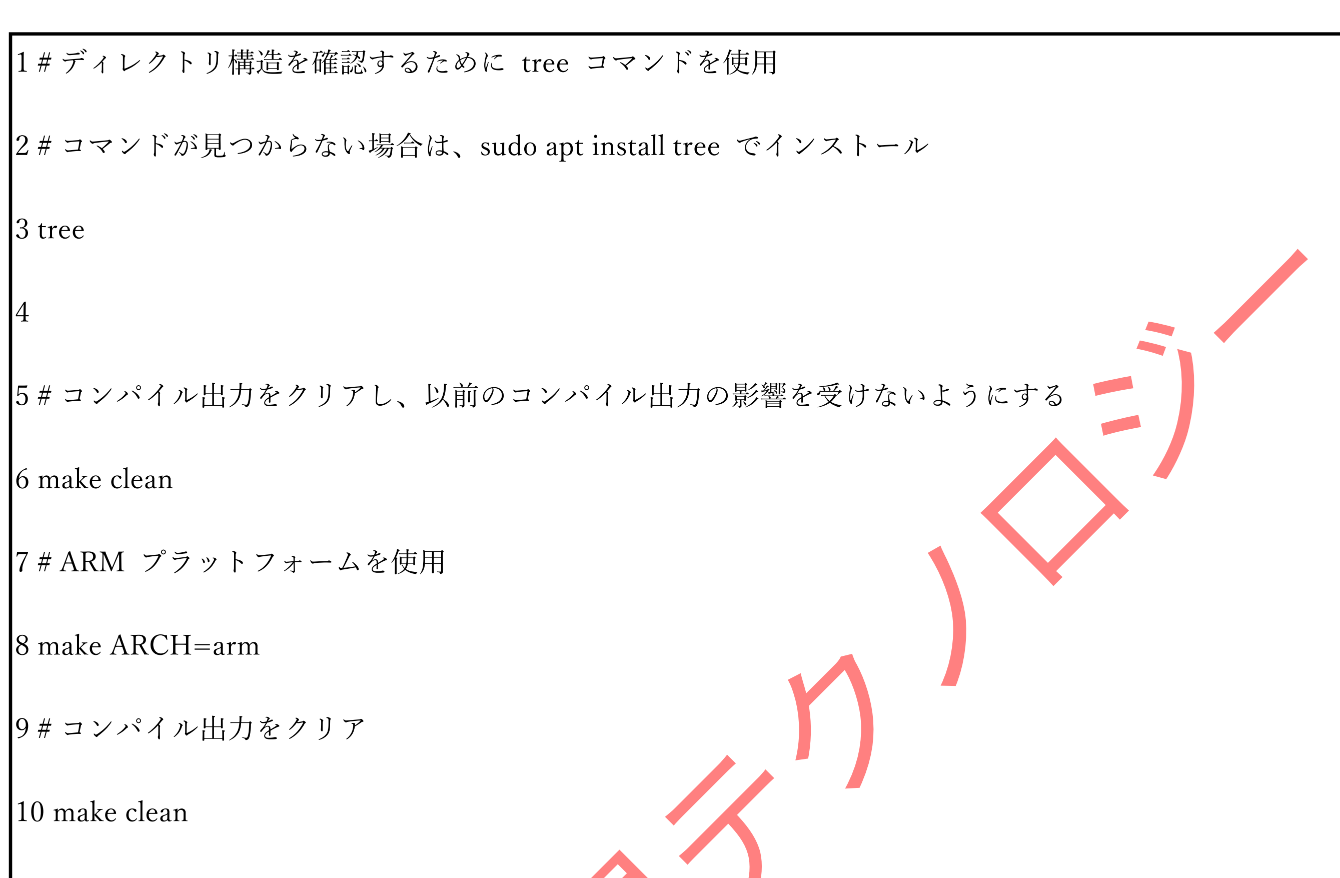
```
45 .PHONY: clean cleanall

46 # アーキテクチャに応じて削除

47 clean:

rm -rf $(BUILD_DIR)

49

50 # すべて削除

51 cleanall:

52 rm -rf build_x86 build_arm
```

この Makefile ファイルは、前述のプロジェクト構造と一緒に使用する必要があります。そうでないと、Makefile が正しく書かれていても、ディレクトリが合わないためにコンパイルエラーが発生します。具体的な内容は、サンプルコード「step5」で直接確認できます。変更後の Makefile ファイルの分析は以下の通りです：

- コードの 1～4 行目：ARCH の割り当てが追加されました。ARCH の値が空の場合は、X86 に割り当てられます。
- コードの 9 行目：異なるコンパイル方法の出力結果のフォルダが追加されました。
- コードの 25～32 行目：gcc コンパイラの選択が追加されました。
- コードの 48 行目：異なるコンパイラの出力を選択して削除します。
- コードの 48 行目：すべての出力結果を削除します。

この Makefile を使用するには、Makefile のディレクトリで直接 make を実行します。

株式会社日昇テクノロジー



```
1 # ディレクトリ構造を確認するために tree コマンドを使用
2 # コマンドが見つからない場合は、sudo apt install tree でインストール
3 tree
4
5 # コンパイル出力をクリアし、以前のコンパイル出力の影響を受けないようにする
6 make clean
7 # ARM プラットフォームを使用
8 make ARCH=arm
9 # コンパイル出力をクリア
10 make clean
11 # デフォルトは x86 プラットフォーム
12 make
```

以下のように表示されます:

株式会社日昇テクノロジー

```
embedfire@dev:~/workdir/example/base_code/makefile_step/step5$ tree
.
├── includes
│   └── hello_func.h
├── Makefile
└── sources
    ├── hello_func.c
    ├── hello_main.c
    └── test.c

2 directories, 5 files
embedfire@dev:~/workdir/example/base_code/makefile_step/step5$ make
gcc -c -o build_x86/hello_func.o sources/hello_func.c  -Iincludes
gcc -c -o build_x86/hello_main.o sources/hello_main.c  -Iincludes
gcc -c -o build_x86/test.o sources/test.c  -Iincludes
gcc -o build_x86/hello_main build_x86/hello_func.o build_x86/hello_main.o build_x86/test.o  -Iin
cludes
embedfire@dev:~/workdir/example/base_code/makefile_step/step5$ make ARCH=arm
arm-linux-gnueabihf-gcc -c -o build_arm/hello_func.o sources/hello_func.c  -Iincludes
arm-linux-gnueabihf-gcc -c -o build_arm/hello_main.o sources/hello_main.c  -Iincludes
arm-linux-gnueabihf-gcc -c -o build_arm/test.o sources/test.c  -Iincludes
arm-linux-gnueabihf-gcc -o build_arm/hello_main build_arm/hello_func.o build_arm/hello_main.o bu
ild_arm/test.o  -Iincludes
embedfire@dev:~/workdir/example/base_code/makefile_step/step5$ tree
.
├── build_arm
│   ├── hello_func.o
│   ├── hello_main
│   ├── hello_main.o
│   └── test.o
├── build_x86
│   ├── hello_func.o
│   ├── hello_main
│   ├── hello_main.o
│   └── test.o
├── includes
│   └── hello_func.h
├── Makefile
└── sources
    ├── hello_func.c
    ├── hello_main.c
    └── test.c

4 directories, 13 files
```

この例の Makefile は現在、単一のソースファイルディレクトリのみをサポートしています。複数のソースファイルディレクトリがある場合は、さらなる改善が必要です。これらについては、今後の学習で続けて蓄積していきます。

第 17 章 ファイル操作とシステムコール

Linux システムでは、「すべてはファイル」という重要な概念があります。これにより、すべてのリソースがファイルとみなされます。これにはハードウェアデバイスも含まれ、通常はデバイスファイルと呼ばれます。以前、スクリプトを使用してファイルの読み書きを行い、ハードウェアにアクセスしたことがあ

株式会社日昇テクノロジー

ります。したがって、Linux のファイル操作を理解していなければ、ボードの LED ライトを点灯させることさえ困難です！

この章では、C 標準ライブラリおよびシステムコールを使用してファイルを操作し、ユーザーアプリケーション、C 標準ライブラリ、およびシステムコール間の関係を明らかにします。

この章のサンプルコードディレクトリは、base_linux/file_io です。

17.1 ストレージデバイスファイルシステム

ファイルシステムについて話すとき、通常最初に思い浮かべるのは、Windows の FAT32、NTFS、exFAT、および Linux の一般的な ext2、ext3、ext4 などのタイプです。これらのファイルシステムはすべて、ストレージスペースを効率的に管理する問題を解決するために生まれました。

EEPROM、Nor FLASH、NAND FLASH、eMMC から機械式ハードディスクまで、さまざまなストレージデバイスは本質的には 0 と 1 のデータユニットを保存できる多数のユニットを持つデバイスです。データを保存する際には、これらのストレージユニットの物理アドレスに直接アクセスして内容を保存する必要があります。これは、有効なデータの位置を記録するのが難しく、ストレージメディアの残りのスペースを決定するのが難しく、データをどのような形式で解釈するべきかを決定するのが難しいなど、大きな不便をもたらします。これは、誰もが管理せずにさまざまな種類の本で無秩序に積み重ねられた巨大な図書館のようなものです。

データを効率的に保存および管理するために、ファイルシステムはストレージメディアに一定の組織構造を構築しました。これには、オペレーティングシステムのブートセクター、ディレクトリ、ファイルが含まれます。これは、図書館が異なる種類の本を分類し、番号を付けて異なる棚に置くようなものです。異なる管理理念により、FAT32、NTFS、exFAT、ext2/3/4 など、異なるタイプのファイルシステム標準が導かれました。それ以外にも、NAND タイプのデバイス向けのファイルシステム jffs2、yaffs2 などがあります。

ファイルシステムのおかげで、コンピュータ上のデータはユーザーにファイルの形で提示されます。ファ

株式会社日昇テクノロジー

イルシステムの基本的な概念については、MCU を使用して直接データを記録する場合とファイルシステムを介してファイルを記録する場合の違いを非常に明確に理解することができます。

以下は、さまざまな標準ファイルシステムの特徴についての簡単な説明です：

- FAT32 フォーマット：互換性が高く、STM32 などの MCU でも Fatfs を介して FAT32 ファイルシステムをサポートしています。多くの SD カードや USB メモリは出荷時に FAT32 ファイルシステムを使用しています。主な欠点は技術が古く、単一のファイルサイズが 4GB を超えることができない非ログ型ファイルシステムであることです。

- NTFS フォーマット：単一のファイル最大 256TB をサポートし、長いファイル名やサーバーファイル管理権限などをサポートしており、NTFS はログ型ファイルシステムです。しかし、ログ型ファイルシステムであるため、詳細な読み書き操作を記録し、相対的にフラッシュメモリの消耗を速める可能性があります。ファイルシステムのログ機能とは、システムに障害が発生した場合にデータの完全性を最大限に保証できるように、ファイルシステムの操作をディスクの特定のパーティションに記録することを指します。

- exFAT フォーマット：FAT32 から改良され、フラッシュメディアのストレージ（SD カード、USB メモリなど）用に設計されており、空間の無駄が少ない。単一のファイル最大 16EB をサポートし、非ログファイルシステムです。

- ext2 フォーマット：シンプルでファイルが少ない場合の性能が良好で、単一のファイルは 2TB を超えることができません。非ログファイルシステムです。

- ext3 フォーマット：ext2 に比べてログ機能のサポートが主な追加点です。

- ext4 フォーマット：ext3 から改良され、ext3 は実際には ext4 のサブセットです。1EB のパーティションをサポートし、単一のファイル最大 16TB をサポートし、無限のサブディレクトリ数をサポートし、遅延割り当て戦略を使用してファイルのデータブロック割り当てを最適化し、ログ機能の使用を自主的に制御できます。

株式会社日昇テクノロジー

- jffs2 および yaffs2 フォーマット：jffs2 と yaffs2 は、FLASH タイプのストレージ用に設計されたファイルシステムで、フラッシュメモリの特性に合わせてウェアレベリングと電源断保護などの特徴が追加されています。Nor、NAND FLASH タイプのストレージブロックの書き換え回数は限られています（通常は 10 万回）、これらのタイプのファイルシステムを使用することで、ストレージの消耗を減らすことができます。

要するに、Linux では、ext2 は USB メモリに適していますが（互換性のためによく使用されますが、FAT32 や exFAT が多く使用されています）、日常的な使用には ext4 を推奨し、ext3 を使用するシーンは ext4 の安定性に疑問を持つユーザーに限られるかもしれませんが、ext4 は 2008 年に実験期間を終え、安定版に入ったので、安心して使用できます。

Linux カーネル自体も FAT32 ファイルシステムをサポートしていますが、NTFS フォーマットを使用するには ntfs-3g などの追加ツールをインストールする必要があります。したがって、工場出荷時のデフォルト Linux システムを使用する場合、FAT32 フォーマットの USB メモリを直接挿入すると自動的にマウントされますが、NTFS フォーマットはサポートされていません。ホストの Ubuntu は、NTFS または FAT32 の USB メモリを自動的に認識してマウントできます。これは、Ubuntu のディストリビューションに対応するサポートがインストールされているためです。現在、マイクロソフトは exFAT ファイルシステムの標準を公開し、Linux にオープンソース化しています。将来的には Linux もデフォルトで exFAT をサポートする可能性があります。

FLASH メモリの消耗を非常に気にする場合は、jffs2 や yaffs2 などのファイルシステムの使用を検討することができます。

Linux では、システムの現在のストレージデバイスが使用しているファイルシステムを以下のコマンドで確認できます：

```
# ホストまたはボード上で以下のコマンドを実行します

df -T
```

株式会社日昇テクノロジー

以下の画像から、

```
cat@lubancat:~$ df -T
Filesystem     Type     1K-blocks    Used Available Use% Mounted on
/dev/root      ext4       7340392 3789384   3204896  55% /
devtmpfs       devtmpfs    996148       8    996140   1% /dev
tmpfs          tmpfs      1005108       0   1005108   0% /dev/shm
tmpfs          tmpfs      1005108  109308    895800  11% /run
tmpfs          tmpfs         5120       4      5116   1% /run/lock
tmpfs          tmpfs      1005108       0   1005108   0% /sys/fs/cgroup
tmpfs          tmpfs      1005108      12   1005096   1% /tmp
/dev/mmcblk0p2 ext4        122835   58092     55569  52% /boot
tmpfs          tmpfs       201020      12    201008   1% /run/user/1000
tmpfs          tmpfs       201020       0    201020   0% /run/user/0
cat@lubancat:~$
```

ホストのハードディスクデバイスは「/dev/sda1」であり、使用しているファイルシステムタイプは ext4 で、マウントポイントはルートディレクトリの「/」であることがわかります。

17.2 仮想ファイルシステム

上記で紹介した記憶装置にファイルを記録するためのファイルシステムの他に、Linux カーネルは procfs、sysfs、devfs などの仮想ファイルシステムを提供しています。

仮想ファイルシステムはメモリ内に存在し、通常はハードディスクスペースを占有しません。ファイルの形式で、システムのカーネルデータへのアクセスインターフェースをユーザーに提供します。ユーザーやアプリケーションは、これらのデータインターフェースにアクセスすることで、システムの情報を得ることができ、さらにカーネルの特定のパラメータを変更することをカーネルは許可しています。

17.2.1 procfs ファイルシステム

procfs は「process filesystem」の略であり、そのためプロセスファイルシステムとも呼ばれます。procfs は通常、ルートディレクトリ下の/proc フォルダに自動的にマウントされます。procfs はユーザーにカーネルの状態やプロセス情報のインターフェースを提供し、Windows のタスクマネージャーに相当する機能を持っています。「システム情報の確認」の章でも/proc ディレクトリを使用していくつかのシステム情報を確認しました。

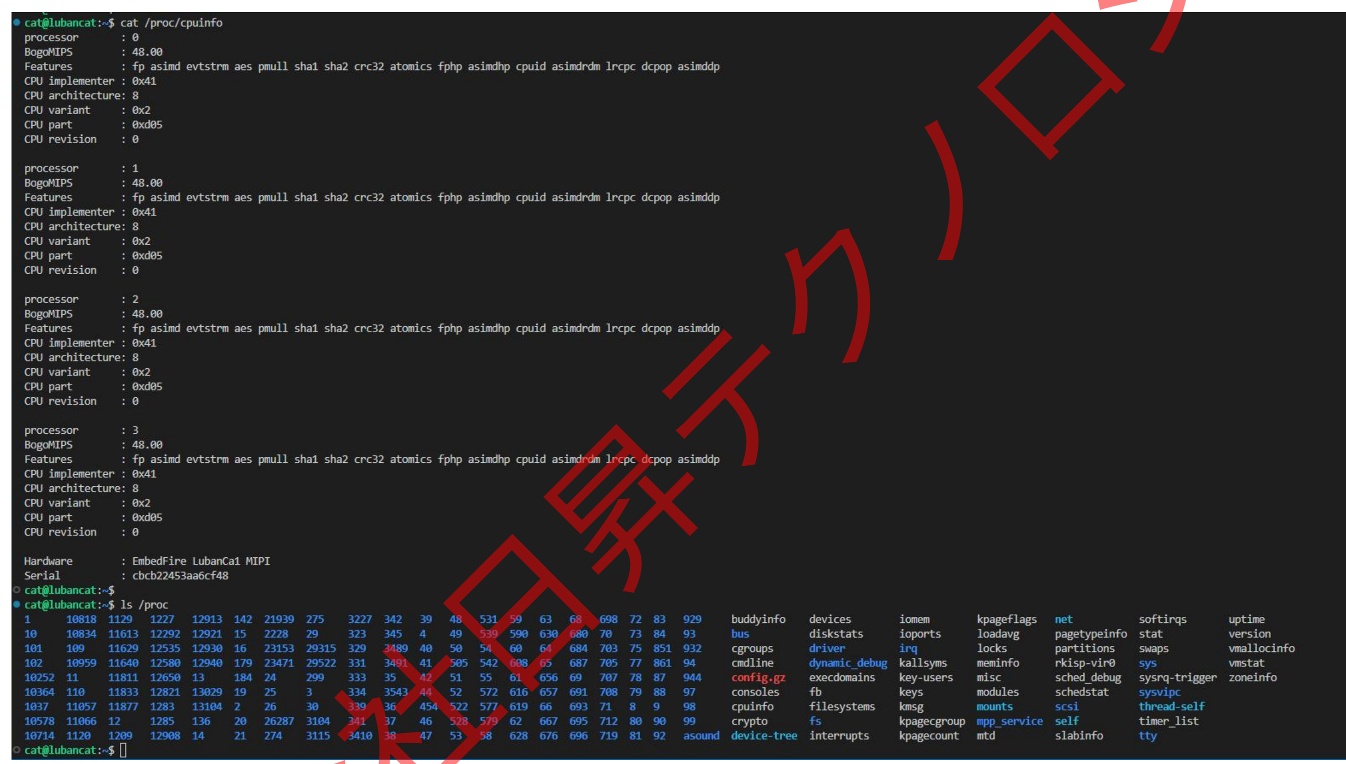
以下のコマンドを使用して proc ファイルシステムの内容を確認できます：

株式会社日昇テクノロジー

ホストまたはボード上で以下のコマンドを実行します

```
# CPU 情報を確認

cat /proc/cpuinfo

# proc ディレクトリを確認

ls /proc
```

以下の画像は、



CPU の情報を確認したことを示しており、/proc には多くの数字で命名されたディレクトリが含まれていることがわかります。これらの数字はプロセスの PID 番号です。その他のファイルやディレクトリの説明は下リストに記載されています。

/proc 内の各ファイルの役割

ファイル名	役割
pid*	「*」はプロセス ID をリストし、システム上で現在実行中の各プロセスには対応するディレクトリがあり、プロセスに関連するすべての情報を記録します。オペレーティングシステムにとって、アプリケーションはプロセスです。
self	このファイルはソフトリンクであり、現在のプロセスのディレクトリを指しています。/proc/self/ディレクトリにアクセスすることで、毎回 pid を取得することな

株式会社日昇テクノロジー

	く、現在のプロセスの情報を取得できます。
thread-self	このファイルもソフトリンクであり、現在のスレッドを指しています。このファイルにアクセスすることは、「現在のプロセス pid/task/現在のスレッド tid」の内容にアクセスすることと等価です。プロセスは複数のスレッドを含むことができますが、少なくとも 1 つのプロセスが必要であり、これらのスレッドがプロセスの実行を支えます。
version	現在実行中のカーネルバージョンを記録しており、通常はコマンド「uname –r」で確認できます。
cpuinfo	システム内の CPU の提供者と関連する設定情報を記録します。
modules	現在システムにロードされているモジュール情報を記録します。
meminfo	システム内のメモリ使用状況を記録しており、free コマンドはこのファイルにアクセスして、システムのメモリ空き容量と使用量を取得します。
filesystems	カーネルがサポートするファイルシステムのタイプを記録しています。通常、デバイスを mount する際にファイルシステムを指定せず、かつそれを特定できない場合、mount はこのファイルに含まれるファイルシステムを試みます。「nodev」とマークされたファイルシステムを除く。

ここでは、現在の bash プロセスの PID ディレクトリを例にして、proc ファイルシステムのいくつかの機能について説明します。

以下のコマンドを使用して、現在の bash プロセスの PID を確認します。

```
# ホストで以下のコマンドを実行します

ps
```

各ユーザーのコンピューターの実行状況が異なるため、取得するプロセス番号も異なります。以下のように、現在の bash プロセスの pid は 10252 となっています。

```
cat@lubancat:~$ ps
  PID TTY          TIME CMD
10252 pts/2    00:00:02 bash
13462 pts/2    00:00:00 ps
cat@lubancat:~$
```

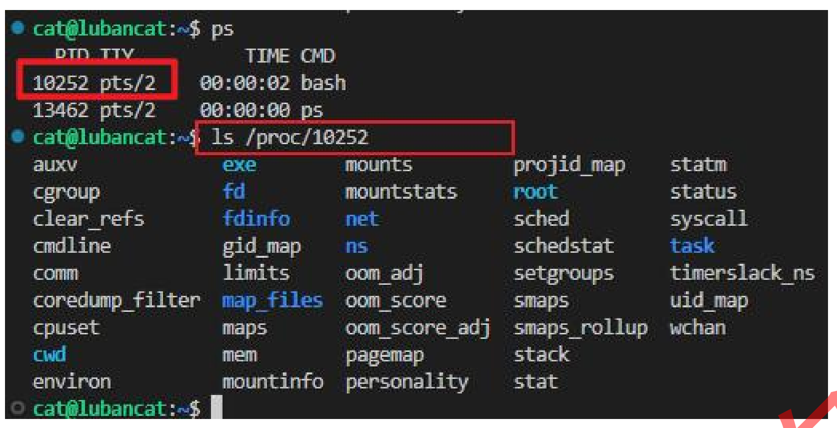
この pid 番号を使用して、proc/3042 ディレクトリの内容を確認します。これはプロセスの実行に関連する情報を記録しています。

以下のコマンドを実行します：

株式会社日昇テクノロジー

```
# ホストで以下のコマンドを実行します

# ディレクトリ内の数字を自分の bash プロセスの pid に変更します

ls /proc/3042
```

次の図に示します。



株式会社日昇テクノロジー

このディレクトリには、以下のリストに示すようなファイルとフォルダが含まれています。

リストフォルダとファイル内容

ファイル名	ファイル内容
cmdline	読み取り専用ファイルで、そのプロセスのコマンドライン情報（コマンドおよびコマンドパラメータ）を記録します。
comm	プロセスの名前を記録します。
environ	プロセスが使用する環境変数。
exe	ソフトリンクファイルで、コマンドが保存されている絶対パスを記録します。
fd	プロセスが開いたファイルの状況を記録し、ファイルディスクリプタをディレクトリ名としています。
fdinfo	プロセスが開いたファイルの関連情報を記録し、アクセス権限やマウントポイントが含まれ、そのファイルディスクリプタによって名付けられます。
io	プロセスの読み取りと書き込みの状況を記録します。
map_files	メモリ内のファイルのマッピング状況を記録し、対応するメモリ領域の開始アドレスと終了アドレスで命名します。
maps	現在マップされているメモリ領域、そのアクセス権限、およびファイルパスを記録します。
stack	現在のプロセスのカーネル呼び出しスタック情報を記録します。
status	ロセスの状態情報を記録します。
syscall	現在のプロセスが実行中のシステムコールを表示します。第一列はシステムコール番号を記録します。
task	そのプロセスのスレッド情報を記録します。

wchan	現在のプロセスが睡眠状態にあり、カーネル呼び出しの関連関数を記録します。

ファイルの場合、cat コマンドを使用して対応するファイルの内容を直接表示できます。例えば、プロセス名を確認するには：

```
# ホストで以下のコマンドを実行します
# ディレクトリ内の数字を自分の bash プロセスの pid に変更します
cat /proc/10252/comm
```

```
cat@lubancat:~$ cat /proc/10252/comm
bash
cat@lubancat:~$
```

これにより、プロセス名が「bash」であることがわかります。実際に、「ps」コマンドも「proc」ファイルシステムを通じて関連するプロセス情報を取得しています。

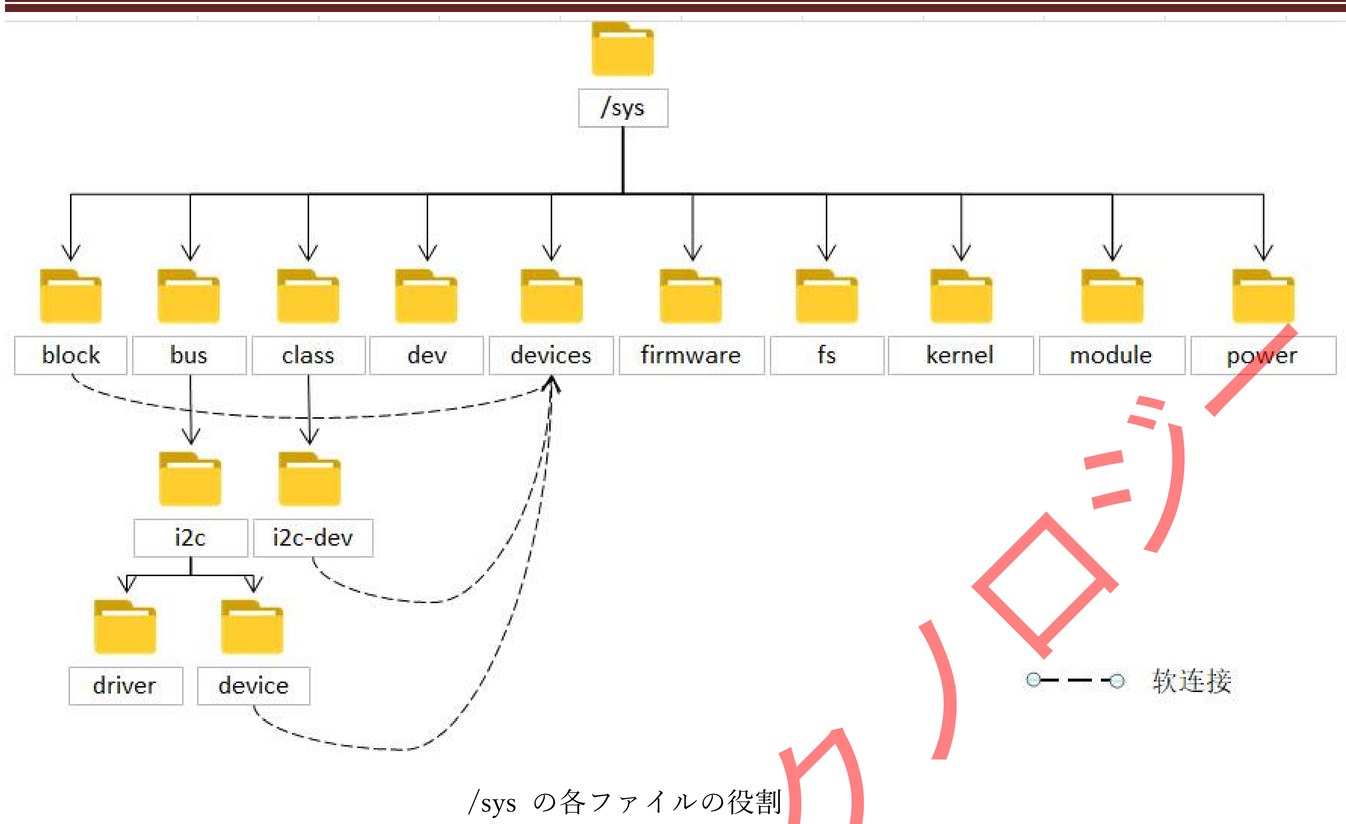
17.2.2 sysfs ファイルシステム

前節で触れた procfs は「タスクマネージャー」でしたが、sysfs も procfs と同様に仮想ファイルシステムですが、その役割は何でしょうか？

Linux カーネルのバージョン 2.6 では sysfs ファイルシステムが導入され、通常はルートディレクトリの下にある sys フォルダに自動的にマウントされます。sys ディレクトリにあるファイルやフォルダは、デバイス、カーネルモジュール、ファイルシステム、その他のカーネルコンポーネントに関する情報をユーザーに提供します。たとえば、子ディレクトリ block にはすべてのブロックデバイスが、bus にはシステム内のすべてのバスタイプ（i2c、usb、sdio、pci など）が格納されています。下の図の点線はシンボリックリンクをリストし、デバイスに関連するすべてのファイルやフォルダが device ディレクトリにリンクされていることがわかります。これにより、/sys フォルダの構造が明確でわかりやすくなっています。

下の通り

株式会社日昇テクノロジー



/sys の各ファイルの役割

ファイル名	その用途
block	システムに登録されているすべてのブロックデバイスを記録します。これらのファイルはシンボリックリンクであり、すべて/sys/devices ディレクトリを指しています。
bus	システム内のすべてのバスタイプを含んでいます。各フォルダーは、各バスのタイプに基づいて命名されています。
class	システムに登録されているすべてのデバイスタイプ（ブロックデバイス、サウンドカード、ネットワークカードなど）を含んでいます。フォルダー下のファイルもまた、リンクファイルであり、/sys/devices ディレクトリを指しています。
devices	システム内のすべてのデバイスを含み、ルートデバイスに関連するファイル/フォルダーは最終的にこのフォルダーを指します。
module	システムにロードされたすべてのカーネルモジュールを記録しているディレクトリで、各フォルダー名はモジュールの名前で命名されています。fs
fs	システムに登録されたファイルシステムを含んでいます。

要約すると、sysfs ファイルシステムは、カーネルがドライバをロードする際に、システム上のデバイスとバスの構成に基づいてエクスポートされる階層的なディレクトリです。これはシステム上のデバイスの直感的な反映であり、sysfs 下の各デバイスには一意の対応するディレクトリがあります。ユーザーは、具体的なデバイスディレクトリ下のファイルを通じてデバイスにアクセスできます。

「コマンドを使用して LED を点灯させるとキー検出する」章の例では、sysfs ファイルシステムにアク

株式会社日昇テクノロジー

セスして LED を制御することに成功しました。

17.2.3 devfs ファイルシステム

Linux 2.6 カーネル以前では、デバイスを管理するために devfs ファイルシステムが使用され、通常は /dev ディレクトリにマウントされます。devfs 内の各ファイルはデバイスに対応しており、ユーザーは /dev ディレクトリ内のファイルを介してハードウェアにアクセスできます。sysfs が登場する前は、devfs はルートファイルシステムを作成する際にすでに固定されていましたが、現代の devfs は通常、システムの実行時に udev というツールを使用して sysfs ディレクトリに基づいて devfs ディレクトリを生成します。ルートファイルシステムの作成を学ぶ際に、静的な devfs と udev を使用して動的に生成される devfs の選択肢について触れることになります。

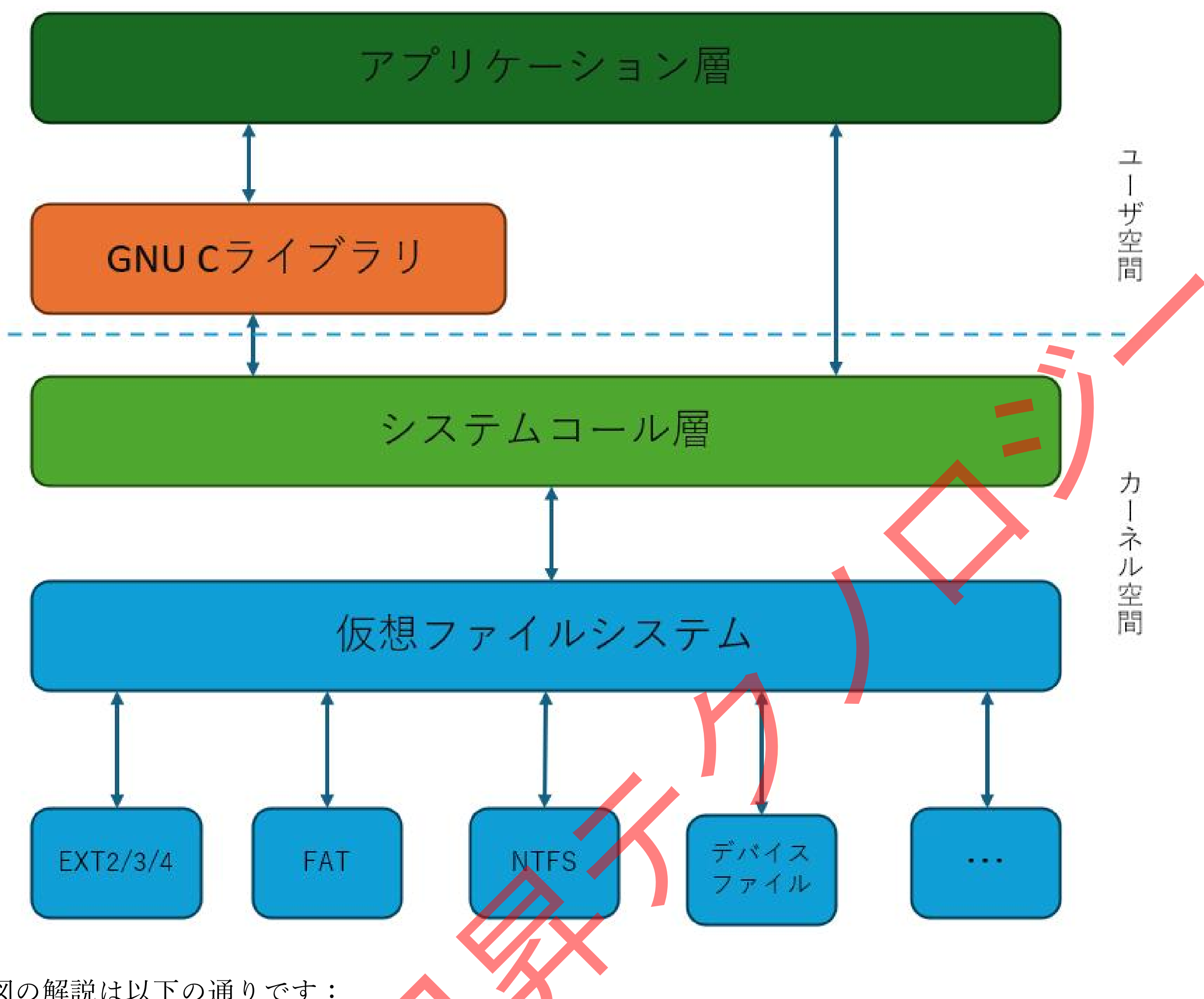
17.3 仮想ファイルシステム

前述の FAT32、ext4 のストレージファイルシステム、仮想ファイルシステム/proc、/sys、/dev のほかに、ramfs のメモリファイルシステム、nfs のネットワークファイルシステムなどがあります。異なるファイルシステム標準には、異なるプログラムロジックを使用してアクセスを実現する必要があり、提供されるアクセスインターフェースにもわずかな違いがあるかもしれません。しかし、アプリケーションを開発する際には、fopen、fread、fwrite などの C 標準ライブラリ関数を介してファイルにアクセスすることが多いです。これはすべて仮想ファイルシステムのおかげです。

Linux カーネルには、ファイル管理サブシステムコンポーネントが含まれており、主に仮想ファイルシステム（Virtual File System、VFS）を実装しています。仮想ファイルシステムは、各種ハードウェア上の違いや具体的な実装の詳細を隠蔽し、すべてのハードウェアデバイスに統一されたインターフェースを提供することで、デバイスの独立性を実現しています。また、アプリケーション層に統一された API インターフェースを提供しています。

Linux 下でのファイル操作に関連するアプリケーションの構造は下図の通りです。

株式会社日昇テクノロジー



この図の解説は以下の通りです：

- アプリケーション層はユーザーが作成したプログラムを指し、例えば hello.c のようなものです。

- GNU C ライブラリ（glibc）は C 言語の標準ライブラリで、コンパイラの章で紹介された libc.so.6 ファイルなどがこれにあたります。printf、malloc、そしてこの章で使用される fopen、fread、fwrite などのファイル操作関数が含まれています。

- ユーザープログラムと glibc ライブラリはどちらもユーザー空間に属しており、本質的にはユーザープログラムです。

- アプリケーション層のプログラムと glibc は「システムコール層（SCI）」の関数を呼び出すことがあります。これらの関数は Linux カーネルが外部に提供する関数インターフェースで、ユーザーはこれらの関数を通じてシステムに操作を要求します。例えば、C ライブラリの printf 関数はシステムの vsprintf と write 関数を使用し、C ライブラリの fopen、fread、fwrite はそれぞれシステムの open、

株式会社日昇テクノロジー

read、write 関数を呼び出します。これについては glibc のソースコードを読むことで詳細を理解できます。

- ファイルシステムの種類が非常に多いため、open、read、write などのファイル操作に関連する関数は仮想ファイルシステム層を経由して具体的なファイルシステムにアクセスします。

総じて、異なるファイルシステムが共存するために、Linux カーネルはユーザー層と具体的なファイルシステムの間に仮想ファイルシステムの中間層を追加しました。これは複雑なシステムを抽象化し、ユーザーに統一されたファイル操作インターフェースを提供します。ext2/3/4、FAT32、NTFS で保存されたファイル、/proc、/sys が提供する情報、またはハードウェアデバイスであろうと、内容がローカルであろうとネットワーク上であろうと、同じ open、read、write を使用してアクセスします。これにより「すべてはファイルである」という理念が実現され、これこそがソフトウェアの中間層の魅力です。

17.4 Linux システムコール

上記の図からわかるように、システムコール（System Call）は、操作システムがユーザープログラムに提供する一連の「特別な」関数インターフェース API であり、ファイル操作はその一種です。実際に、Linux が提供するシステムコールには以下の内容が含まれます：

- プロセス制御：fork、clone、exit、setpriority などのプロセスの作成、終了、プロセス優先度の設定など。

- ファイルシステム制御：open、read、write などのファイルの開閉、読み取り、書き込み操作。

- システム制御：reboot、stime、init_module などのシステムの再起動、システム時間の調整、モジュールの初期化など。

- メモリ管理：mlock、mremap などのメモリページのロックや仮想メモリの再マッピング操作。

- ネットワーク管理：sethostname、gethostname などのホスト名の設定や取得。

- ソケット制御：socket、bind、send などの TCP、UDP のネットワーク通信操作。

- ユーザー管理：setuid、getuid などのユーザー ID の設定や取得。

株式会社日昇テクノロジー

- プロセス間通信：セマフォ、パイプ、共有メモリなどの操作。

システムコールは、論理的に見れば、Linux カーネルとユーザースペースプログラム間の仲介者と見なすことができます。ユーザープロセスのリクエストをカーネルに伝え、カーネルがリクエストを処理した後、結果をユーザースペースに返します。これは、ユーザースペースとカーネルスペースを隔離し、ユーザーに指定された方法でシステムリソースにアクセスさせることで、システムを保護する目的を果たすためのものです。

つまり、注目している Linux アプリケーションとハードウェアドライバーの間には、さまざまなシステムコールが存在し、どのような目的であれ、システムコールの学習は Linux 開発を理解する上で避けては通れません。

この章では、ファイル操作の二つの実験を通じて、C 標準ライブラリとシステムコールを使用する違いを示します。

17.5 ファイル操作（C 標準ライブラリ）

このセクションでは、一般的な C 標準ライブラリインターフェースを使用してファイルにアクセスする方法について説明します。標準ライブラリは実際にはシステムコールを再びラップしています。C 標準ライブラリを使用して書かれたコードは、異なるシステム間で容易に移植することができます。

例えば、Windows システムでのファイルを開く操作のシステム API は OpenFile であり、Linux では open ですが、C 標準ライブラリはこれらを fopen としてラップしています。Windows の C ライブラリは fopen を通じて OpenFile 関数を呼び出して操作を行い、Linux では glibc を通じて open を呼び出してファイルを開きます。ユーザーコードが fopen を使用する場合、異なるシステムに応じてプログラムを再コンパイルするだけで、対応するコードを変更する必要はありません。

株式会社日昇テクノロジー

17.5.1 一般的なファイル操作（C 標準ライブラリ）

開発時には、馴染みのないライブラリ関数やシステムコールに遭遇した場合、インターネットで検索するのではなく、man マニュアルを活用することが大切です。C 標準ライブラリが提供する一般的なファイル操作について簡単に説明します。

17.5.1.1 fopen 関数

fopen ライブラリ関数は、ファイルを開いたり作成したりするために使用され、対応するファイルストリームを返します。その関数プロトタイプは以下の通りです：

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *mode);
```

- pathname パラメータは、開くまたは作成するファイル名を指定します。

- mode パラメータは、ファイルの開き方を指定します。このパラメータは文字列で、入力時にはダブルクォーテーションが必要です：

- "r"：読み取り専用で開く、ファイルポインタはファイルの先頭に位置します。

- "r+"：読み書きの方式で開く、ファイルポインタはファイルの先頭に位置します。

- "w"：書き込み専用で開く、元のファイルに内容があってもすべて消去します、ファイルポインタはファイルの先頭に位置します。

- "w+"：上記と同じですが、ファイルが存在しない場合、前の"w"モードはエラーを返しますが、この場合は新しいファイルを作成します。

- "a"：追加モードで開く、ファイルが存在しない場合は新しいファイルを作成します、ファイルポインタはファイルの末尾に位置します。"w+"との違いは、元のファイルの内容を消去せずに追加することです。

- "a+"：読み取りと追加の方式で開く、他は上記と同じです。

株式会社日昇テクノロジー

- fopen の戻り値は FILE 型のファイルストリームで、NULL でない場合は正常に開かれたことを意味し、その後の fread、fwrite などの関数はこのファイルストリームを通じて対応するファイルにアクセスできます。

17.5.1.2 fread 関数

fread ライブラリ関数は、ファイルストリームからデータを読み取るために使用されます。関数プロトタイプは以下の通りです：

```
1 #include <stdio.h>
2 size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

stream は fopen を使用して開かれたファイルストリームで、fread はそれを使用してアクセスしたいファイルを指定します。それはそのファイルから nmemb 項目のデータを読み取り、各項目のサイズは size で、読み取られたデータは ptr が指す配列に格納されます。fread の戻り値は成功した読み取り項目数（項目の単位は size）です。

17.5.1.3 fwrite 関数

fwrite ライブラリ関数は、データをファイルストリームに書き込むために使用されます。関数プロトタイプは以下の通りです：

```
1 #include <stdio.h>
2 size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

その操作は fread とは逆で、ptr 配列の内容を stream ファイルストリームに書き込みます。書き込まれる項目数は nmemb で、各項目のサイズは size です。戻り値は成功した書き込み項目数（項目の単位は size）です。

株式会社日昇テクノロジー

17.5.1.4 fclose 関数

fclose ライブラリ関数は、指定されたファイルストリームを閉じるために使用されます。閉じる際には、まだファイルに書き出されていない内容がすべて書き出されます。標準ライブラリはデータをバッファリングするため、データが書き出されることを保証するために fclose を使用する必要があります。関数プロトタイプは以下の通りです：

```
1 #include <unistd.h>
2 int close(int fd);
```

17.5.1.5 fflush 関数

fflush 関数は、まだファイルに書き出されていない内容を直ちに書き出すために使用されます。前の操作のデータがディスクに書き込まれることを保証するためによく使用されます。

fclose 関数自体にも fflush の操作が含まれています。fflush の関数プロトタイプは以下の通りです：

```
1 #include <stdio.h>
2 int fflush(FILE *stream);
```

17.5.1.6 fseek 関数

fseek 関数は、次回の読み書き関数操作の位置を設定するために使用されます。関数プロトタイプは以下の通りです：

```
1 #include <stdio.h>
2 int fseek(FILE *stream, long offset, int whence);
```

offset パラメータは位置を指定するために使用され、whence パラメータは offset の意味を定義します。

whence の取り得る値は以下の通りです：

- SEEK_SET：offset は絶対位置です。
- SEEK_END：offset はファイルの末尾を基準とした相対位置です。

株式会社日昇テクノロジー

• SEEK_CUR：offset は現在の位置を基準とした相対位置です。

17.5.2 実験コード分析

以下に示すように、C 標準ライブラリを使用してファイル操作の実験を行います。

リスト 1: ファイル操作実験-C 標準ライブラリ

（base_linux/file_io/stdio/stdio.c ファイル）

```
1 #include <stdio.h>
2 #include <string.h>
3
4 // 書き込む文字列
5 const char buf[] = "filesystem_test:Hello World!¥n";
6 // ファイルディスクリプタ
7 FILE *fp;
8 char str[100];
9
10
11 int main(void)
12 {
13 // ファイルを作成
14 fp = fopen("filesystem_test.txt", "w+");
15 // 正常にファイルポインタを返す
16 // 例外は NULL を返す
17 if(NULL == fp){
```

株式会社日昇テクノロジー

```
18 printf("Fail to Open File¥n");

19 return 0;

20 }

21 // buf の内容をファイルに書き込む

22 // 1 回あたり 1 バイト、合計長さは strlen で与えられる

23 fwrite(buf, 1, strlen(buf), fp);

24

25 // "Embedfire"を書き込む

26 // 1 回あたり 1 バイト、合計長さは strlen で与えられる

27 fwrite("Embedfire¥n", 1, strlen("Embedfire¥n"), fp);

28 // バッファのデータを直ちにファイルに書き込む

30 fflush(fp);

31

32 // この時点でのファイル位置ポインタはファイルの末尾にあるため、fseek 関数を使ってファイルポイ
ンタをファイルの先頭に戻す

33 fseek(fp, 0, SEEK_SET);

34

35 // ファイルから str に内容を読み込む

36 // 1 回あたり 100 バイトを読み取り、1 回読み取る

37 fread(str, 100, 1, fp);

38

39 printf("File content:¥n%s ¥n", str);
```

株式会社日昇テクノロジー

```
40

41 fclose(fp);

42

43 return 0;

44 }
```

C 言語のファイル操作を学習したことがあれば、この実験コードは非常に理解しやすいでしょう。流れとしては、fopen でファイルを作成し、fwrite で内容を書き込み、fflush でバッファの内容をファイルに確実に書き出し、fseek でファイル位置ポインタをリセットし、fread でファイルの内容を読み出し、最後に fclose でファイルを閉じる、というものです。

fopen 関数呼び出し時に使用される引数"w+"は、新しい空のファイルを毎回作成し、読み取り権限付きで開くことを意味します。関数呼び出し後にファイルディスクリプタ fp が得られ、以降の fwrite、fread、fflush などの関数はこの fp ファイルディスクリプタを通じてファイルにアクセスします。

fwrite と fread の間にある fflush 関数が最も大きな違いです。C 標準ライブラリのファイルシステムはバッファを持っているのに対し、システムコールのファイル操作にはバッファがなく、それに対応する flush 関数がありません。

17.5.3 Makefile 説明

Makefile はプロジェクトディレクトリに合わせており、この実験では C ファイルが 1 つだけであり、Makefile と同じディレクトリに位置しています。プロジェクトのファイル構造は以下の図のようになっています。

```
cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/stdio$ tree
.
├── Makefile
└── stdio.c

0 directories, 2 files
cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/stdio$
```

ここで記述された Makefile は前述の「デフォルトルールの変更」セクションと基本的に一致しています

株式会社日昇テクノロジー

が、以下のような違いがあります。

リスト 2: ファイル操作実験-C 標準ライブラリ

（base_linux/file_io/stdio/Makefile ファイル）

```
1 # 変数定義
2 TARGET = stdio
3 # コンパイラの定義
4 CC = gcc
5 # ヘッダーファイルの位置の定義()
6 CFLAGS = -I.
7 # ヘッダーファイルの定義
8 DEPS =
9 # オブジェクトファイルの定義
10 OBJS = $(TARGET).o
11 # .o ファイルの出力位置の定義
12 BUILD_DIR = build
13
14 # ターゲットファイル
15 $(TARGET): $(OBJS)
16 $(CC) -o $@ $^ $(CFLAGS)
17 # コンパイル出力フォルダの作成
18 @mkdir -p $(BUILD_DIR)
19 # .o ファイルを出力フォルダに移動
```

株式会社日昇テクノロジー

```
20 @mv *.o $(BUILD_DIR)
21 # 実行可能プログラムを出力フォルダにコピー
22 @cp $(TARGET) $(BUILD_DIR)
23
24 #*.o ファイルの生成ルール
25 %.o: %.c $(DEPS)
26 $(CC) -c -o $@ $< $(CFLAGS)
27
28 # ターゲット
29 .PHONY: clean
30 #make clean でコンパイル結果をクリア
31 clean:
32 # 実行可能プログラムを削除
33 rm -f $(TARGET)
34 # 出力フォルダを削除
35 rm -rf $(BUILD_DIR)
36 # プログラムが作成したファイルを削除
37 rm -f filesystem_test.txt
```

株式会社日昇テクノロジー

先に説明した Makefile との主な違いは以下の通りです：

1. 2 行目、10 行目：ファイルが 1 つに変更されました。

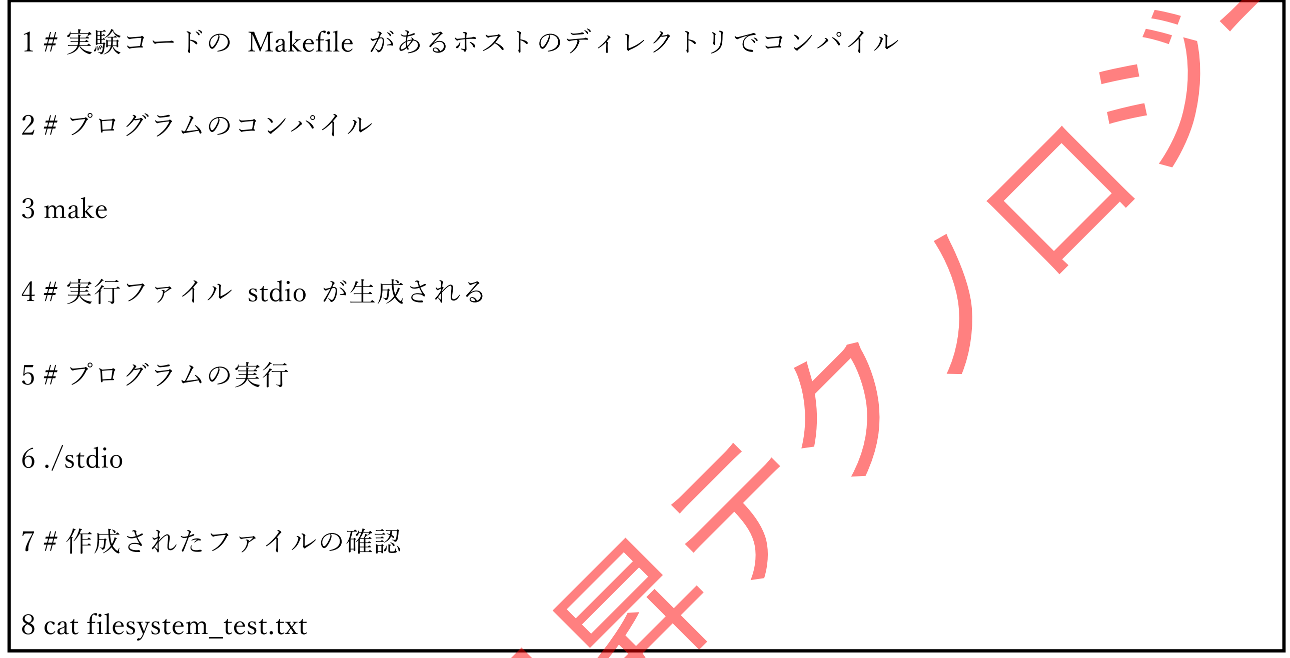
2. 12 行目：ビルド成果物を保存するための build フォルダが追加されました。

3. 17-22 行目：フォルダを作成し、ビルド後の成果物を移動します。

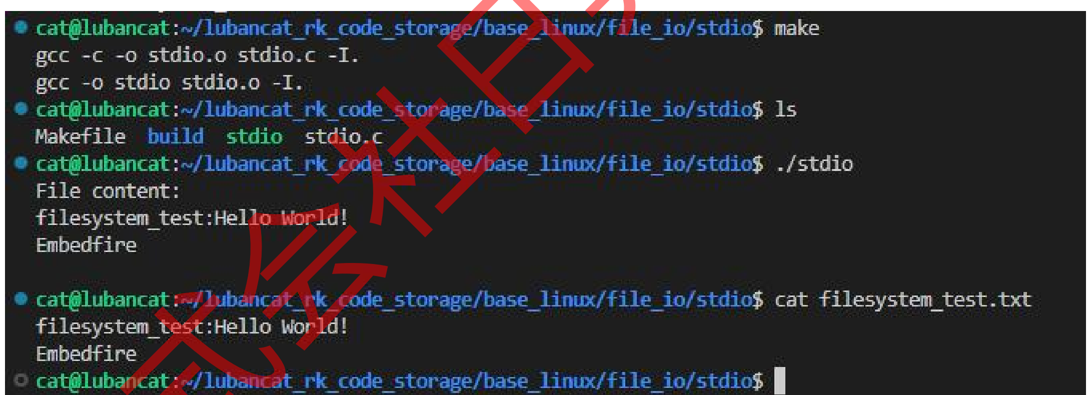
4. 32-37 行目：プログラムとプログラム実行によって作成されたファイルを削除し、ビルドプログラムの成果物を削除します。

17.5.4 コンパイルとテスト

以下の手順でコンパイルを行います：



```
1 # 実験コードの Makefile があるホストのディレクトリでコンパイル
2 # プログラムのコンパイル
3 make
4 # 実行ファイル stdio が生成される
5 # プログラムの実行
6 ./stdio
7 # 作成されたファイルの確認
8 cat filesystem_test.txt
```



17.6 ファイル操作（システムコール）

Linux が提供するファイル操作のシステムコールには、open、write、read、lseek、close などがあります。

株式会社日昇テクノロジー

17.6.1 open 関数

```
1 #include <sys/types.h>

2 #include <sys/stat.h>

3 #include <fcntl.h>

4 int open(const char *pathname, int flags);

5 int open(const char *pathname, int flags, mode_t mode);
```

Linux では、open 関数を使用してファイルを開き、そのファイルに対応するファイルディスクリプタを返します。関数パラメータの具体的な説明は以下の通りです：

- pathname：開くまたは作成するファイル名；
- flags：ファイルの開き方を指定するパラメータで、以下のリストに示す flag パラメータ値があります。

リスト flag パラメータ値

フラグ	意味
O_RDONLY	ファイルを読み取り専用で開く。このパラメータは O_WRONLY と O_RDWR の 3 つのうちの 1 つだけを選択できる
O_WRONLY	ファイルを書き込み専用で開く
O_RDWR	ファイルを読み書きの両方で開く
O_CREAT	新しいファイルを作成する
O_APPEND	データを現在のファイルの末尾に書き込む
O_TRUNC	pathname ファイルが存在する場合、その内容をクリアする

C ライブラリ関数 fopen の mode パラメータとシステムコール open の flags パラメータは、以下のリストに示すように等価な関係があります。

リスト fopen の mode と open の flags パラメータの関係

fopen の mode パラメータ	open の flags パラメータ
r	O_RDONLY
w	O_WRONLY | O_CREAT | O_TRUNC

株式会社日昇テクノロジー

a	O_WRONLY | O_CREAT | O_APPEND
r+	O_RDWR
w+	O_RDWR | O_CREAT | O_TRUNC
a+	O_RDWR | O_CREAT | O_APPEND

• mode：open 関数の flags 値に O_CREAT が設定されている場合、mode パラメータを使用してファイルとユーザー関連の権限を設定する必要があります。mode で使用できる権限は、以下のリストに示されているように、「|」で組み合わせて使用できるパラメータです。

リストファイル権限

	フラグ	意味
現在のユーザー	S_IRUSR	ユーザーは読み取り権限を持つ
	S_IWUSR	ユーザーは書き込み権限を持つ
	S_IXUSR	ユーザーは実行権限を持つ
	S_IRWXU	ユーザーは読み取り、書き込み、実行の権限を持つ
現在のユーザーグループ	S_IRGRP	現在のユーザーグループの他のユーザーは読み取り権限を持つ
	S_IWGRP	現在のユーザーグループの他のユーザーは書き込み権限を持つ
	S_IXGRP	現在のユーザーグループの他のユーザーは実行権限を持つ
	S_IRWXG	現在のユーザーグループの他のユーザーは読み取り、書き込み、実行の権限を持つ
他のユーザー	S_IROTH	他のユーザーは読み取り権限を持つ
	S_IWOTH	他のユーザーは書き込み権限を持つ
	S_IXOTH	他のユーザーは実行権限を持つ
	S_IRWXO	他のユーザーは読み取り、書き込み、実行の権限を持つ

17.6.2 read 関数

```
1 #include <unistd.h>

2 ssize_t read(int fd, void *buf, size_t count);
```

read 関数は、ファイルからいくつかのバイトのデータを読み取り、データバッファ buf に保存し、実際に読み取ったバイト数を返します。具体的な関数パラメータは以下の通りです：

株式会社日昇テクノロジー

• fd：ファイルに対応するファイルディスクリプタで、fopen 関数を通じて取得できます。また、プログラムが実行されるとき、Linux は標準入力、標準出力、標準エラー出力に対応する 0、1、2 の 3 つの既に開かれているファイルディスクリプタをデフォルトで持っているため、これらのファイルディスクリプタに直接アクセスすることができます；

• buf：データバッファへのポインタ；

• count：読み取るバイト数。

17.6.3 write 関数

```
1 #include <unistd.h>
2 ssize_t write(int fd, const void *buf, size_t count);
```

write 関数は、ファイルに内容を書き込み、実際に書き込まれたバイト長を返します。具体的な関数パラメータは以下の通りです：

• fd：ファイルに対応するファイルディスクリプタで、fopen 関数を通じて取得できます。

• buf：データバッファへのポインタ；

• count：ファイルに書き込むバイト数。

17.6.4 close 関数

```
1 int close(int fd);
```

ファイルの操作を終えた後、そのファイルを閉じたい場合は、close 関数を呼び出してその fd ファイルディスクリプタに対応するファイルを閉じることができます。

17.6.5 lseek 関数

lseek 関数は、ファイルポインタの位置を設定し、ファイル先頭からのファイルポインタの位置を返すために使用できます。その関数プロトタイプは以下の通りです。

株式会社日昇テクノロジー

```
1 off_t lseek(int fd, off_t offset, int whence);
```

その使用法は flseek と同様で、offset パラメータは位置を指定するために使用され、whence パラメータは offset の意味を定義します。whence の取りうる値は以下の通りです：

・SEEK_SET：offset は絶対位置です。

・SEEK_END：offset はファイル末尾を基準点とした相対位置です。

・SEEK_CUR：offset は現在位置を基準点とした相対位置です。

17.7 実験コード分析

このセクションでは、ファイル操作を行う方法について、ある実験を通じて説明します。

リスト 3: ファイル操作実験-システムコール

（base_linux/file_io/systemcall/systemcall.c ファイル）

```
1 #include <sys/stat.h>

2 #include <unistd.h>

3 #include <fcntl.h>

4 #include <stdio.h>

5 #include <string.h>

6

7 //ファイルディスクリプタ

8 int fd;

9 char str[100];

int main(void)

13 {

14 //ファイルを作成

15 fd = open("testscript.sh", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU);
```

株式会社日昇テクノロジー

```
16 //ファイルディスクリプタ fd は非負の整数
17 if(fd < 0){
18 printf("ファイルを開けません¥n");
19 return 0;
20 }
21 //文字列 pwd を書き込み
22 write(fd, "pwd¥n", strlen("pwd¥n"));
23
24 //文字列 ls を書き込み
25 write(fd, "ls¥n", strlen("ls¥n"));
26
27 //この時点でのファイルポインタはファイルの末尾にあり、lseek 関数を使用してファイルポインタをファイルの頭に戻す
28 lseek(fd, 0, SEEK_SET);
29
30 //ファイルから 100 バイトの内容を str に読み込み、この関数は実際に読み込んだバイト数を返します
31 read(fd, str, 100);
32
33 printf("ファイル内容:¥n%s ¥n", str);
34
```

株式会社日昇テクノロジー

```
35 close(fd);
```

```
36

37 return 0;

38 }
```

17.7.1 実行フロー

この実験は、ファイルを作成し、内容を書き込み、そして読み出すという、C ライブラリのファイル操作と似ていますが、ここでは open、write、lseek、read、close などのシステムコール関数を使用しています。具体的には以下の通りです：

・コードではまず、読み書き可能な方法でテキストファイルを開くために open 関数が呼ばれ、O_CREAT はファイルが存在しない場合に新しいファイルを作成することを指定し、ファイルの権限は S_IRWXU、つまり現在のユーザーに読み書き実行可能で、現在のユーザーグループと他のユーザーにはいかなる権限も与えないことを意味します。

・open と fopen の返り値は似ており、どちらもファイルディスクリプタを使用しますが、open は正常時に非負の整数を返し、失敗時には-1 を返します。一方、fopen は失敗時に NULL を返します。

・ファイルを作成した後、write 関数を呼び出して「pwd¥n」、「ls¥n」という文字列、実際には単純な Shell コマンドを書き込みます。

・read 関数で内容を読み取る前に、lseek 関数を呼び出してファイルポインタをファイルの開始位置にリセットします。C ライブラリのファイル操作との違いは、write と read の間に fflush を使用してバッファの内容を確実に書き込む必要がないことです。なぜなら、システムコールのファイル操作にはバッファがないからです。

株式会社日昇テクノロジー

・最後にファイルを閉じ、ファイルディスクリプタを解放します。

17.7.2 ヘッダーファイルのディレクトリ

サンプルコードの冒頭には、Linux システムで一般的に使用される一連のヘッダーファイルが含まれています。Linux を学ぶ過程で、さまざまなヘッダーファイルに触れる可能性がありますので、Linux でのヘッダーファイルの使用方法を理解することは非常に重要です。

Linux では、ほとんどのヘッダーファイルはシステムの「/usr/include」ディレクトリにあります。これは、システムに付属の GCC コンパイラがデフォルトで使用するヘッダーファイルのディレクトリです。下図のように、このディレクトリの stdio.h ファイルを削除したり、名前を変更したりすると（試す場合はバックアップを取ってください）、GCC を使用して hello world プログラムをコンパイルすると、stdio.h ファイルが見つからずにエラーが発生します。

```
cat@lubancat:/usr/include$ ls
EGL                asm-generic  err.h       getopt.h        jbig.h      libmount            lzma        netdb.h     orc-0.4             proc_service.h  sched.h        stdio.h        thread_db.h  wayland-client-core.h      xen
GL                 assert.h     errno.h     gio-unix-2.0    jbig85.h    libpng              lzma.h      neteconet   paths.h             protocols       scsi           stdio_ext.h    threads.h    wayland-client-protocol.h  xf86drm.h
GLES2              blkid        error.h     glib-2.0        jbig_ar.h   libpng16            malloc.h    netinet     pciaccess.h         pthread.h       search.h       stdlib.h       time.h       wayland-client.h           xf86drmMode.h
GLES3              byteswap.h   execinfo.h  glob.h          jerror.h    libraw1394          math.h      netipx      pcre.h              pty.h           selinux        string.h       ttyent.h     wayland-cursor.h           xfce4
KHR                c++          fcntl.h     glvnd           jmorecfg.h  libsync.h           mcheck.h    netiucv     pcre_scanner.h      pwd.h           semaphore.h    strings.h      uchar.h      wayland-egl-core.h         xorg
OpenEXR            complex.h    features.h  gnu-versions.h  jpegint.h   libv4l-plugin.h     memory.h    netpacket   pcre_stringpiece.h  rdma            sepol          stropts.h      ucontext.h   wayland-egl.h              zconf.h
X11                cpio.h       fenv.h      gnumake.h       jpeglib.h   libv4l1-videodev.h  misc        netrom      pcrecpp.h           re_comp.h       setjmp.h       sudo_plugin.h  ulimit.h     wayland-server-core.h      zlib.h
aarch64-linux-gnu  crypt.h      fmtmsg.h    gphoto2         langinfo.h  libv4l1.h           mntent.h    netrose     pcrecpparg.h        regex.h         sgtty.h        syscall.h      unistd.h     wayland-server-protocol.h
aio.h              ctype.h      fnmatch.h   grp.h           lastlog.h   libv4l2.h           monetary.h  nfs         pcreposix.h         regexp.h        shadow.h       sysexits.h     utime.h      wayland-server.h
aliases.h          dc1394       freetype2   gshadow.h       libdrm      libv4l2rds.h        mqueue.h    nl_types.h  pixman-1            reglib          signal.h       syslog.h       utmp.h       wayland-util.h
alloca.h           dirent.h     fstab.h     gstreamer-1.0   libdvbv5    libv4lconvert.h     mtd         nss.h       png.h               resolv.h        sound          tar.h          utmpx.h      wayland-version.h
ar.h               dlfcn.h      fts.h       iconv.h         libexif     limits.h            net         obstack.h   pngconf.h           rga             spawn.h        tbb            uuid         wchar.h
argp.h             elf.h        ftw.h       ifaddrs.h       libgen.h    link.h              netash      opencv      pnglibconf.h        rockchip        stab.h         termio.h       values.h     wctype.h
argz.h             endian.h     gconv.h     inttypes.h      libintl.h   linux               netatalk    opencv2     poll.h              rpc             stdc-predef.h  termios.h      video        wordexp.h
arpa               envz.h       gdcm-2.8    iproute2        libkms      locale.h            netax25     openvpn     printf.h            rpcsvc          stdint.h       tgmath.h       wait.h       xcb
cat@lubancat:/usr/include$
```

コード中には、例えば sys/stat.h のように、特定のディレクトリを含むヘッダーファイルがあり、これらのヘッダーファイルはコンパイラのフォルダ内のディレクトリで見つけることができます。通常、次のコマンドを使用して検索することができます：

```
1 # このソフトウェアをインストールする必要があります
2 sudo apt install locate
3 # その後、データベースを更新する必要があります
4 sudo updatedb
5 # それから使用することができます
```

以下の図のように：

```
cat@lubancat:~$ sudo updatedb
cat@lubancat:~$ locate sys/stat.h
/usr/include/aarch64-linux-gnu/sys/stat.h
cat@lubancat:~$
```

この図では、sys/stat.h が/usr/include/aarch64-linux-gnu に存在していることが見つかりました。これはボードコンパイラのヘッダーファイルが配置されている場所の一つです。

17.7.3 一般的なヘッダーファイル

これからの学習で、以下のヘッダーファイルをよく使用することになります。特定のヘッダーファイルの内容を見たい場合は、locate コマンドを使用してそのファイルのディレクトリを見つけた後に開くことができます：

- ヘッダーファイル stdio.h：C の標準入出力（standard input & output）ヘッダーファイルで、よく使用する printf 関数がこのヘッダーファイルにあります。
- ヘッダーファイル stdlib.h：C の標準ライブラリ（standard library）ヘッダーファイルで、malloc 関数や free 関数などが含まれています。
- ヘッダーファイル sys/stat.h：ファイルの権限定義、例えば S_IRWXU や S_IWUSR など、および fstat 関数を使用してファイルの状態を問い合わせるために使用されるヘッダーファイルです。システムコールによるファイル関連の操作を行う場合、通常は sys/stat.h ファイルが必要です。
- ヘッダーファイル unistd.h：UNIX C 標準ライブラリのヘッダーファイルで、unix、linux 系のオペレーティングシステム関連の C ライブラリで、unix 系システムの POSIX 標準の記号定数ヘッダーファイルです。例えば、Linux 標準の入力ファイルディスクリプタ（STDIN）、標準出力ファイルディスクリプタ（STDOUT）、read、write などのシステムコールの宣言があります。
- ヘッダーファイル fcntl.h：unix 標準の一般的なヘッダーファイルで、open、fcntl、close などの操作が含まれています。
- ヘッダーファイル sys/types.h：Unix/Linux システムのデータタイプのヘッダーファイルで、size_t、time_t、pid_t などのタイプがよく使用されます。

株式会社日昇テクノロジー

17.8 コンパイルとテスト

この実験で使用される Makefile は、前のセクションのものとほぼ同じで、再分析する必要はありません。

```
1 # ホストの実験コード Makefile ディレクトリでコンパイルします
2 make
3 # ファイルをリストアップします
4 tree
5 # 実行します
6 ./systemcall
7 # プログラムは自身で出力を行い、ファイルを作成します
8 ls
9 # ファイルの内容を確認します
10 cat testsript.sh
11 # 生成された testscript.sh ファイルを実行します
12 ./testscript.sh
```

以下の図のように：

株式会社日昇テクノロジー

```
cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/systemcall$ make
gcc -c -o systemcall.o systemcall.c -I.
gcc -o systemcall systemcall.o -I.
cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/systemcall$ tree
.
├── Makefile
├── build
│   ├── systemcall
│   └── systemcall.o
├── systemcall
└── systemcall.c

1 directory, 5 files
cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/systemcall$ ./systemcall
File content:
pwd
ls

cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/systemcall$ ls
Makefile  build  systemcall  systemcall.c  testscript.sh
cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/systemcall$ cat testscript.sh
pwd
ls
cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/systemcall$ ./testscript.sh
/home/cat/lubancat_rk_code_storage/base_linux/file_io/systemcall
Makefile  build  systemcall  systemcall.c  testscript.sh
cat@lubancat:~/lubancat_rk_code_storage/base_linux/file_io/systemcall$
```

pwd コマンド

ls コマンド

上の図から、systemcall プログラムを実行した後、作成された testscript.sh ファイルに実行権限があり、./testscript.sh を実行するとそのスクリプトを実行できます。

17.9 どちらを選ぶべきか

C 標準ライブラリとシステムコールの両方がファイル操作を行うことができるのであれば、どちらを選択すべきでしょうか？考慮すべき要素は以下の通りです：

- システムコールの使用はシステムのパフォーマンスに影響を与えます。システムコールを実行する際には、Linux はユーザーモードからカーネルモードへの切り替えが必要で、実行が完了した後にはユーザーモードに戻ります。そのため、システムコールの使用を減らすことで、この面のオーバーヘッドを減らすことができます。たとえば、fwrite 関数によるファイルへのデータ書き込みも最終的には write システムコールを実行しますが、少量のデータを書き込む場合は、直接 write を実行する方が効率的かもしれません。しかし、頻繁な書き込み操作の場合、fwrite のバッファリングにより write の呼び出し回数を減らすことができるため、この場合は fwrite を使用する方が時間を節約できます。
- ハードウェア自体が、システムコールが一度に読み書きできるデータブロックのサイズを制限します。たとえば、特定のストレージデバイスに対する write 関数は、一度に 4kB のデータを書き込む必要が

株式会社日昇テクノロジー

あるかもしれません。そのため、実際に書き込みたいデータが 4kB 未満の場合でも、write は 4kB を書き込む必要があり、一部のスペースが無駄になります。一方、バッファを持つ fwrite 関数は、このような状況に対して、データの長さの要件を満たすまでシステムコールを実行することを避け、スペースの無駄を減らすことができます。

• ライブラリ関数にはバッファがあり、実際にいつハードウェアに内容が書き込まれるかを明確に知ることができないため、ハードウェアを確実に制御する必要がある場合は、システムコールを実行する方が好まれます。

第 18 章 デバッグツール

プログラムを書く際には避けられないバグの問題。良いデバッグツールを使用することで、バグの原因をより速く見つけ出し、簡単に解決することができます。ここでは、使い勝手の良い 2 つのデバッグツールを紹介します。

- gdb

- strace

注意: ここで紹介する 2 つのデバッグツール以外にも、Linux には様々な強力なデバッグツールがあります。読者は自分のニーズに合わせて、必要なデバッグツールを検索し、取得してください。

18.1 gdb

gdb は、非常に強力な機能を持つクラシックなデバッグツールです。

この際、コンパイルする時には-g オプションを追加し、-Og オプションで最適化する必要があります。

マルチスレッド環境では、プロセスにアタッチしてデバッグすることができます。

コマンドラインで gdb を入力し、gdb コマンドラインモードに入ります。

G db には多数のコマンドがありますが、ここでは一つ一つを挙げることはできません。ここではいくつかの基本的でよく使われるコマンドを紹介します。

株式会社日昇テクノロジー

リスト 1: 基本コマンド

コマンド	省略形	使用法	効果
start		start	プログラムの実行を開始し、main 関数の最初の文の前で停止
file		file xxx	デバッグするファイルを開く
run	r	ru	プログラムを実行する（最初の文から実行を開始）
break	b	下リスト 1-2 参照	ブレークポイントの設定
continue	c	c	プログラムの実行を続ける、次のブレークポイントまで
list	l	下リスト 1-3 参照	複数行のソースコードを表示
step	s	s	次の文を実行、その文が関数呼び出しの場合、関数に入り最初の文を実行
next	n	n	次の文を実行、その文が関数呼び出しの場合、関数内部をステップ実行しない（関数内部の文を一つ一つデバッグしない）
display	disp	disp n	ある変数をトラッキングし、停止するたびにその値を表示
print	p	p または p n	内部変数の値を表示
watch		watch	変数の値の変化を監視
backtrace	bt	bt	関数呼び出し情報（スタック）を表示
frame	f	f	スタックフレームを表示
quit	q	q	GDB 環境から退出
kill	k	k	デバッグ中のプログラムを終了
変数の 設定		set var=x	変数の値を設定
return		return x	現在の関数呼び出しを終了して指定した値を返し、一つ上の関数呼び出しの場所でプログラムの実行を停止
finish	fi	fi	現在実行中の関数を終了し、関数から抜けた後でプログラムの実行を一時停止
jump	j	j x	現在の関数呼び出しを終了して指定した値を返し、一つ上の関数呼び出しの場所でプログラムの実行を停止

株式会社日昇テクノロジー

リスト 2: リスト 1-2

コマンド	機能
break n	n 行目にブレークポイントを設定する（list と組み合わせて使用）
info breakpoints	設定されたブレークポイントをリストアップ
delete breakpoints 1	n 番目のブレークポイントを削除する（n は info breakpoints で得られたブレークポイント番号）

disable/enable n	n 番のブレークポイントを一時的に無効または有効にする

リスト 3: リスト 1-3

コマンド	機能
list	デフォルトでは、一度に 10 行を表示します。最初に使用するときは、コードの開始位置から表示します。
list n	n 行目を中心に 10 行のコードを表示します。
list functionname	functionname の関数を中心に 10 行のコードを表示します。
list -	直前に表示したソースコードの前の部分を表示します。

18.2 strace

strace はシステムコールをトレースする簡易ツールです。

最も簡単な使い方は、プログラムのライフサイクル全体のすべてのシステムコールをトレースし、その呼び出しパラメーターと戻り値をテキスト形式で出力することです。Strace は、プロセスに送られるシグナルもトレースできます。実行中のプロセスにアタッチするには strace -p <pid>を使用します。マルチスレッド環境で、特定のスレッドのシステムコールをトレースする場合は、まず ps -efL | grep <Process Name>で該当プロセスのスレッドを検索し、その後 strace –p <pid>を使用して分析します。

```
1 strace ./hello

2

3 cat@lubancat:~/test$ strace -f ./hello

4 execve("./hello", ["./hello"], 0x7fe43d6ac8 /* 48 vars */) = 0

5 brk(NULL) = 0x558e1ee000

6 faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = 0

7 openat(AT_FDCWD, "/etc/ld.so.preload", O_RDONLY|O_CLOEXEC) = 3

8 fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
```

株式会社日昇テクノロジー

9 close(3) = 0

10 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

11 fstat(3, {st_mode=S_IFREG|0644, st_size=77810, ...}) = 0

12 mmap(NULL, 77810, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fb76a7000

13 close(3) = 0

14 openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

15 read(3, "¥177ELF¥2¥1¥1¥3¥0¥0¥0¥0¥0¥0¥0¥0¥3¥0¥267¥0¥1¥0¥0¥0`¥17¥2¥0¥0¥0¥0¥0"..., 832) = 832

16 fstat(3, {st_mode=S_IFREG|0755, st_size=1450832, ...}) = 0

17 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =

0x7fb76e7000

18 mmap(NULL, 1519552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =

0x7fb7534000

19 mprotect(0x7fb768f000, 61440, PROT_NONE) = 0

20 mmap(0x7fb769e000, 24576, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x15a000) = 0x7fb769e000

21 mmap(0x7fb76a4000, 12224, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fb76a4000

22 close(3) = 0

23 mprotect(0x7fb769e000, 16384, PROT_READ) = 0

24 mprotect(0x555d37a000, 4096, PROT_READ) = 0

25 mprotect(0x7fb76eb000, 4096, PROT_READ) = 0

26 munmap(0x7fb76a7000, 77810) = 0

株式会社日昇テクノロジー

```
27 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x3), ...}) = 0
28 brk(NULL) = 0x558e1ee000
29 brk(0x558e20f000) = 0x558e20f000
30 write(1, "hello, world! This is a C progra"..., 35hello, world! This is a C program.
31 ) = 35
32 write(1, "output i=0¥n", 11output i=0
33 ) = 11
34 write(1, "output i=1¥n", 11output i=1
35 ) = 11
36 write(1, "output i=2¥n", 11output i=2
37 ) = 11
38 write(1, "output i=3¥n", 11output i=3
39 ) = 11
40 write(1, "output i=4¥n", 11output i=4
41 ) = 11
42 write(1, "output i=5¥n", 11output i=5
43 ) = 11
44 write(1, "output i=6¥n", 11output i=6
45 ) = 11
46 write(1, "output i=7¥n", 11output i=7
47 ) = 11
```

株式会社日昇テクノロジー

```
48 write(1, "output i=8¥n", 11output i=8

49 ) = 11

50 write(1, "output i=9¥n", 11output i=9

51 ) = 11

52 exit_group(0) = ?

53 +++ exited with 0 +++
```

- 第 3-29 行では、プログラムの実行環境の設定や、必要なライブラリの呼び出しが行われます。/lib/aarch64-linux-gnu/libc.so.6 は、前章で述べた glibc のライブラリファイルです。これらの環境設定により、システムがプログラムの干渉から保護されます。例えば、システムファイルを改ざんしたり、システムを破壊しようとするソフトウェアがあった場合、これらの保護環境の下では、システムはプログラムが定められた実行空間を超えて実行することを許可しません。プログラムがシステムファイルを意図せずに改ざんしたり（実行空間を超えた場合）、システムはそれを阻止します。プログラムが無限ループに陥ったりクラッシュした場合でも、システムはこれらの保護措置を頼りに干渉を受けずにプログラムを直接終了させることができます。

- 第 30-52 行は、プログラムの本体部分です。30 行目から 51 行目の出力形式に誤りがありますが、これはマルチスレッドが原因で形式が少し乱れていますが、全体としては修正後に問題はありません。正しい形式は以下の通りです。

```
1 write(1, "hello, world! This is a C program.¥n", 35) = 35

2 hello, world! This is a C program.

3

4 write(1, "output i=0¥n", 11) = 11

5 output i=0
```

なぜ画面にテキストを出力できるのかというと、Linux システムではコマンドライン出力をファイルディ

株式会社日昇テクノロジー

スクリプタとして抽象化しているからです。ファイルディスクリプタを操作することで、そのデバイスを操作できます。

```
1 # ディスクリプタの確認
2 cat@lubancat:~/test$ ls -l /dev/std*
3 lrwxrwxrwx 1 root root 15 Apr 21 20:54 /dev/stderr -> /proc/self/fd/2
4 lrwxrwxrwx 1 root root 15 Apr 21 20:54 /dev/stdin -> /proc/self/fd/0
5 lrwxrwxrwx 1 root root 15 Apr 21 20:54 /dev/stdout -> /proc/self/fd/1
```

- stdin は標準入力(fd=0)で、端末デバイスからの入力を意味します。
- stdout は標準出力(fd=1)で、端末デバイスへの出力を意味します。
- stderr は標準エラー(fd=2)で、端末デバイスへのコマンドエラーメッセージの出力を意味します。

```
1 write(1, "output i=0¥n", 11) = 11
2 output i=0
```

- write: 内容を書き込む。
- 1: ファイルディスクリプタで、ここでは標準出力、つまり端末に内容を出力することを意味します。
- “output i=0¥n”: 出力する内容。
- 11: 出力する内容のバイトサイズ、スペースと改行を含めてちょうど 11 バイト。
- = 11: write 関数の戻り値で、何バイト書き込まれたかをリストします。

printf を write で代替してみることもできます。

株式会社日昇テクノロジー

```
1 cat@lubancat:~/test$ cat hello_sys.c

2

3 #include <unistd.h>

4 int main()

5 {

6 write(1, "hello, world! This is a C program.¥n", 35);

7 write(1, "output i=0¥n", 11);

8 return 0;

9 }

10

11 cat@lubancat:~/test$ gcc hello_sys.c -o hello_sys

12

13 cat@lubancat:~/test$ ./hello_sys

14 hello, world! This is a C program.

15 output i=0
```

結果は printf と同じです。

第 19 章 GPIO サブシステム

この章では、Linux GPIO サブシステムドライバー関連のアプリケーションレイヤープログラムの制御原理について説明します。

この章のサンプルコードのディレクトリは以下の通りです：base_linux/gpio/

19.1 紹介

GPIO は General Purpose I/O の略で、一般用途の入出力ポートのことを指します。簡単に言えば、

株式会社日昇テクノロジー

MCU/CPU で制御可能なピンのことで、これらのピンは通常、多くの機能を持っています。最も基本的なのは、高電位と低電位の入力検出と出力です。一部のピンは、主制御器のオンチップ外部デバイスにバインドされ、たとえば、シリアルポート、I2C、ネットワーク、電圧検出の通信ピンとして機能します。

Linux は GPIO サブシステムドライバーフレームワークを提供しており、このドライバーフレームワークを使用して CPU の GPIO ピンをユーザースペースにエクスポートできます。ユーザーは/sys ファイルシステムにアクセスして制御を行います。GPIO サブシステムは、基本的な入出力機能にピンを使用することをサポートしており、入力機能は中断検出もサポートしています。Linux カーネルのソースコードの「Documentation/gpio」ディレクトリには、GPIO サブシステムに関する説明があります。

19.1.1 GPIO デバイスディレクトリ

GPIO ドライバーサブシステムがユーザースペースにエクスポートするディレクトリは「/sys/class/gpio」です。

以下のコマンドを使用して確認できます：

```
1 # root ユーザーに切り替えて以下のコマンドを実行する必要があります
2 su
3 パスワードを入力
4
5 # GPIO をユーザースペースにエクスポート
6 echo 42 > /sys/class/gpio/export
7 # ディレクトリの変更を確認し、gpio19 ディレクトリが追加されました
8 ls /sys/class/gpio/
9 # gpio42 をユーザースペースからアンエクスポート
```

株式会社日昇テクノロジー

```
10 echo 42 > /sys/class/gpio/unexport

11 # ディレクトリの変更を確認し、gpio42 ディレクトリが消えました

12 ls /sys/class/gpio/
```

以下の画像のように：

```
root@lubancat:~# ls /sys/class/gpio
export  gpiochip0  gpiochip128  gpiochip32  gpiochip511  gpiochip64  gpiochip96  unexport
root@lubancat:~# echo 42 > /sys/class/gpio/export
root@lubancat:~# ls /sys/class/gpio
export  gpio42  gpiochip0  gpiochip128  gpiochip32  gpiochip511  gpiochip64  gpiochip96  unexport
root@lubancat:~# echo 42 > /sys/class/gpio/unexport
root@lubancat:~# ls /sys/class/gpio
export  gpiochip0  gpiochip128  gpiochip32  gpiochip511  gpiochip64  gpiochip96  unexport
root@lubancat:~#
```


このディレクトリの主な内容の説明は以下の通りです：

- export ファイル：GPIO をエクスポートします。このファイルは書き込み専用で読み取りはできません。ユーザーがこのファイルに GPIO の番号 N を書き込むと、カーネルはその番号の GPIO をユーザースペースにエクスポートするように要求します。カーネルがその GPIO を他の機能に使用していない場合、/sys/class/gpio ディレクトリには対応する番号の gpioN ディレクトリが新たに追加されます、上の画像では gpio42 がエクスポートされました。

- unexport ファイル：export の反対操作で、GPIO のエクスポートをキャンセルします。このファイルも書き込み専用で、読み取りはできません。上の画像では、unexport に 42 を書き込んだ後、gpio42 ディレクトリが消えました。

- gpiochipX ディレクトリ：このディレクトリは GPIO コントローラーデバイスを指します。

- gpioN ディレクトリ：export によってエクスポートされた具体的な GPIO ピンの制御ディレクトリです。上の画像の gpio42 ディレクトリには、そのピンを制御するための関連ファイルが含まれています。

株式会社日昇テクノロジー

19.1.2 GPIO デバイス属性

gpioN ディレクトリ下の関連デバイスファイルは、以下のコマンドで確認できます：

```
1 # ボードの端末で以下のコマンドを使用します
2 # GPIO 番号 42 をエクスポート
3 echo 42 > /sys/class/gpio/export
4
5 # gpiox のディレクトリに移動
6 cd /sys/class/gpio/gpio42
7
8 # gpio42 ディレクトリの内容を確認
9 ls -lh
```

以下の画像のように：

```
root@lubancat:/sys/class/gpio/gpio42# ls -lh
total 0
-rw-r--r-- 1 root root 4.0K Sep  8 13:53 active_low
lrwxrwxrwx 1 root root    0 Sep  8 13:53 device -> ../../../gpiochip1
-rw-r--r-- 1 root root 4.0K Sep  8 13:53 direction
-rw-r--r-- 1 root root 4.0K Sep  8 13:53 edge
drwxr-xr-x 2 root root    0 Sep  8 13:53 power
lrwxrwxrwx 1 root root    0 Sep  8 13:53 subsystem -> ../../../../../../class/gpio
-rw-r--r-- 1 root root 4.0K Sep  8 13:53 uevent
-rw-r--r-- 1 root root 4.0K Sep  8 13:53 value
root@lubancat:/sys/class/gpio/gpio42#
```

以下は、よく使われる属性ファイルの紹介です：

- direction：GPIO ピンの方向をリストします。取り得る値は以下の通りです：

    1. in：ピンが入力モードです。

    2. out：ピンが出力モードで、デフォルトの出力レベルは低です。

    3. low：ピンが出力モードで、デフォルトの出力レベルは低です。

    4. high：ピンが出力モードで、デフォルトの出力レベルは高です。

株式会社日昇テクノロジー

- value：GPIO の電位をリストします。1 は高電位、0 は低電位です。GPIO が出力モードに設定されている場合、このファイルの内容を変更することでピンの電位を変えることができます。

- edge：GPIO の割り込みトリガー方式を設定するために使用されます。GPIO が割り込みに設定されている場合、システムの poll 関数で監視することができます。edge ファイルは以下の属性値を取ることができます：

1. none：割り込みモードを使用していません。
2. rising：ピンが割り込み入力モードで、上昇エッジでトリガーされます。
3. falling：ピンが割り込み入力モードで、下降エッジでトリガーされます。
4. both：ピンが割り込み入力モードで、エッジトリガーされます。

このピンがデバイスに使用されている場合、その機能はユーザースペースで変更することはできませんが、GPIO サブシステムを使用するデバイスは、入力、出力、または割り込みモードとしてユーザースペースで柔軟に設定することができます。

19.2 ピン番号の変換

Rockchip Pin の ID は、コントローラ(bank) + ポート(port) + インデックス番号(pin)で構成されます。

- コントローラと GPIO コントローラの数は一致しています
- ポートは A、B、C、D で固定され、各ポートは 8 つのインデックス番号を持ちます (A=0,B=1,C=2,D=3)
- インデックス番号は 0、1、2、3、4、5、6、7 で固定されています

rk356x は 5 つの GPIO コントローラを持ち、各コントローラは 32 個の IO を制御できます。GPIO 機能として使用される場合、ポートの動作は GPIO コントローラレジスタによって設定されます。例えば：gpio0_xx, gpio1_xx, gpio2_xx, gpio3_xx, gpio4_xx

GPIO1_A4 は、第 1 グループのコントローラ、ポート番号が A、インデックス番号が 4 を意味します。このピン番号の計算式は 32 x 1 + 0 x 8 + 4 = 36 です。

株式会社日昇テクノロジー

すべてのピンが export ファイルを通じてユーザースペースにエクスポートできるわけではないことに注意してください。使用中のピンはエクスポートできません。

19.3 GPIO sysfs インターフェース制御

19.3.1 コマンドライン

ボード上で以下のコマンドを実行してテストします。テスト前に現在のユーザーが root であることを確認してください：

```
1 # 以下のすべての操作には管理者権限が必要です
2 # ピンを有効にする
3 echo 42 > /sys/class/gpio/export
4
5 # ピンを入力モードに設定
6 echo in > /sys/class/gpio/gpio42/direction
7 # ピンの値を読み取る
8 cat /sys/class/gpio/gpio42/value
9
10 # ピンを出力モードに設定
11 echo out > /sys/class/gpio/gpio42/direction
12 # ピンを低電位に設定
13 echo 0 > /sys/class/gpio/gpio42/value
14 # ピンを高電位に設定
15 echo 1 > /sys/class/gpio/gpio42/value
16
```

株式会社日昇テクノロジー

```
17 # ピンをリセット

18 echo 42 > /sys/class/gpio/unexport
```

以下の画像のように：

```
root@npi:/home/debian# echo 19 > /sys/class/gpio/export
root@npi:/home/debian# ls /sys/class/gpio/
export   gpiochip0    gpiochip32   gpiochip64  unexport
gpio19   gpiochip128  gpiochip504  gpiochip96
root@npi:/home/debian# echo out > /sys/class/gpio/gpio19/direction
root@npi:/home/debian# echo 1 > /sys/class/gpio/gpio19/value
root@npi:/home/debian# echo 0 > /sys/class/gpio/gpio19/value
root@npi:/home/debian#
```

コマンドの実行原理は非常にシンプルです：

- GPIO の番号を export ファイルに書き込み、GPIO デバイスをエクスポートします。

- GPIO デバイスの属性ファイル direction の値を out に変更し、GPIO を出力方向に設定します。

- GPIO デバイスの属性ファイル value の値を 1 または 0 に変更し、GPIO の高電位または低電位を制御します。

19.3.2 プログラム作成

リスト 1: base_linux/gpio/gpio_sys/gpio_sys.c

```
1 #include <string.h>

2 #include <sys/stat.h>

3 #include <unistd.h>

4 #include <fcntl.h>

5 #include <stdio.h>

6

7 #define GPIO_INDEX "42"

8 static char gpio_path[75];

9 int gpio_init(char *name)
```

株式会社日昇テクノロジー

```
10 {
11 int fd;
12 //index config
13
14 sprintf(gpio_path, "/sys/class/gpio/gpio%s", name);
15
16 if (access("gpio_path", F_OK)){
17 fd = open("/sys/class/gpio/export", O_WRONLY);
18 if(fd < 0)
19 return 1 ;
20
21 write(fd, name, strlen(name));
22 close(fd);
23
24 //direction config
25 sprintf(gpio_path, "/sys/class/gpio/gpio%s/direction", name);
26 fd = open(gpio_path, O_WRONLY);
27 if(fd < 0)
28 return 2;
29
30 write(fd, "out", strlen("out"));
31 close(fd);
```

株式会社日昇テクノロジー

```
32 }

33

34 return 0;

35 }

36

37 int gpio_deinit(char *name)

38 {

39 int fd;

40 fd = open("/sys/class/gpio/unexport", O_WRONLY);

41 if(fd < 0)

42 return 1;

43

44 write(fd, name, strlen(name));

45 close(fd);

46

47 return 0;

48 }

49

50

51 int gpio_high(char *name)

52 {
```

株式会社日昇テクノロジー

```
53 int fd;

54 sprintf(gpio_path, "/sys/class/gpio/gpio%s/value", name);

55 fd = open(gpio_path, O_WRONLY);

56 if(fd < 0){

57 printf("open %s wrong¥n",gpio_path);

58 return -1;

59 }

60

61 if(2 != write(fd, "0", sizeof("0")))

62 printf("wrong set ¥n");

63 close(fd);

64 return 0;

65 }

66

67

68 int gpio_low(char *name)

69 {

70 int fd;

71 sprintf(gpio_path, "/sys/class/gpio/gpio%s/value", name);

72 fd = open(gpio_path, O_WRONLY);

73 if(fd < 0){
```

株式会社日昇テクノロジー

```
74 printf("open %s wrong¥n",gpio_path);

75 return -1;

76 }

77

78 if(2 != write(fd, "1", sizeof("1")))

79 printf("wrong set ¥n");

80 close(fd);

81 return 0;

82 }

83

84 int main(int argc, char *argv[])

85 {

86 char buf[10];

87 int res;

88

89 /* 校验传参*/

90 if (2 != argc) {

91 printf( "usage: %s <PinNum>¥n",argv[0]);

92 return -1;

93 }

94 res = gpio_init(argv[1]);
```

株式会社日昇テクノロジー

```
95 if(res){
96 printf("gpio init error,code = %d",res);
97 return 0;
98 }
99
100 while(1){
101 printf("Please input the value : 0--low 1--high q--exit¥n");
102 scanf("%10s", buf);
103
104 switch (buf[0]){
105 case '0':
106 gpio_low(argv[1]);
107 break;
108
109 case '1':
110 gpio_high(argv[1]);
111 break;
112
113 case 'q':
114 gpio_deinit(argv[1]);
115 printf("Exit¥n");
```

株式会社日昇テクノロジー

```
116 return 0;

117

118 default:

119 break;

120 }

121 }

122 return 0;

123 }
```

このコードの説明は以下の通りです：

- gpio_init 関数：open、write、close などの関数を使用して export および gpioN/direction ファイルを変更し、GPIO を使用するピンを出力モードで初期化します。

- bgpio_deinit 関数：unexport ファイルに番号を書き込み、エクスポートをキャンセルします。

- gpio_high および gpio_low 関数：gpioN/value ファイルに 1 と 0 を書き込み、ピンの高電位と低電位を制御します。

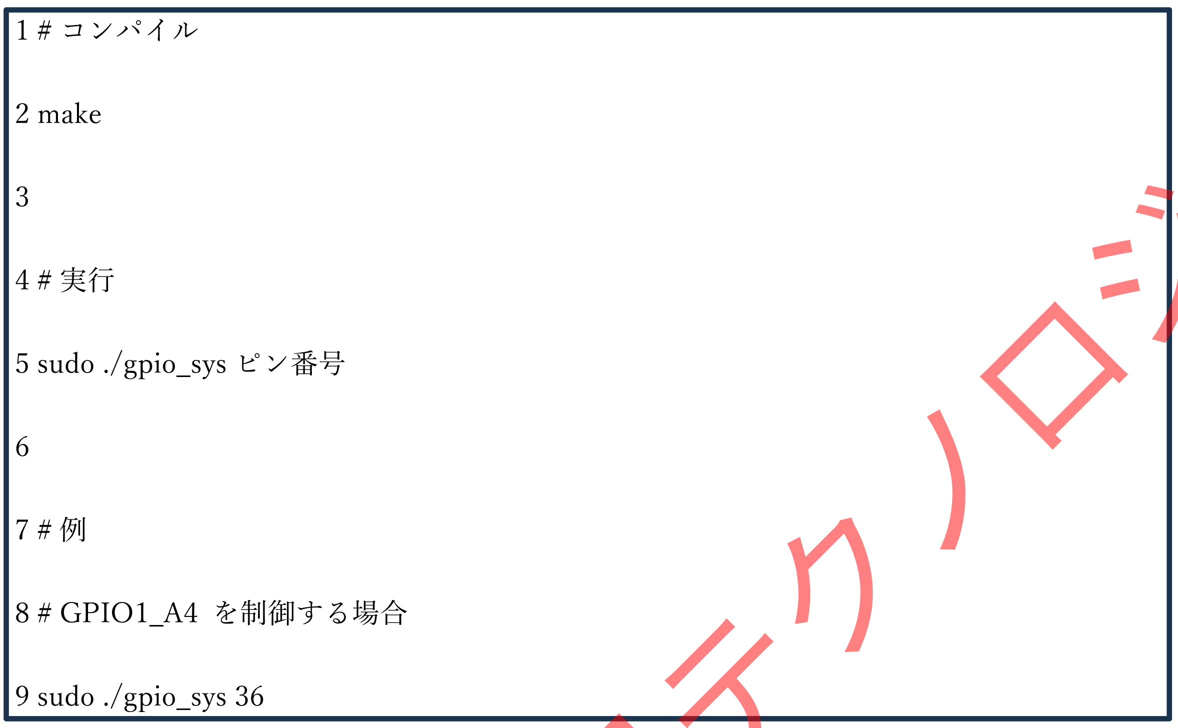
- scanf はユーザー入力を検出し、ユーザー入力に基づいて対応する関数を呼び出して GPIO を制御します。

このコードで特に注意すべき点は、export および unexport ファイルは書き込み専用であるため、open で開く際には「O_WRONLY」フラグを使用して書き込みモードで開く必要があり、「O_RDWR」などの読み取りモードを含むフラグは使用できないことです。

株式会社日昇テクノロジー

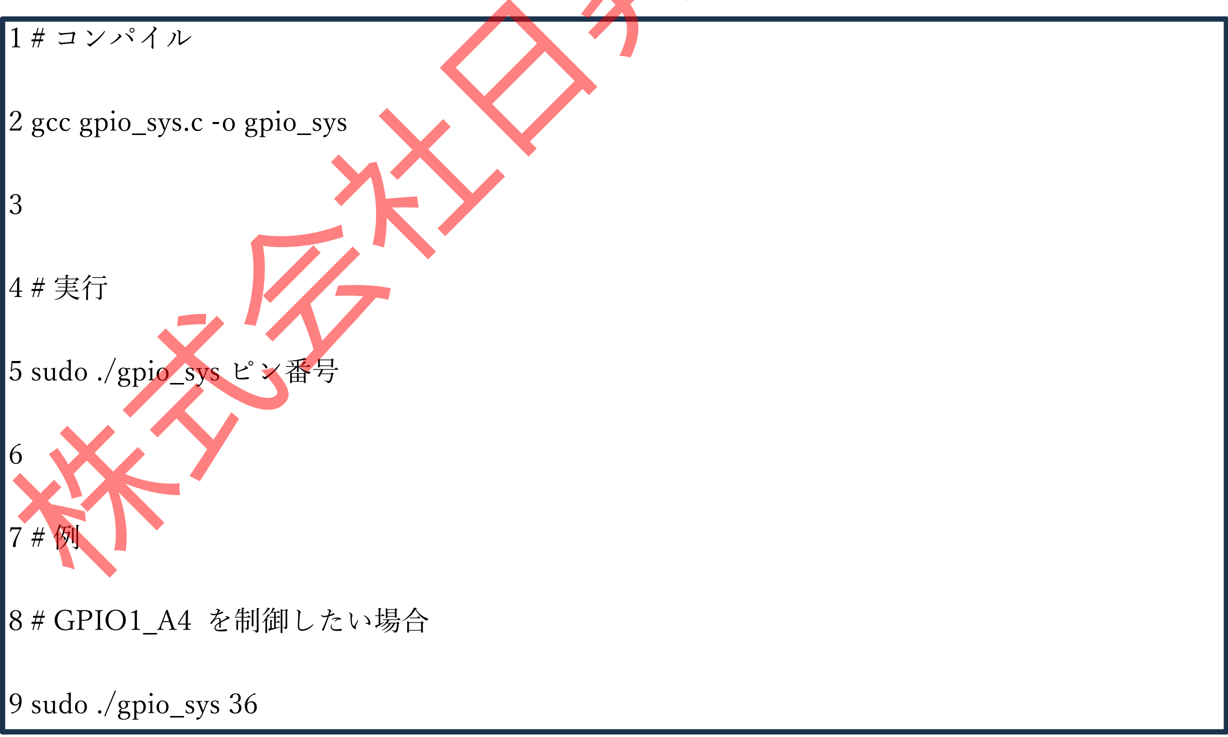
19.3.3 コンパイル＆実行

方法 1：



```
1 # コンパイル
2 make
3
4 # 実行
5 sudo ./gpio_sys ピン番号
6
7 # 例
8 # GPIO1_A4 を制御する場合
9 sudo ./gpio_sys 36
```

方法 2：



```
1 # コンパイル
2 gcc gpio_sys.c -o gpio_sys
3
4 # 実行
5 sudo ./gpio_sys ピン番号
6
7 # 例
8 # GPIO1_A4 を制御したい場合
9 sudo ./gpio_sys 36
```

株式会社日昇テクノロジー

以下の画像のように：

```
root@lubancat:/home/cat/all_test/gpio# gcc gpio_sys.c -o gpio_sys
root@lubancat:/home/cat/all_test/gpio# ./gpio_sys
This is the gpio demo
Please input the value : 0--low 1--high q--exit
1
Please input the value : 0--low 1--high q--exit
0
Please input the value : 0--low 1--high q--exit
1
Please input the value : 0--low 1--high q--exit
q
Exit
root@lubancat:/home/cat/all_test/gpio#
```

プログラム実行後、端末に 1 を入力してエンターを押すと GPIO が高電位になり、0 を入力してエンターを押すと GPIO が低電位になります。

19.4 libgpiod を使った IO 制御

libgpiod はキャラクタデバイスインターフェースで、GPIO アクセス制御は、たとえば/dev/gpiodchip0 のようなキャラクタデバイスファイルを操作することで実現されます。そして libgpiod はいくつかのコマンドツール、C ライブラリ、Python ラッパーを提供します。

libgpiod を使用するには、ボード上に libgpiod ライブラリをインストールする必要があります。

```
1 # libgpiod ライブラリ及びヘッダーファイルのインストール
2 sudo apt install libgpiod-dev
3
4 # gpiod コマンドラインツールのインストール
5 sudo apt install gpiod
```

19.4.1 コマンドライン制御

一般的なコマンドラインは以下の通りで、-h を使用してコマンドに対応する使用説明を見ることができます（GPIO1_A4 を例として）。

リスト 1: libgpiod コマンド

株式会社日昇テクノロジー

コマンド	機能	使用例	説明
gpiodetect	全ての GPIO コントローラをリストアップ	gpiodetect（引数なし）	全ての GPIO コントローラをリストアップ
gpioinfo	GPIO コントローラのピン情報をリストアップ	gpioinfo 1	第 1 のコントローラのピン情報をリストアップ
gpioset	GPIO を設定する	gpioset 1 4=0	第 1 のコントローラのピン番号 4 をローレベルに設定
gpioget	GPIO ピンの状態を取得する	gpioget 1 4	第 1 のコントローラのピン番号 4 の状態を取得
gpiomon	GPIO の状態を監視する	gpiomon 1 4	第 1 のコントローラのピン番号 4 の状態を監視

重要: Rockchip Pin の ID は、コントローラ(bank) + ポート(port) + インデックス番号(pin)で構成されます。ポート番号とインデックス番号は gpiod に渡す際に一つの数値に統合されますが、すべてのピンが libgpiod で制御できるわけではありません。たとえば、LED などのすでに使用されているピンなどです。

19.4.2 libgpiod プログラミング

libgpiod-dev がボードにインストールされた後、以下のコマンドで具体的なヘッダーファイルとライブラリファイルを見つけることができます：

```
1 # ボード上で libgpiod ライブラリを探す
2 dpkg -L libgpiod-dev
3 # 以下が出力です
4 dpkg -L libgpiod-dev
5 /.
6 /usr
```

株式会社日昇テクノロジー

```
7 /usr/include

8 /usr/include/gpiod.h

9 /usr/include/gpiod.hpp

10 /usr/lib

11 /usr/lib/aarch64-linux-gnu

12 /usr/lib/aarch64-linux-gnu/libgpiod.a

13 /usr/lib/aarch64-linux-gnu/libgpiodcxx.a

14 /usr/lib/aarch64-linux-gnu/pkgconfig

15 /usr/lib/aarch64-linux-gnu/pkgconfig/libgpiod.pc

16 /usr/lib/aarch64-linux-gnu/pkgconfig/libgpiodcxx.pc

17 /usr/share

18 /usr/share/doc

19 /usr/share/doc/libgpiod-dev

20 /usr/share/doc/libgpiod-dev/README.gz

21 /usr/share/doc/libgpiod-dev/changelog.Debian.gz

22 /usr/share/doc/libgpiod-dev/copyright

23 /usr/lib/aarch64-linux-gnu/libgpiod.so

24 /usr/lib/aarch64-linux-gnu/libgpiodcxx.so

25 /usr/include/gpiod.h

26 /usr/lib/arm-linux-gnueabihf/libgpiod.a

27 /usr/lib/arm-linux-gnueabihf/libgpiod.so
```

株式会社日昇テクノロジー

検索結果に含まれる gpiod.h、libgpiod.so、libgpiod.a は、Debian 10 buster で apt によってデフォルトでインストールされるバージョンのヘッダーファイル、動的リンクライブラリ、静的リンクライブラリです。

一般的な libgpiod API（C ライブラリ）は以下の通りです：

リスト 2: Source/libgpio_beep/libgpiod/gpiod.h ファイル

```
1 //メンバ変数
2 struct gpiod_chip; //GPIO チップハンドル
3 struct gpiod_line; //GPIO ピンハンドル
4
5 //GPIO コントローラ（GPIO チップ）の取得
6 struct gpiod_chip *gpiod_chip_open(const char *path);
7
8 //GPIO ピンの取得
9 struct gpiod_line *gpiod_chip_get_line(struct gpiod_chip *chip, unsigned int offset);
10
11 //ピンの方向を入力モードに設定
12 int gpiod_line_request_input(struct gpiod_line *line, const char *consumer);
13
14 //ピンを出力モードに設定
15 int gpiod_line_request_output(struct gpiod_line *line, const char *consumer, int default_val);
16
17 //ピンの高低電位の設定
18 int gpiod_line_set_value(struct gpiod_line *line, int value);
19
```

株式会社日昇テクノロジー

```
20 //ピンの状態読み取り

21 int gpiod_line_get_value(struct gpiod_line *line);

22

23 //GPIO ピンの解放

24 void gpiod_line_release(struct gpiod_line *line);

25

26 //GPIO チップハンドルを閉じて、割り当てられたすべてのリソースを解放。

27 void gpiod_chip_close(struct gpiod_chip *chip);
```

19.4.2.1 プログラム作成

リスト 3: base_linux/gpio/gpio_lib/gpio_lib.c

```
1 #include <gpiod.h>

2 #include <stdio.h>

3 #include <unistd.h>

4 #include <stdlib.h>

5

6 int main(int argc, char **argv)

7 {

8 int i;

9 int ret;

10 char buf[10];

11 struct gpiod_chip *chip; //GPIO コントローラハンドル

12 struct gpiod_line *line; //GPIO ピンハンドル
```

株式会社日昇テクノロジー

```
13
14 /* GPIO コントローラの取得 */
15 chip = gpiod_chip_open("/dev/gpiochip1");
16 if (chip == NULL) {
17 printf("gpiod_chip_open error¥n");
18 return -1;
19 }
20
21 /* GPIO ピンの取得 */
22 line = gpiod_chip_get_line(chip, 8);
23 if (line == NULL) {
24 printf("gpiod_chip_get_line error¥n");
25 goto release_line;
26 }
27
28 /* GPIO を出力モードに設定 */
29 ret = gpiod_line_request_output(line, "led", 0);
30 if (ret < 0) {
31 printf("gpiod_line_request_output エラー¥n");
32 goto release_chip;
33 }
34
35 for(i = 0; i < 10; i++)
```

株式会社日昇テクノロジー

```
36 {
37 gpiod_line_set_value(line, 1);
38 usleep(500000); // 0.5 秒遅延
39 gpiod_line_set_value(line, 0);
40 usleep(500000);
41 }
42
43 release_line:
44 /* GPIO ピンの解放 */
45 gpiod_line_release(line);
46 release_chip:
47 /* GPIO コントローラの解放 */
48 gpiod_chip_close(chip);
49 return 0;
50 }
```

19.4.2.2 コンパイル＆実行

実行前に、制御したいピンに応じてコードを変更することができます。

ここでは GPIO1_A4 の制御を例とします。

```
1 chip = gpiod_chip_open("/dev/gpiochip1");
```

この関数は GPIO のコントローラを特定するために使います。gpiochip0, 1, 2, 3, 4

```
1 line = gpiod_chip_get_line(chip, 4);
```

株式会社日昇テクノロジー

- ポートは A、B、C、D で固定され、各ポートは 8 つのインデックス番号を持ちます。(A=0, B=1, C=2, D=3)

- インデックス番号は 0、1、2、3、4、5、6、7 で固定されています。

計算方法：ポート*8 + インデックス番号、例：0*8+4=4

方法 1：

```
1 # コンパイル
2 make
3
4 # 実行
5 sudo ./gpio_lib
```

現象：GPIO に外部接続されたランプが点滅し、10 回点滅した後に消灯します。

方法 2：

```
1 # コンパイル、外部ライブラリを使用
2 gcc gpio_lib.c -o gpio_lib `pkg-config --cflags libgpiod` `pkg-config --libs libgpiod`
3
4 # 実行
5 sudo ./gpio_lib
```

現象：GPIO に外部接続されたランプが点滅し、10 回点滅した後に消灯します。

株式会社日昇テクノロジー

19.5 system 関数を使ったプログラミング

リスト 4: base_linux/gpio/gpio_system/gpio_system.c

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(void)
7 {
8 pid_t result;
9 int i;
10 for(i=0; i<10; i++){
11 /* system()関数の呼び出し */
12 result = system("gpioset 1 8=1");
13 usleep(500000);
14 result = system("gpioset 1 8=0");
15 usleep(500000);
16 }
17 return result;
18 }
```

このコードの原理は、C プログラム内で「system()」を使うことは、シェル端末でコマンドを使うのと同じです。root 権限での実行が必要です。

欠点：この方法で IO を制御すると、カーネルのコンテキストスイッチが発生し、カーネルの処理に影響

株式会社日昇テクノロジー

を与えます。そのため、短時間に GPIO を多次操作する場合には推奨されません。GPIO に多次操作が必要な場合は、gpio_lib.c を使用した方が速く、効率的です。

現象: 上記のプログラムと同様です。

第 20 章 input サブシステム

この章のサンプルコードのディレクトリは以下の通りです：base_linux/ev_test

20.1 input サブシステム

input サブシステムは、Linux が入力デバイスに提供する統一的なドライバーフレームワークです。ボタン、キーボード、タッチスクリーン、マウスなどの入力デバイスのドライバーは似ています。キー操作やタッチ操作が行われると、ハードウェアが割り込みを生成し、その後 CPU が直接ピンの電位を読み取るか、SPI、I2C などの通信方式を介してデバイスのレジスタから具体的なキー値やタッチ座標を読み取り、これらの情報をカーネルに提出します。input サブシステムドライバーを使用する入力デバイスは、統一されたデータ構造を介してカーネルに情報を提出することができ、このデータ構造には入力の時間、タイプ、コード、具体的なキー値や座標が含まれています。内部では、/dev/input ディレクトリ下のファイルインターフェースを通じてユーザースペースに情報が伝達されます。

Linux カーネルソースコードの「Documentation/input」ディレクトリには、input サブシステムに関する説明が含まれています。

ボードのデフォルトのファクトリーイメージでは、ボタン、タッチスクリーン、マウス、キーボードがすべて input サブシステムドライバーを使用しています。この章では、ボタンを使用して説明します。

20.2 input イベントディレクトリ

20.2.1 evtest ツールを使ったテスト

input サブシステムドライバーの開発時には、よく evtest ツールをテストに使用します。ここでは、このツールを使ってボード上の入力デバイスを理解します。

株式会社日昇テクノロジー

Ubuntu ホストで以下のコマンドを使用してテストします：

1 sudo evtest

2 # ホストの出力に基づいて、テストするデバイスを選択します。下の画像では「7」のマウスを選択しています

3 # 選択したデバイスに応じてテストします。キーボードを選択した場合はキーボードを押し、マウスを選択した場合はマウスを動かします

以下の画像のように：

```
^Croot@lubancat:/home/cat/all_test/gpio# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      fdd70030.pwm
/dev/input/event1:      rk805 pwrkey
/dev/input/event2:      adc-keys
/dev/input/event3:      rk-headset
/dev/input/event4:      USB Keyboard
/dev/input/event5:      USB Keyboard System Control
/dev/input/event6:      USB Keyboard Consumer Control
/dev/input/event7:      A4Tech USB Mouse Mouse
/dev/input/event8:      A4Tech USB Mouse
Select the device event number [0-8]: 7
Input driver version is 1.0.1
Input device ID: bus 0x3 vendor 0x9da product 0xc10a version 0x110
Input device name: "A4Tech USB
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 272 (BTN_LEFT)
    Event code 273 (BTN_RIGHT)
    Event code 274 (BTN_MIDDLE)
    Event code 275 (BTN_SIDE)
    Event code 276 (BTN_EXTRA)
    Event code 277 (BTN_FORWARD)
    Event code 278 (BTN_BACK)
    Event code 279 (BTN_TASK)
  Event type 2 (EV_REL)
    Event code 0 (REL_X)
    Event code 1 (REL_Y)
    Event code 8 (REL_WHEEL)
  Event type 4 (EV_MSC)
    Event code 4 (MSC_SCAN)
Properties:
Testing ... (interrupt to exit)
Event: time 1662625593.654109, type 2 (EV_REL), code 0 (REL_X), value -1
Event: time 1662625593.654109, -------------- SYN_REPORT ------------
Event: time 1662625599.134273, type 2 (EV_REL), code 0 (REL_X), value -1
Event: time 1662625599.134273, type 2 (EV_REL), code 1 (REL_Y), value 1
Event: time 1662625599.134273, -------------- SYN_REPORT ------------
Event: time 1662625599.142195, type 2 (EV_REL), code 0 (REL_X), value -3
Event: time 1662625599.142195, type 2 (EV_REL), code 1 (REL_Y), value 4
Event: time 1662625599.142195, -------------- SYN_REPORT ------------
Event: time 1662625599.150210, type 2 (EV_REL), code 0 (REL_X), value -3
Event: time 1662625599.150210, type 2 (EV_REL), code 1 (REL_Y), value 1
Event: time 1662625599.150210, -------------- SYN_REPORT ------------
Event: time 1662625599.158211, type 2 (EV_REL), code 0 (REL_X), value -3
Event: time 1662625599.158211, type 2 (EV_REL), code 1 (REL_Y), value 3
```

USB キーボー

USB マウス

マウスを観測対象として選択

同期信号

サポートされているボタンのタイプ

マウスが伝えることができる情報

サポートされている相対座標タイプ x、y

出力される値

株式会社日昇テクノロジー

上記の実行プロセスの説明は以下の通りです：

- evtest ツールを実行すると、システムで現在利用可能な/dev/input/event0~8 の入力イベントファイルが列挙され、これらのイベントに対応するデバイス名が表示されます。

- 「A4Tech USB Mouse」がボードに接続されたマウスであると推測できるため、対応する event7 のイベント番号 7 を入力しました。実験時は、自分のボードの出力に基づいて選択してください。

- 番号を入力した後、event7 のデバイス情報が表示され、ドライバーバージョン、デバイス ID、デバイス名、サポートされているイベントタイプ、イベントコード、入力値の範囲が含まれています。

- この時点でマウスを動かすと、詳細なイベント情報が出力されます。動かしても何も出力されない場合は、マウスデバイスを選択していない可能性があるため、終了して再選択してください。出力情報の各行には、マウスのイベント報告の具体的な時間 time、イベントタイプ type 2（EV_REL）、イベントコード code1 または code0（REL_Y または REL_X）と具体的な値 value が含まれています。この値はマウスの X/Y 座標です。

20.2.2 input イベント構造

evtest ツールの原理は神秘的なものではありません。この章を学習した後、自分でコードを書いてその一部の機能を実装することもできます。使用可能なイベントを列挙する時、それは「/dev/input/」ディレクトリを見ることによって実現されます。この例では、ホストの「/dev/input」ディレクトリの内容は以下の画像のように表示されます。

```
root@lubancat:~# ls /dev/input/
by-id  by-path  event0  event1  event2  event3  event4  event5  event6  event7  event8
root@lubancat:~#
```

「/dev/input」ディレクトリ下には、各種の event デバイスがユーザースペースへのアクセスインターフェースファイルとして露出しています。これらのファイルの内容を読むことで、デバイスが報告する情報を取得できます。

GPIO サブシステムでは、direction などのデバイスファイルは具体的な情報を文字列で直接記録してい

株式会社日昇テクノロジー

るため、cat コマンドでファイルの内容を出力する時、文字列の形式は読むのに非常に便利です。しかし、event ファイルに含まれる情報は多く、文字列を使うと他のプログラムが処理しにくいため、純粋なカーネルイベントデータ構造を使用して内容を記録しています。他のプログラムが使用する際は、読み取った内容をデータの構造に従ってフォーマット変換する必要があります。このデータ構造は以下のように定義されています。

リスト 1: input_event 構造体（カーネルソースコードの/include/uapi/linux/input.h ファイル）

```
1 struct input_event {
2 struct timeval time;
3 __u16 type;
4 __u16 code;
5 __s32 value;
6 };
```

- time：イベントが生成されたタイムスタンプを記録する変数です。evtest の出力の time 値です。

- type：入力デバイスのイベントタイプ。システムで一般的に使用されるデフォルトタイプには EV_KEY、EV_REL、EV_ABS があり、それぞれキーの状態変更イベント、相対座標変更イベント、絶対座標変更イベントをリストします。特に、EV_SYN はイベントを区切るために使用され、特別な意味はありません。マウスを選択する場合（この章の最初の画像）、evtest の出力タイプは EV_ABS です。

- code：イベントコードで、イベントをより正確にリストします。例えば、EV_KEY イベントタイプでは、code の値はキーボード上の具体的なキーをリストすることが多く、その範囲は 0~127 の間です。例えば、キー Q は KEY_Q に対応し、その列挙値は 16 です。マウスを選択する場合、evtest の出力に含まれる code には ABS_X/ABS_Y があり、X または Y 座標が報告されていることを示します。

- value：イベントの値。EV_KEY イベントタイプの場合、キーが押された時はこの値が 1、キーが離

株式会社日昇テクノロジー

された時はこの値が 0 になります。マウスを選択する場合、evtest の出力に含まれる ABS_X イベントタイプの value 値は X 座標をリストし、ABS_Y タイプの value 値は Y 座標をリストします。

同じく cat コマンドでイベントファイルを見ると、イベントが発生すると、cat は内容を文字列に変換して表示し、乱码が見えるため、この方法ではデバイスがイベントを報告しているかを簡単に確認することができます。

以下の方法でテストすることができます：

```
1 # 自分のホスト上の具体的なイベントファイルを見るためにイベント番号を変更してください
2 # ここでは event6 は本ホストのマウスデバイスですが、sudo 権限を使用する必要があります
3 sudo cat /dev/input/event6
4 # コマンドを入力した後にマウスを動かすと、文字が表示されます
```

以下の画像のように：

他のファイルと異なり、通常 cat コマンドでファイルの内容を読み取るとすぐに戻るのに対し、この場合は event ファイルを読み取ると、コマンドは入力を待ち続けます。

```
1 # 自分のホストのイベント番号に基づいて、特定のイベントファイルを確認するための変更
2 # ここで使用される event6 は、このホストのマウスデバイスです。sudo 権限の使用に注意してください
3 sudo cat /dev/input/event6
4 # コマンドを入力した後、マウスを動かすと、文字が表示されます
```

株式会社日昇テクノロジー

5

6

7 同様に、hexdump コマンドを使用して出力されたコマンドを確認することもできます

8

9 hexdump /dev/input/

10

11 バイナリデータは、ドライバーが異なると効果的に分析するのが難しい場合があります

20.2.3 input イベントデバイス名

"/dev/input/event*"のイベント番号とデバイスの関連付けは固定されておらず、通常はシステムがデバイスを検出した順序に応じてイベントファイルの番号が割り当てられます。これはアプリケーションの制御には不便ですが、"/dev/input/by-id"や"/dev/input/by-path"ディレクトリを通じて具体的なハードウェアデバイスを確認することができます。下の画像では

株式会社日昇テクノロジー

```
root@lubancat:/dev/input# ls -lh by-path/
total 0
lrwxrwxrwx 1 root root 9 Sep  8 09:50 platform-adc-keys-event -> ../event2
lrwxrwxrwx 1 root root 9 Sep  8 16:15 platform-fd840000.usb-usb-0:1:1.0-event-kbd -> ../event4
lrwxrwxrwx 1 root root 9 Sep  8 16:15 platform-fd840000.usb-usb-0:1:1.1-event -> ../event5
lrwxrwxrwx 1 root root 9 Sep  8 09:50 platform-fdd40000.i2c-platform-rk805-pwrkey-event -> ../event1
lrwxrwxrwx 1 root root 9 Sep  8 09:50 platform-fdd70030.pwm-event -> ../event0
lrwxrwxrwx 1 root root 9 Sep  8 09:50 platform-rk-headset-event -> ../event3
lrwxrwxrwx 1 root root 9 Sep  8 16:19 platform-xhci-hcd.4.auto-usb-0:1:1.0-event -> ../event8
lrwxrwxrwx 1 root root 9 Sep  8 16:19 platform-xhci-hcd.4.auto-usb-0:1:1.0-event-mouse -> ../event7
root@lubancat:/dev/input#
```

by-path ディレクトリ下の内容がリストアップされています。このディレクトリ下のファイルは実際にはリンクで、たとえば第一行の platform-xhci-hcd.4.auto-usb-0:1:1.0-event-mouse -> ../event7 は"platform-xhci-hcd.4.auto-usb-0:1:1.0-event-mouse"ファイルが event7 のショートカットであることを示しており、これは本ホストで使用されているマウスであることを意味します。つまり、このファイルにアクセスすることで、そのマウスのイベントデバイスにアクセスでき、このファイル名はハードウェアとの関連が固定されています。これからの実験ではこの方法を使用します。

```
root@lubancat:/sys/class/input# ls
event0  event1  event2  event3  event4  event5  event6  event7  event8  input0  input1  input2  input3  input4  input5  input6  input7  input8
root@lubancat:/sys/class/input# ls event7
dev  device  power  subsystem  uevent
root@lubancat:/sys/class/input# ls event7/device
capabilities  device  event7  id  modalias  name  phys  power  properties  subsystem  uevent  uniq
root@lubancat:/sys/class/input# cat event7/device/name
A4Tech USB Mouse Mouse
root@lubancat:/sys/class/input#
```

"/dev"下のデバイスは"/sys"を通じてエクスポートされるため、"/sys/class/input"ディレクトリを通じて確認することもできます。"/sys/class/input"下には各イベントに名前付けられたディレクトリが含まれており、そのディレクトリ下の device/name ファイルにはイベントに対応するデバイス名が含まれています。たとえば、この例では"/sys/class/input/event7/device/name"ファイルの内容が"platform-xhci-hcd.4.auto-usb-0:1:1.0-event-mouse"であり、evtest ツールがリストアップしたイベントとデバイス名の関係はここから読み取られます。

20.3 ボードキー検出実験

20.3.1 実験コード分析

入力イベントの検出アプリケーションでは、通常メインスレッドを使って"/dev/input/event*"デバイスフ

株式会社日昇テクノロジー

ァイルを直接ループ読み取り、イベントのデータ構造を取得し、その後メッセージキューを通じて他のサブスレッドに通知し、入力操作に応答します。

リスト 2: base_linux/ev_test/ev_test.c

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h>
7 #include <linux/input.h>
8 #include <linux/input-event-codes.h>
9
10 //実際の状況に応じて変更してください
11 const char default_path[] = "/dev/input/by-path/platform-xhci-hcd.4.auto→usb-0:1:1.0-event-mouse";
12
13 int main(int argc, char *argv[])
14 {
15 int fd;
16 struct input_event event;
17 char *path;
```

株式会社日昇テクノロジー

```
18
19 printf("This is a input device demo.¥n");
20
21 //入力パラメータがない場合は、デフォルトのイベントデバイスを使用します
22 if(argc > 1)
23 path = argv[1];
24 else
25 path = (char *)default_path;
26
27 fd = open(path, O_RDONLY);
28 if(fd < 0){
29 printf("Fail to open device:%s.¥n"
30 "Please confirm the path or you have permission to do this.¥
,→n", path);
31 exit(1);
32 }
33
34 printf("Test device: %s.¥nWaiting for input...¥n", path);
35
36 while(1){
37 if(read(fd, &event, sizeof(event)) == sizeof(event)){
38
39 // EV_SYN はイベント区切りフラグで、プリントしません
```

株式会社日昇テクノロジー

```
40 if(event.type != EV_SYN)

41 printf("Event: time %ld.%ld, type %d, code %d,value %d¥n",

42 event.time.tv_sec,event.time.tv_usec,

43 event.type,
```

```
44 event.code,

45 event.value);

46 }

47 }

48 close(fd);

49

50 return 0;

51 }
```

株式会社日昇テクノロジー

このコードの説明は以下の通りです：

- 第 11 行：デフォルトのデバイスパスを定義しています。ここでは、"/dev/input/by-path"下のマウスデバイスのリンクファイル名を使用しています。これは"/dev/input/event*"を使用しない理由は、他の入力デバイスによってイベント番号が影響を受けないようにするためです。

- 第 22～25 行：main 関数の入力パラメータをチェックし、プログラム実行時に入力パラメータがある場合は、最初の入力パラメータを開くべきイベントデバイスファイルのパスとして使用します。プログラムにパラメータがない場合は、上記のデフォルトのデバイス、マウスデバイスのイベントデバイスファイルを使用します。

- 第 27～32 行：イベントデバイスファイルを O_RDONLY モードで開きます。O_RDONLY モードはデフォルトでブロッキングモードであり、イベントデバイスファイルはブロッキング操作をサポー

トしています。つまり、後ほど read 関数を使用して読み取りを行う際、イベントが報告されるまで待機し、読み取りが成功するか失敗するまで戻りません。

- 第 36 行：while ループ内で read システムコールを使用してイベントファイルを読み取り、読み取った内容を"struct input_event"型の event 変数に格納します。"struct input_event"型は、前に紹介したカーネルイベントデータ構造です。読み取りに成功した場合、この変数の構造体メンバを通じてイベントのタイムスタンプ、タイプ、コード、値にアクセスできます。

- 第 41～45 行：読み取った event 変数の各メンバの値を出力します。報告されたイベントには、多くの EV_SYN タイプのイベントが含まれていますが、このタイプのイベントは区切り用であり、特に意味はありませんので、コードではこのタイプのイベントの内容を出力しません。

イベントが報告されない場合、第 36 行の read イベントデバイスファイルの読み取り操作はブロックされます。簡単に言うと、第 41 行の printf コードをコメントアウトしない限り、ループ内で継続的に出力されることはなく、イベントが発生して read が終了した後にのみ printf 関数が一度実行され、再び read イベントで再びブロックされます。このブロッキングプロセス中、プロセスは休眠し、CPU の使用を解放します。

GPIO サブシステムフレームワークを使用してキーボードドライバプログラムを記述する場合、アプリケーションレイヤでの操作では、"/sys/class/gpio/gpio*/direction"ファイルを入力方向に設定し、その後"/sys/class/gpio/gpio*/value"ファイルの値をループ読み取ってキーの状態を取得する必要がありますが、value ファイルの read 読み取り操作はブロックされないため、プロセスはファイルの内容を読み取ってキー値を判断するために CPU の貴重な計算リソースを継続的に使用します。

read イベントファイル操作がブロックされるため、この方法では 2 つの入力デバイスを同時に検出することはできません。このような場合は、select や poll などの IO 多重化操作を使用して目的を達成することができますが、これについては後の章で説明します。

株式会社日昇テクノロジー

20.3.2 コンパイル

方法 1：

```
1 # コンパイル

2 make
```

方法 2：

```
1 # コンパイル

2 gcc ev_test.c -o ev_test
```

20.3.3 実行

```
1 # 実行

2 sudo ./ev_test
```

以下の画像のように：

```
root@lubancat:/home/cat/all_test/ev_key# ./ev_test
This is a input device demo.
Test device: /dev/input/by-path/platform-xhci-hcd.4.auto-usb-0:1:1.0-event-mouse.
Waiting for input...
Event: time 1662629182.28632, type 2, code 1,value 2
Event: time 1662629182.36516, type 2, code 0,value -3
Event: time 1662629182.36516, type 2, code 1,value 5
Event: time 1662629182.44537, type 2, code 0,value -4
Event: time 1662629182.52510, type 2, code 0,value -4
Event: time 1662629182.52510, type 2, code 1,value 2
Event: time 1662629182.60509, type 2, code 0,value -4
Event: time 1662629182.68517, type 2, code 0,value -2
Event: time 1662629182.68517, type 2, code 1,value 2
Event: time 1662629182.84541, type 2, code 0,value -2
Event: time 1662629182.100540, type 2, code 0,value -3
Event: time 1662629182.100540, type 2, code 1,value 1
Event: time 1662629182.108505, type 2, code 0,value -3
Event: time 1662629182.108505, type 2, code 1,value 3
Event: time 1662629182.116543, type 2, code 0,value -5
Event: time 1662629182.116543, type 2, code 1,value 1
Event: time 1662629182.124561, type 2, code 0,value -4
Event: time 1662629182.124561, type 2, code 1,value 2
Event: time 1662629182.132545, type 2, code 0,value -3
Event: time 1662629182.132545, type 2, code 1,value 3
Event: time 1662629182.140573, type 2, code 0,value -5
Event: time 1662629182.140573, type 2, code 1,value 2
Event: time 1662629182.148555, type 2, code 0,value -2
```

株式会社日昇テクノロジー

第 21 章 シリアル通信

この章では、LubanCat-RK のボードの 40 ピンコネクタにおけるシリアルの基本的な使用方法について説明します。

この章のサンプルコードのディレクトリは以下の通りです：base_linux/uart

21.1 シリアルピンの関係

シリアルのピン関係は以下のリストのようになります。

シリアル	ピン	機能
TXD	8	送信信号線
RXD	10	受信信号線

LUBANCAT RK PIN 機能			
機能	PIN		機能
3.3V	1	2	5V
I2C3_SDA	3	4	5V
I2C3_SCL	5	6	GND
GPIO	7	8	UART_TX
GND	9	10	UART_RX
GPIO	11	12	PWM
GPIO	13	14	GND
GPIO	15	16	GPIO
3.3V	17	18	GPIO
MOSI	19	20	GND
MISO	21	22	GPIO
SCLK	23	24	CSO
GND	25	26	CSI
I2C5_SDA	27	28	I2C5_SCL
GPIO	29	30	GND
GPIO	31	32	PWM
PWM	33	34	GND
PWM	35	36	GPIO
GPIO	37	38	GPIO
GND	39	40	GPIO

株式会社日昇テクノロジー

21.2 シリアルインターフェイスの有効化

シリアルはデフォルトでオフになっているため、使用するには有効化する必要があります。

/boot/uEnv/board.txt（board は使用しているボードの名前）を開いて、uart 関連のデバイストリープラグインが有効になっているかを確認できます。

ファイルを編集し、uart（例えば uart3）を含む行のコメントアウトを解除します。下の画像のように：

```
uname_r=4.19.232
size=0x1000000
#dtb=rk3566-lubancat1.dtb

enable_uboot_overlays=1
#overlay_start

#dtoverlay=/dtb/overlay/lubancat-i2c3-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-i2c5-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm10-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm14-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm8-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm9-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-spi3-m1-gpio-cs-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-spi3-m1-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-uart3-m1-overlay.dtbo

#overlay_end

~
~
                                                  19,0-1        All
```

その後、デバイスを再起動して有効にします。

注意: 直接電源を抜いて再起動する場合、ファイルが変更されない可能性があります（理由：ファイルがメモリからストレージデバイスに同期されなかったため。解決方法は、ターミナルで「sync」を入力してから電源を切ることです）。

21.3 シリアルデバイスの確認

シリアルが正常に有効になったかを確認します。

```
1 # 端末デバイスを確認するコマンドを実行
2 ls /dev/tty*
```

下の画像のように：

株式会社日昇テクノロジー

```
cat@lubancat:~$ ls /dev/tty*
/dev/tty    /dev/tty16  /dev/tty24  /dev/tty32  /dev/tty40  /dev/tty49  /dev/tty57  /dev/tty8
/dev/tty0   /dev/tty17  /dev/tty25  /dev/tty33  /dev/tty41  /dev/tty5   /dev/tty58  /dev/tty9
/dev/tty1   /dev/tty18  /dev/tty26  /dev/tty34  /dev/tty42  /dev/tty50  /dev/tty59  /dev/ttyFIQ0
/dev/tty10  /dev/tty19  /dev/tty27  /dev/tty35  /dev/tty43  /dev/tty51  /dev/tty6   /dev/ttyS3
/dev/tty11  /dev/tty2   /dev/tty28  /dev/tty36  /dev/tty44  /dev/tty52  /dev/tty60
/dev/tty12  /dev/tty20  /dev/tty29  /dev/tty37  /dev/tty45  /dev/tty53  /dev/tty61
/dev/tty13  /dev/tty21  /dev/tty3   /dev/tty38  /dev/tty46  /dev/tty54  /dev/tty62
/dev/tty14  /dev/tty22  /dev/tty30  /dev/tty39  /dev/tty47  /dev/tty55  /dev/tty63
/dev/tty15  /dev/tty23  /dev/tty31  /dev/tty4   /dev/tty48  /dev/tty56  /dev/tty7
cat@lubancat:~$
```

21.4 シリアル通信実験（Shell）

この実験は LubanCat-1 ボードを例に説明しますが、他のボードの操作もこの実験と類似しているため、詳細は省略します。ボード上のシリアル 3 を使用して実験し、対応するデバイスファイルは/dev/ttyS3 です。tty デバイスファイルを直接読み書きすることで、デバイスを介してシリアルでデータを受信または送信することができます。以下では、ボードを Windows のシリアルデバッグアシスタントまたは Linux の minicom と連携してテストします。

21.4.1 シリアルの接続

実験前に、シリアルケーブルまたは USB からシリアルへの変換ケーブルを使用して、ボードとコンピュータを接続する必要があります。

- ボード - コンピュータ

- TXD — RXD

- RXD — TXD

- GND — GND

21.4.2 シリアル 3 の通信パラメータの照会

シリアル 3 の外部デバイスが有効になると、/dev ディレクトリに ttyS3 デバイスファイルが生成されます。stty ツールを使用してその通信パラメータを照会します。

```
1 # ボードのターミナルで以下のコマンドを実行
2 stty -F /dev/ttyS3
```

株式会社日昇テクノロジー

下の画像のように：

```
cat@lubancat:~$ sudo stty -F /dev/ttyS3
speed 9600 baud; line = 0;
-brkint -imaxbel
cat@lubancat:~$
```

21.4.3 シリアルのボーレートの変更

```
1 # 通信速度を設定します。ispeed は入力速度、ospeed は出力速度です。
2 stty -F /dev/ttyS3 ispeed 115200 ospeed 115200
```

下の画像のように：

```
root@lubancat:~# stty -F /dev/ttyS3
speed 115200 baud; line = 0;
-brkint -icrnl -imaxbel
-echo
root@lubancat:~#
```

21.4.4 エコーの無効化

デフォルトでシリアルはエコーが有効になっていますが、以下のコマンドでエコーを無効にすることができます。

```
1 stty -F /dev/ttyS3 -echo
```

21.4.5 Windows ホストとの通信

21.4.5.1 シリアル通信実験

シリアルデバッグアシスタントを設定した後、以下のコマンドを使用してデータ送信のテストを行います：

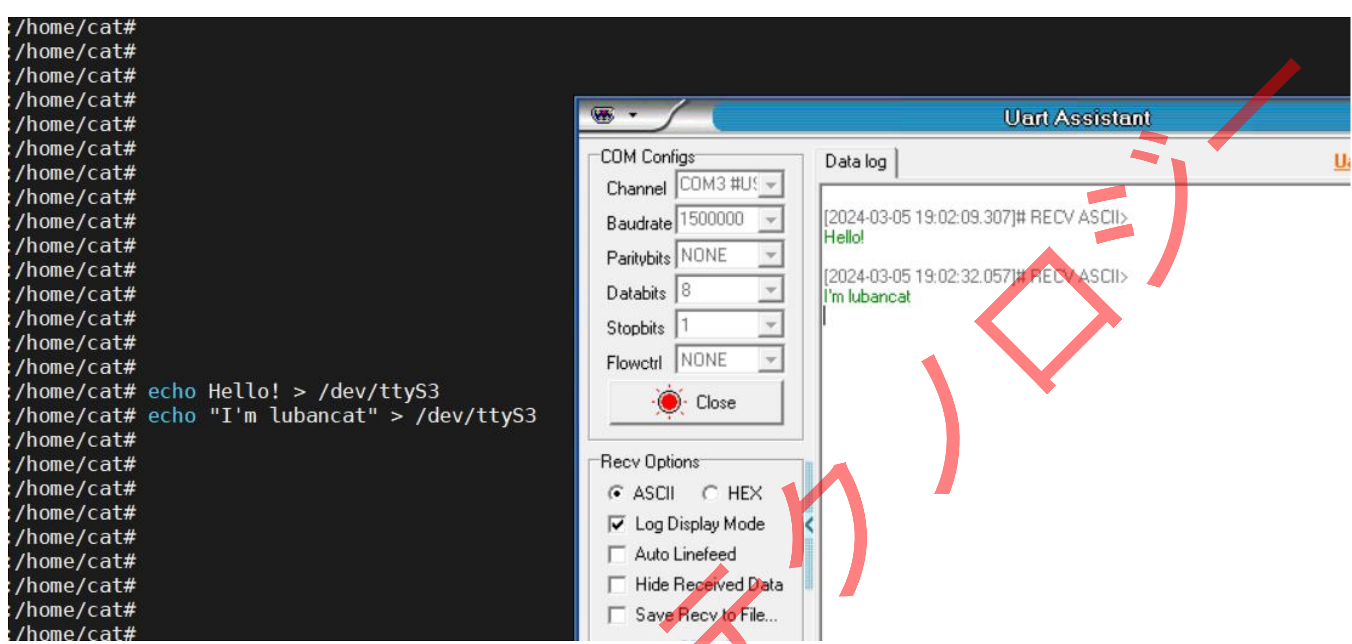
```
1 # ボード上のターミナルで以下の指示を実行します。
2 # echo コマンドを使用して端末デバイスファイルに文字列"Hello!"、"I'm lubancat"を書き込みます。
3 echo Hello! > /dev/ttyS3
```

株式会社日昇テクノロジー

4 echo "I'm lubancat" > /dev/ttyS3

5 #Windows 上のシリアルポートデバッグツールが受信する

下図のように、



/dev/ttyS3 デバイスファイルに書き込まれた内容は直接シリアルケーブルを介して Windows のホストに送信されます。

デバイスファイルを読むことで、Windows ホストからボードに送られた内容を受信できます。cat コマンドを使って読み取ることができます：

1 # ボード上のターミナルで以下のコマンドを実行します

2 # ターミナルデバイスファイルを読み取るために cat コマンドを使用します

3 cat /dev/ttyS3

4 #cat コマンドは待機します

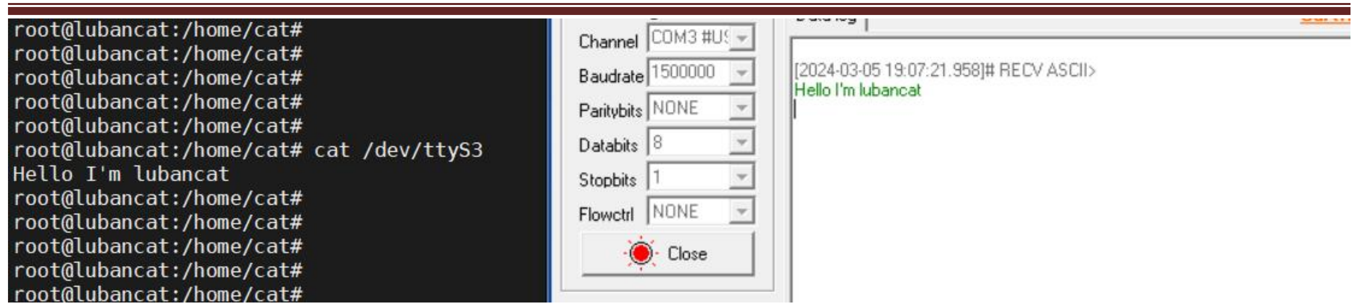
5 # シリアルポートデバッグツールを使用して文字列を送信します

6 # 文字列の最後には改行が必要です！

7 # ボードのターミナルは受信した内容を表示します

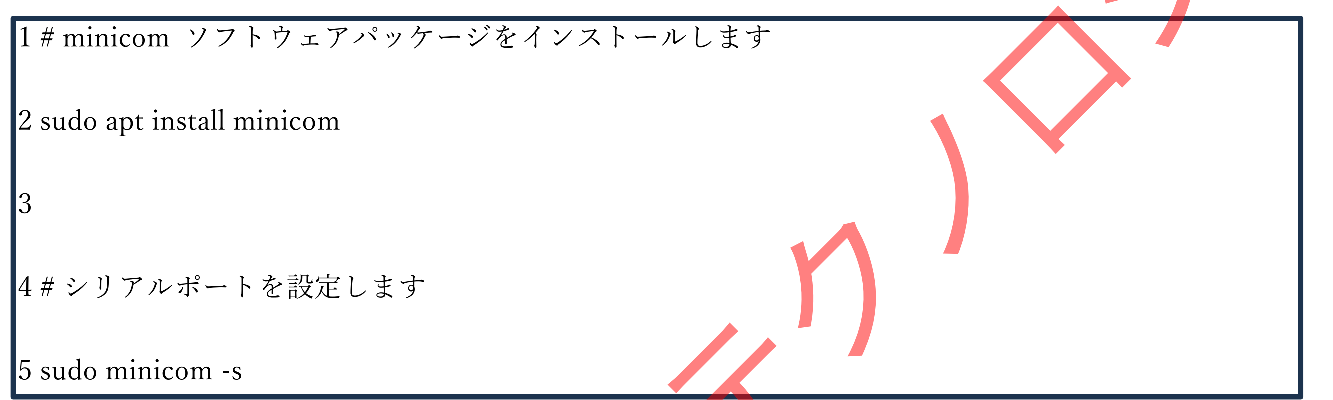
下図のように：

株式会社日昇テクノロジー



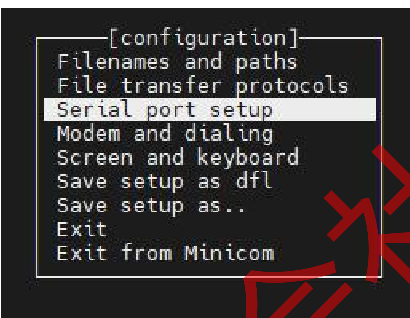
Hello lubancat!

21.4.5.2 minicom 通信



```
1 # minicom ソフトウェアパッケージをインストールします

2 sudo apt install minicom

3

4 # シリアルポートを設定します

5 sudo minicom -s
```

下図のように：

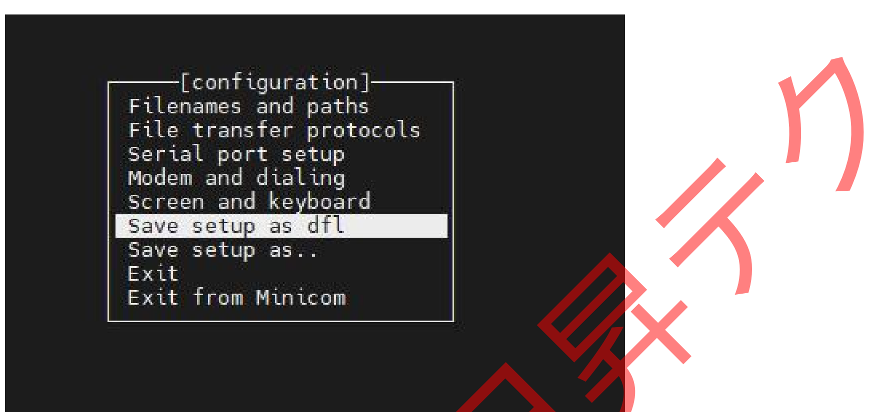


設定に入り、シリアルポートやボーレートを変更します。キーボードの文字を押してそれぞれ設定する項目に入り、enter キーで確認します。

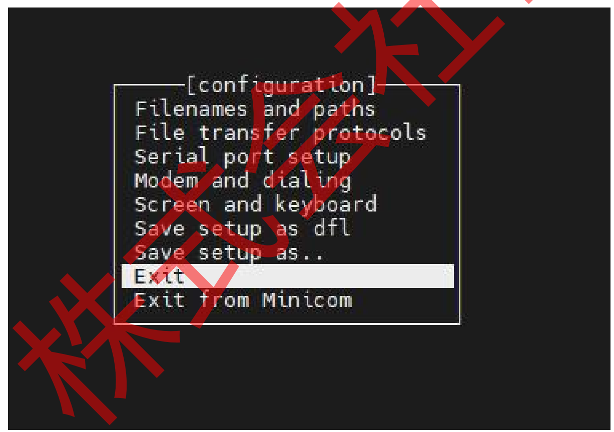
株式会社日昇テクノロジー

```
A -    Serial Device      : /dev/ttyS3
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 115200 8N1
F - Hardware Flow Control : Yes
G - Software Flow Control : No

   Change which setting?

        Screen and keyboard
        Save setup as dfl
        Save setup as..
        Exit
        Exit from Minicom
```

設定が完了したら、このボタンを押して設定を保存します。保存した後、開く時に設定する必要はありません。



その後、exit キーを押して minicom のターミナルに入ります。



文字を入力しても画面に反応がない場合は、エコーを開いて表示させることができます。「ctrl + A」を押した後に'z'キーを押して

株式会社日昇テクノロジー

```
                Minicom Command Summary

            Commands can be called by CTRL-A <key>

             Main Functions                  Other Functions

 Dialing directory..D  run script (Go)....G | Clear Screen.......C
 Send files.........S  Receive files......R | cOnfigure Minicom..O
 comm Parameters....P  Add linefeed.......A | Suspend minicom....J
 Capture on/off.....L  Hangup.............H | eXit and reset.....X
 send break.........F  initialize Modem...M | Quit with no reset.Q
 Terminal settings..T  run Kermit.........K | Cursor key mode....I
 lineWrap on/off....W  local Echo on/off..E | Help screen........Z
 Paste file.........Y  Timestamp toggle...N | scroll Back........B
 Add Carriage Ret...U

           Select function or press Enter for none.
```

メニューに入り、'e'を押すとエコーが成功的に開かれます（左下にヒントが表示され、開いたり閉じたりします）。キーを押してテストしてみて、エコーがあるかどうかを確認します。また、「ctrl + A」を押した後に'z'キーを押し、さらに'c'キーを押して画面をクリアできます。ボードとコンピュータをシリアルケーブルで接続し、115200 に設定します。ボード上で、「Hello! I'm lubancat!」と入力します。

株式会社日昇テクノロジー

```
Hello! I'm lunbancat
```

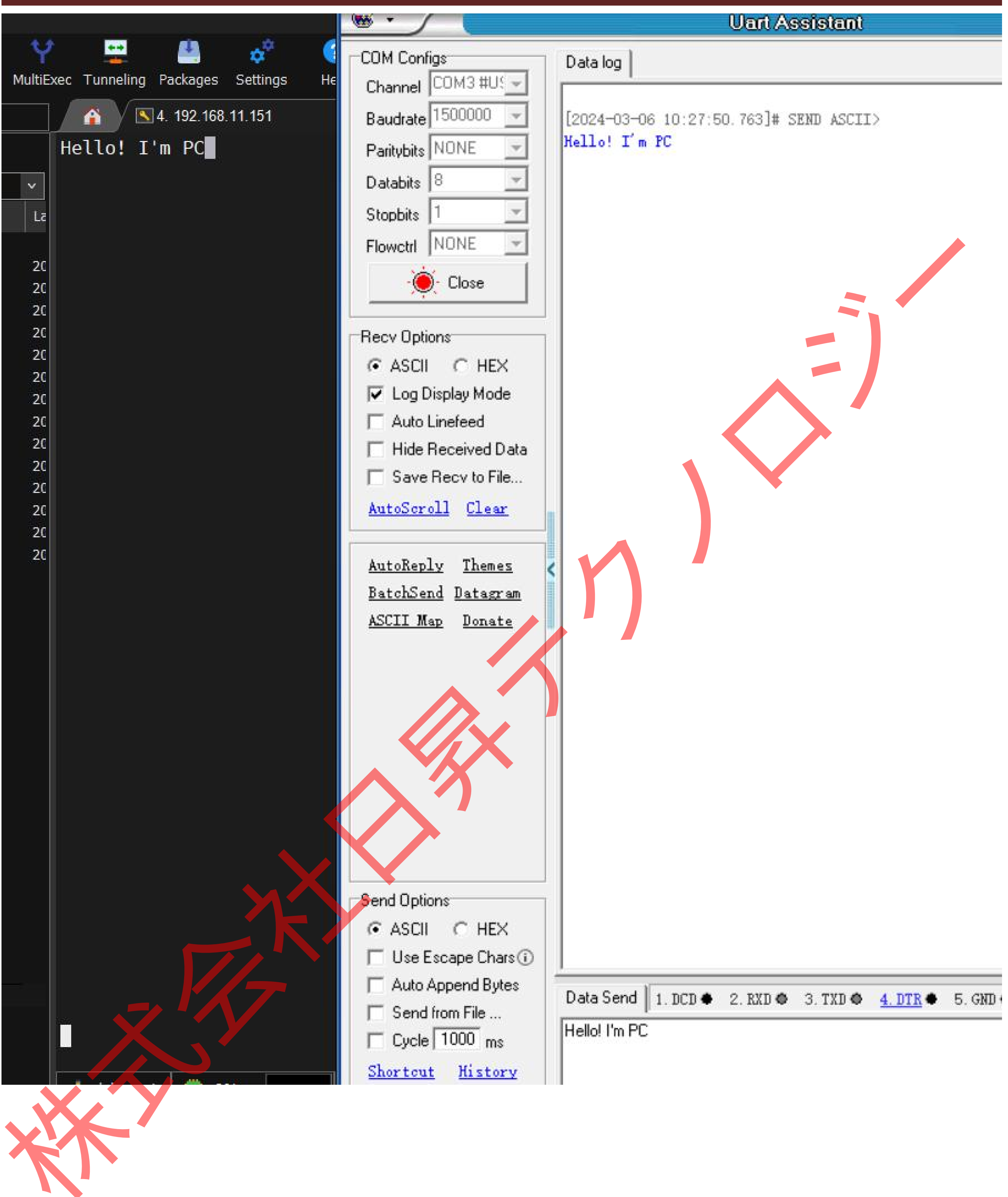
```
[2024-03-06 10:26:06.337]# RECV ASCII>
H
[2024-03-06 10:26:07.171]# RECV ASCII>
e
[2024-03-06 10:26:07.568]# RECV ASCII>
l
[2024-03-06 10:26:07.782]# RECV ASCII>
l
[2024-03-06 10:26:08.190]# RECV ASCII>
o
[2024-03-06 10:26:11.361]# RECV ASCII>
!
[2024-03-06 10:26:12.803]# RECV ASCII>

[2024-03-06 10:26:14.422]# RECV ASCII>
I
[2024-03-06 10:26:17.910]# RECV ASCII>
'
[2024-03-06 10:26:19.347]# RECV ASCII>
m
[2024-03-06 10:26:21.490]# RECV ASCII>

[2024-03-06 10:26:22.723]# RECV ASCII>
l
[2024-03-06 10:26:23.541]# RECV ASCII>
u
[2024-03-06 10:26:23.861]# RECV ASCII>
n
[2024-03-06 10:26:24.335]# RECV ASCII>
b
[2024-03-06 10:26:24.775]# RECV ASCII>
a
[2024-03-06 10:26:25.076]# RECV ASCII>
n
[2024-03-06 10:26:25.395]# RECV ASCII>
c
[2024-03-06 10:26:25.788]# RECV ASCII>
a
[2024-03-06 10:26:26.212]# RECV ASCII>
t
```

注意：minicom を使用する際、バックスペースキーを使用して送信した内容を削除することはできません。

minicom は 1 文字ずつ送信する方式を採用しています。

PC 側からテキストを送信できます

株式会社日昇テクノロジー



株式会社日昇テクノロジー

```
1 minicom のメニューには多くの機能があります、
2 s ---- ファイルを送信
3 p ---- 通信パラメータを設定、ボーレート、データフォーマット、データビットなどのプリセットを含む
4 l ---- ログ情報をファイルに保存できます。確認が容易になります
5 t ---- ターミナルパラメータやキー設定を設定
6 w ---- 1 行を超えるデータ後に自動的に改行
7 r ---- ファイルを受信
8 a ---- 改行送信時にタイムスタンプを追加
9 n ---- タイムスタンプを追加
10 c ---- 画面をクリア
11 o ---- minicom を設定、sudo minicom -s と同等
12 j ---- スリープ状態
13 x ---- 退出時にリセット
14 q ---- 退出
15
16 "ctrl + a" + 'z' + '?'を押す方法でなくても、
17 直接"ctrl + a" + '?'を使用して設定できます
```

21.5 シリアル通信実験（システムコール）

シリアルターミナルデバイスを介してデータを送受信したいだけの場合、open、read、write などのシステムコールを使用して簡単に実現できます。操作の原理は、以前の LED、GPIO、input デバイスと変わりません。すべてデバイスファイルの読み書きです。しかし LED、GPIO、input は主デバイスファイルの他に、デバイスの動作パラメータを設定するための多くの属性ファイルがあります。例えば LED の trigger ファイル、GPIO の direction ファイルですが、ターミナルデバイスにはその他の属性ファイルが

株式会社日昇テクノロジー

ありません。では、stty コマンドや minicom ツールはどのようにしてターミナルデバイスのパラメータを設定しているのでしょうか？

21.5.1 実験

21.5.1.1 コード

リスト 1: base_linux/uart/uart_t/uart_t.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <sys/stat.h>
6 #include <sys/types.h>
7 #include <termios.h>
8 #include <string.h>
9 #include <sys/ioctl.h>
10
11 //第一部のコード/
12 //具体的なデバイスに応じて変更してください
13 const char default_path[] = "/dev/ttyS3";
14
15 int main(int argc, char *argv[])
16 {
17 int fd;
```

株式会社日昇テクノロジー

```
18 int res;
19 char *path;
20 char buf[1024] = "Embedfire tty send test.¥n";
21
22 //第二部のコード/
23 //入力パラメータがない場合はデフォルトのターミナルデバイスを使用します
24 if (argc > 1)
25 path = argv[1];
26 else
27 path = (char *)default_path;
28
29 //シリアルデバイスのディスクリプタを取得します
30 printf("This is tty/usart demo.¥n");
31 fd = open(path, O_RDWR);
32 if (fd < 0) {
33 printf("Fail to Open %s device¥n", path);
34 return 0;
35 }
36
37
38 //第三部のコード/
39 struct termios opt;
40 //シリアル受信バッファをクリアします
```

株式会社日昇テクノロジー

```
41 tcflush(fd, TCIOFLUSH);
42 // シリアルパラメータ opt を取得します
43 tcgetattr(fd, &opt);
44 //シリアル出力ボーレートを設定します
45 cfsetospeed(&opt, B9600);
46 //シリアル入力ボーレートを設定します
47 cfsetispeed(&opt, B9600);
48
49 //データビット数を設定します
50 opt.c_cflag &= ~CSIZE;
51 opt.c_cflag |= CS8;
52 //パリティビット
53 opt.c_cflag &= ~PARENB;
54 opt.c_iflag &= ~INPCK;
55 //ストップビットを設定します
56 opt.c_cflag &= ~CSTOPB;
57 //設定を更新します
58 tcsetattr(fd, TCSANOW, &opt);
59 printf("Device %s is set to 9600bps,8N1¥n",path);
60
61 //第四部のコード/
62 do {
63 //文字列を送信します
```

株式会社日昇テクノロジー

```
64 write(fd, buf, strlen(buf));
65 //文字列を受信します
66 res = read(fd, buf, 1024);
67 if (res >0 )
68 //受信した文字列に終端記号を追加します
69 buf[res] = '¥0';
70 printf("Receive res = %d bytes data: %s¥n",res, buf);
71 } while (res >= 0);
72
73 printf("read error,res = %d",res);
74 close(fd);
75 return 0;
76 }
```

21.5.1.2 コンパイル

方法 1:

```
1 # コンパイルします
2 make
```

方法 2:

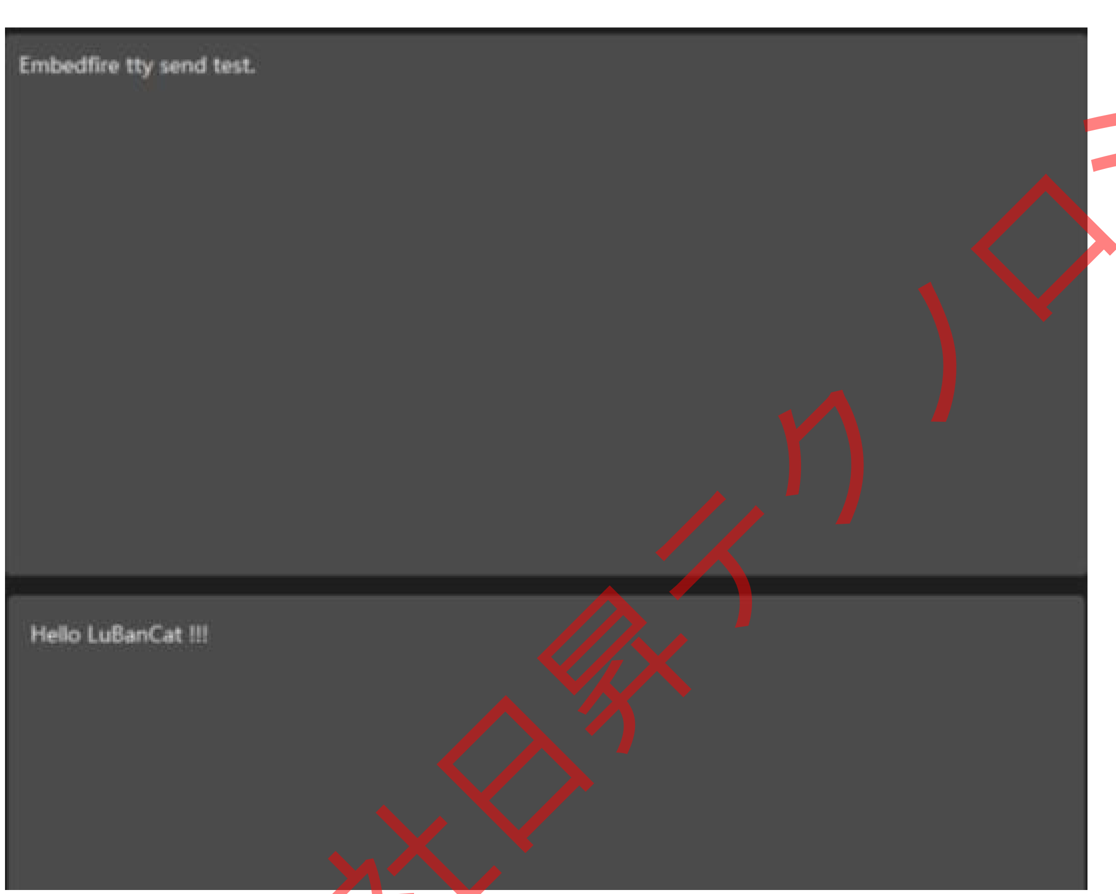
```
1 # コンパイルします
2 gcc uart_t.c -o uart_t
```

株式会社日昇テクノロジー

21.5.1.3 実行

```
1 # 実行します
2 sudo ./uart_t
```

• プログラムを実行します



• シリアルポートアシスタント上で文字列を送信します

株式会社日昇テクノロジー

• 2 つの Hello LubanCat !!!が返されます。1 つはエコーで、もう 1 つはプログラムが内容をホストに送信したものです

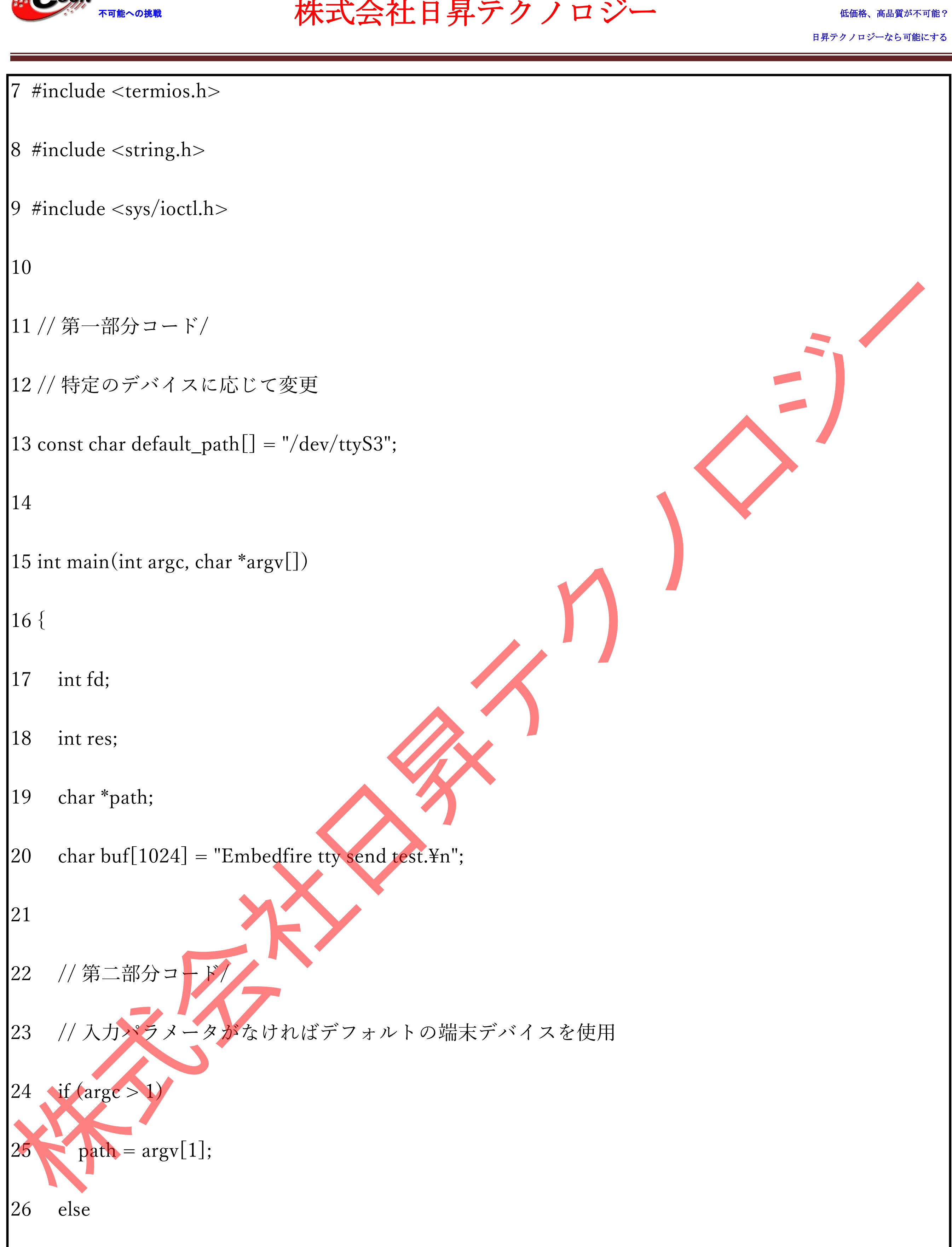
21.5.2 コード分析

このセクションでは、シリアルポートのパラメータを変更する例のソースコードを通じて、疑問に答えます。ソースコードは以下の通りです。

リスト 2: base_linux/uart/uart_t/uart_t.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <sys/stat.h>
6 #include <sys/types.h>
```

株式会社日昇テクノロジー



```
7  #include <termios.h>
8  #include <string.h>
9  #include <sys/ioctl.h>
10
11 // 第一部分コード/
12 // 特定のデバイスに応じて変更
13 const char default_path[] = "/dev/ttyS3";
14
15 int main(int argc, char *argv[])
16 {
17     int fd;
18     int res;
19     char *path;
20     char buf[1024] = "Embedfire tty send test.¥n";
21
22     // 第二部分コード/
23     // 入力パラメータがなければデフォルトの端末デバイスを使用
24     if (argc > 1)
25         path = argv[1];
26     else
27         path = (char *)default_path;
28
```

株式会社日昇テクノロジー

```
29    // シリアルデバイスのディスクリプタを取得
30    printf("This is tty/usart demo.¥n");
31    fd = open(path, O_RDWR);
32    if (fd < 0) {
33        printf("Fail to Open %s device¥n", path);
34        return 0;
35    }
36
37    // 第三部分コード/
38    struct termios opt;
39    // シリアル受信バッファをクリア
40    tcflush(fd, TCIOFLUSH);
41    // シリアルパラメータ opt を取得
42    tcgetattr(fd, &opt);
43    // シリアル出力ボーレートを設定
44    cfsetospeed(&opt, B9600);
45    // シリアル入力ボーレートを設定
46    cfsetispeed(&opt, B9600);
47
48    // データビット数を設定
49    opt.c_cflag &= ~CSIZE;
50    opt.c_cflag |= CS8;
```

株式会社日昇テクノロジー

```
51    // パリティビットを設定
52    opt.c_cflag &= ~PARENB;
53    opt.c_iflag &= ~INPCK;
54    // ストップビットを設定
55    opt.c_cflag &= ~CSTOPB;
56    // 設定を更新
57    tcsetattr(fd, TCSANOW, &opt);
58    printf("Device %s is set to 9600bps,8N1¥n", path);
59
60    // 第四部分コード/
61    do {
62        // 文字列を送信
63        write(fd, buf, strlen(buf));
64        // 文字列を受信
65        res = read(fd, buf, 1024);
66        if (res > 0)
67            // 受信した文字列に終端文字を追加
68            buf[res] = '¥0';
69        printf("Receive res = %d bytes data: %s¥n", res, buf);
70    } while (res >= 0);
71
72    printf("read error, res = %d", res);
```

株式会社日昇テクノロジー

```
73    close(fd);

74    return 0;

75 }
```

説明のために、コードを 4 つの部分に分けました：

- **第一部分**：デフォルトで使用するシリアル端末デバイスのパスとその他の変数を定義しています。

- **第二部分**：main 関数が入力パラメータを持つかどうかに基づいて、どのデバイスパスを使用するかを決定し、open の O_RDWR モードでデバイスを開きます。

- **第三部分**：termios 構造体を定義して、端末デバイスのパラメータを取得・設定します。これにはボーレート、データビット、パリティビットなどが含まれます。これは本章の重点であり、次の小節で詳しく説明します。

- **第四部分**：while ループ内で端末デバイスに対して read と write を使用してデータの送受信を制御します。コードは受信した内容の末尾に'¥0'終端文字を追加しており、これは文字列として内容を扱うのに便利です。

21.5.2.1 termios 構造体

例のコードの第三部分で使用される termios 構造体は、POSIX 規格で定義された標準インターフェースです。Linux システムは termios を使用してシリアルポートのパラメータを設定します。これは、ヘッダーファイル<termios.h>に含まれる<bits/termios.h>で定義されています。このファイルには、各構造体メンバーで使用できるマクロ値も含まれています。自分で locate コマンドを使用してこのファイルを探し、開いて読むことをお勧めします。termios 構造体の定義は以下の通りです。

株式会社日昇テクノロジー

リスト 3: termios 構造体（/usr/include/bits/termios.h ファイルに位置）

```
1 struct termios {

2 tcflag_t c_iflag; /* input mode flags */

3 tcflag_t c_oflag; /* output mode flags */

4 tcflag_t c_cflag; /* control mode flags */

5 tcflag_t c_lflag; /* local mode flags */

6 cc_t c_line; /* line discipline */

7 cc_t c_cc[NCCS]; /* control characters */

8 speed_t c_ispeed; /* input speed */

9 speed_t c_ospeed; /* output speed */

10 #define _HAVE_STRUCT_TERMIOS_C_ISPEED 1

11 #define _HAVE_STRUCT_TERMIOS_C_OSPEED 1

12 };
```

株式会社日昇テクノロジー

ここでは、特に c_iflag、c_cflag、c_ispeed、c_ospeed の各構造体メンバーに焦点を当てます：

- **c_iflag**：入力モードフラグ。シリアル入力の文字をどのように処理するかを制御します。よく使用されるオプション値は下リストを参照してください。

c_iflag オプション値:

オプション値	効果
INPCK	入力のパリティチェックを有効にする
IGNPAR	フレームエラーとパリティエラーを無視する
IGNCR	入力中のキャリッジリターンを無視する
IXON	XON/XOFF フロー制御を有効にする
IXOFF	XON/XOFF フロー制御を無効にする

- **c_oflag**：出力モードフラグ。シリアル出力モードを制御します。よく使用されるオプション値は

下リストを参照してください。

c_oflag オプション値

オプション値	効果
ONLCR	出力中の改行 NL をキャリッジリターン CR にマッピングする
OCRNL	出力のキャリッジリターンを改行符にマッピングする
ONLRET	キャリッジリターンを出力しない
OFILL	填充文字列を送信する

- c_cflag：制御モードフラグで、データビット、ストップビットなど、シリアルポートの基本パラメータを制御するために使用されます。よく使用される設定は下リストに示されています。特に、c_cflag 構造体メンバーには、ボーレートのパラメータも含まれています。

_cflag オプション値

オプション値	効果
CSIZE	データビットの長さを設定する。
CSTOPB	CSTOPB フラグが設定されている場合、2 つのストップビットを使用する
PARENB	パリティチェックを有効にする
PARODD	奇数パリティを設定する

- **c_lflag**：ローカルモードフラグ。主にドライバプログラムとユーザーのやりとりを制御します。シリアル通信では実際には使用されません。

オプション値	効果
ISIG	ISIG フラグが設定されている場合、INTR、QUIT などの文字を受信すると、システムは対応するシグナルを生成します。
ECHO	文字をエコーバックするかどうか
ICANON	ICANON フラグが設定されている場合、端末は正規モード入力状態にあり、EOF、KILL などの特殊文字の使用が許可されます。
ECHONL	このフラグと ICANON フラグが同時に設定され

株式会社日昇テクノロジー

	ている場合、改行文字 NL をエコーバックします。

- **c_cc[NCCS]**：特別なキャラクターの配列。特別なキャラクターに対応するキー値を変更できます（例：Ctrl+C が生成する^C、ASCII コードは 0x03）。

リスト c_cc の各メンバーに対応するインデックス値

配列のインデックス値	効果
VINTR	中断文字。この文字を受信すると、SIGINT シグナルが送信されます。c_lflag の ISIG フラグが設定されている場合、この文字は入力として渡されなくなります。
VERASE	削除文字、前の文字を削除します。
VIM	非標準モード読み取りの最小文字数を設定します。
VTIM	非標準モードの読み取り時のタイムアウト値を設定します。単位は十分の一秒です。

• c_ispeed と c_ospeed：シリアルポートの入力および出力ボーレート（input speed および output speed）を記録します。部分的に取り得る値は以下のコードに示されています。マクロ定義の数字は「0」で始まり、C 言語ではこれは 8 進数をリストす方法です。

リスト 4: ボーレート定義（/usr/include/bits/termios.h に位置）

```
1 //「0」で始まる数字は、C 言語における 8 進数の形式であることに注意
2 #define B1200 0000011
3 #define B1800 0000012
4 #define B2400 0000013
5 #define B4800 0000014
6 #define B9600 0000015
7 #define B19200 0000016
```

株式会社日昇テクノロジー

8 #define B38400 0000017

例のコードでは、現在のパラメータを取得し、設定を変更し、設定を更新するという手順を踏んでいます。設定を変更するコードでは、&=~、|= というビット操作方法を使用しています。これは、変数内の他のビットに影響を与えないようにするためです。c_cflag の他のビットには、パリティビット、データビット、ボーレートに関連する設定が含まれているため、"=" で値を直接設定すると、他の設定が影響を受け、操作も不便になります。後に裸のマシン開発を学ぶ際には、このような操作を頻繁に使用することになります。これらのビット操作方法に慣れていない場合は、本書の付録「第 65 章 ビット操作方法」を参照してください。簡単に言うと、例の中の &=~ は c_cflag 変数中の CSTOPB に対応するデータビットを 0 にクリアし、|= は c_cflag 変数中の CSTOPB に対応するデータビットを 1 に設定します。これにより、他の設定に影響を与えることなく、ストップビットを 1 ビットまたは 2 ビットに設定することができます。

21.5.2.1.3 シリアルポートのパリティビットの設定

シリアルポートのパリティビットを設定するには、termios のメンバー c_cflag のフラグビット PARENB、PARODD および c_iflag のフラグビット INPCK が関与します。PARENB と INPCK は共に、奇数パリティか偶数パリティかを決定する PARODD を使用して、パリティチェックを有効にするかどうかを決定します。設定のサンプルコードは以下の通りです。

株式会社日昇テクノロジー

```
1 //bits/termios.h のビット定義
2 //注意：C 言語では、0 で始まる数字は 8 進数の形式です
3 /* c_cflag ビットの意味 */
4 #define PARENB 0000400
5 #define PARODD 0001000
6 /* c_iflag ビット */
7 #define INPCK 0000020
8 //
9 //termios 型の変数 opt を定義
10 struct termios opt;
11
12 // シリアルポートのパラメータを opt に取得
13 tcgetattr(fd, &opt);
14
15 switch (parity)
16 {
17 case 'n':
18 case 'N':
19 opt.c_cflag &= ~PARENB; /* パリティチェックを使用しない */
20 opt.c_iflag &= ~INPCK; /* 入力パリティチェックを無効にする */
21 break;
```

株式会社日昇テクノロジー

```
22
23 case 'o':
24 case 'O':
25 opt.c_cflag |= PARENB; /* パリティチェックを有効にする */
26 opt.c_iflag |= INPCK; /* 入力パリティチェックを有効にする */
27 opt.c_cflag |= PARODD; /* 奇数パリティを設定 */
28 break;
29
30 case 'e':
31 case 'E':
32 opt.c_cflag |= PARENB; /* パリティチェックを有効にする */
33 opt.c_iflag |= INPCK; /* 入力パリティチェックを有効にする */
34 opt.c_cflag &= ~PARODD; /* 偶数パリティを設定 */
35 break;
36 }
37
38 //設定を更新
39 tcsetattr(fd, TCSANOW, &opt);
```

設定は非常にシンプルです。検証を行わない場合は、PARENB と INPCK を同時に 0 に設定し、検証を有効にする場合は、PARENB と INPCK を同時に 1 に設定します。PARODD が 1 の場合は奇数パリティを指定し、0 の場合は偶数パリティを指定します。

株式会社日昇テクノロジー

21.5.2.1.4 シリアルポートのデータビットの設定

シリアルポートのデータビットは、c_cflag 内の CSIZE で設定されます。シリアルポートは 5、6、7、8 ビットの設定をサポートしているため、合計 4 つの設定があります。そのため、c_cflag では 2 つのデータビットを使用して設定します。設定前には、まず CSIZE データビットをクリアし、次に 5、6、7、8 のマクロ設定値を割り当てます。具体的なコードは以下の通りです。

「リスト 8: データビット長を設定するサンプルコード」

```
1 //bits/termios.h のビット定義
2 //注意：C 言語では、0 で始まる数字は 8 進数の形式です
3 #define CSIZE 0000060
4 #define CS5 0000000
5 #define CS6 0000020
6 #define CS7 0000040
7 #define CS8 0000060
8 //
9 //termios 型の変数 opt を定義
10 struct termios opt;
11 // シリアルポートのパラメータを opt に取得
12 tcgetattr(fd, &opt);
13
14 //まず、CSIZE データビットの内容をクリア
15 opt.c_cflag &= ~CSIZE;
16
```

株式会社日昇テクノロジー

```
17 switch (databits) /* データビット数を設定 */
18 {
19 case 5:
20 opt.c_cflag |= CS5;
21 break;
22 case 6:
23 opt.c_cflag |= CS6;
24 break;
25 case 7:
26 opt.c_cflag |= CS7;
27 break;
28 case 8:
29 opt.c_cflag |= CS8;
30 break;
31 }
32 //設定を更新
33 tcsetattr(fd, TCSANOW, &opt);
```

termios 構造体を使用してシリアルポートのパラメータを設定するさまざまな方法を学んだ後、もう一度前の main.c サンプルファイルコードを見てみましょう。もはや紹介する必要はないでしょう。

21.6 ioctl システムコール

これまでの学習を通じて、デバイスファイルを操作してシリアルポートのパラメータを設定する方法を習得しました。前のコードで使用されたファイル操作の抜粋は以下の通りです。

株式会社日昇テクノロジー

リスト 9: デバイスファイルに関連する関数操作

```
1 //前の実験でデバイスファイルを操作するために使用された関数
2 fd = open(path, O_RDWR);
3 write(fd, buf, strlen(buf));
4 read(fd, buf, 1024);
5 close(fd);
6 tcgetattr(fd, &opt);
7 tcsetattr(fd, TCSANOW, &opt);
```

これらの操作を詳しく分析すると、晴れ渡った空に 2 つの暗雲が現れたことがわかります。open、write、read、close はすべて Linux のシステムコールであり、tcgetattr、tcsetattr はライブラリ関数です。そして、伝統的な認識によれば、ファイル操作は大抵内容と関連があるものですが、前章での input イベントデバイスファイルは報告されたイベント情報を記録しており、tty デバイスのファイルはシリアルポートの設定パラメータを記録しているわけではありません。なぜなら、ファイルへの write 操作は外部へのデータ送信であり、read 操作は受信データの読み取りであるため、つまり、「tty*」ファイルはシリアルポートの設定情報を記録していないのです。では、tcgetattr、tcsetattr の 2 つの関数は一体何をしているのでしょうか？

実際には、これらは ioctl システムコールのラッパーです。

21.6.1 ioctl の原型

ioctl システムコールは、デバイスファイルにコマンドを送信し、特殊な操作を制御する機能を提供します。関数の原型は以下の通りです。

```
1 #include <sys/ioctl.h>
2 int ioctl(int fd, unsigned long request, ...);
```

株式会社日昇テクノロジー

- 引数 fd：write、read と同様に、fd ファイルハンドルはどのファイルを操作するかを指定します。

- 引数 request：操作リクエストのコードで、これはハードウェアデバイスドライバに関連しています。異なるドライバデバイスは異なるコードをサポートしており、ドライバプログラムは通常、使用可能なコードを上層ユーザーに提供するためにヘッダーファイルを使用します。

- 引数「...」：これは定義されていない型のポインタで、printf 関数定義の「...」に似ていますが、ioctl では 1 つの引数のみを渡すことができます。一部のドライバプログラムが操作リクエストを実行する際に設定パラメータが必要であったり、操作完了時にデータを返す必要がある場合、このポインタを介してアクセスされます。

21.6.2 tcgetattr と tcsetattr の代わりに ioctl を使用する

リスト 10: base_linux/uart/uart_i/uart_i.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <sys/stat.h>
6 #include <sys/types.h>
7 #include <termios.h>
8 #include <string.h>
9 #include <sys/ioctl.h>
10
11 //具体的なデバイスに応じて変更してください
```

株式会社日昇テクノロジー

12 const char default_path[] = "/dev/ttyS3";

```
13
14 int main(int argc, char *argv[])
15 {
16 int fd;
17 int res;
18 struct termios opt;
19 char *path;
20 char buf[1024] = "Embedfire tty 送信テスト。¥n";
21 //入力パラメータがない場合はデフォルトの端末デバイスを使用
22 if(argc > 1)
23 path = argv[1];
24 else
25 path = (char *)default_path;
26 //シリアルポートデバイス記述子を取得
27 printf("これは tty/usart デモです。¥n");
28 fd = open(path, O_RDWR);
29 if(fd < 0){
30 printf("%s デバイスのオープンに失敗しました¥n", path);
31 return 0;
32 }
33 //シリアルポート受信バッファをクリア
34 tcflush(fd, TCIOFLUSH);
```

株式会社日昇テクノロジー

35

```
36 // シリアルポートのパラメータを取得

37 // tcgetattr(fd, &opt);

38 res = ioctl(fd, TCGETS, &opt);

39 opt.c_ispeed = opt.c_cflag & (CBAUD | CBAUDEX);

40 opt.c_ospeed = opt.c_cflag & (CBAUD | CBAUDEX);

41

42 //マクロの値を出力、比較のため

43 printf("マクロ B9600 = %#o¥n", B9600);

44 printf("マクロ B115200 = %#o¥n", B115200);

45

46 //読み取った値を出力

47 printf("ioctl TCGETS, opt.c_ospeed = %#o¥n", opt.c_ospeed);

48 printf("ioctl TCGETS, opt.c_ispeed = %#o¥n", opt.c_ispeed);

49 printf("ioctl TCGETS, opt.c_cflag = %#x¥n", opt.c_cflag);

50 speed_t change_speed = B9600;

51

52 if(opt.c_ospeed == B9600)

53 change_speed = B115200;

54

55 //シリアルポートの出力ボーレートを設定

56 cfsetospeed(&opt, change_speed);

57 //シリアルポートの入力ボーレートを設定
```

株式会社日昇テクノロジー

```
58 cfsetispeed(&opt, change_speed);

59

60 //データビット数を設定

61 opt.c_cflag &= ~CSIZE;

62 opt.c_cflag |= CS8;

63 //パリティビット

64 opt.c_cflag &= ~PARENB;

65 opt.c_iflag &= ~INPCK;

66 //ストップビットを設定

67 opt.c_cflag &= ~CSTOPB;

68

69 //設定を更新

70 // tcsetattr(fd, TCSANOW, &opt);

71 res = ioctl(fd, TCSETS, &opt);

72 //再度読み取り

73 res = ioctl(fd, TCGETS, &opt);

74 opt.c_ispeed = opt.c_cflag & (CBAUD | CBAUDEX);

75 opt.c_ospeed = opt.c_cflag & (CBAUD | CBAUDEX);

76 printf("ioctl TCGETS after TCSETS¥n");

77

78 //読み取った値を出力

79 printf("ioctl TCGETS, opt.c_ospeed = %#o¥n", opt.c_ospeed);
```

株式会社日昇テクノロジー

```
80 printf("ioctl TCGETS, opt.c_ispeed = %#o¥n", opt.c_ispeed);
81 printf("ioctl TCGETS, opt.c_cflag = %#x¥n", opt.c_cflag);
82 do{
83 //文字列を送信
84 write(fd, buf, strlen(buf));
85 //文字列を受信
86 res = read(fd, buf, 1024);
87 if(res > 0){
88 //受信した文字列に終端記号を追加
89 buf[res] = '¥0';
90 printf("受信データ：%d バイト：%s¥n", res, buf);
91 }
92 }while(res >= 0);
94 printf("読み取りエラー、res = %d", res);
95 close(fd);
96 return 0;
97 }
```

この実験コードは、tcgetattr と tcsetattr の 2 つのライブラリ関数を ioctl システムコールで直接置き換えています。

- サンプルコードの第 41 行と 76 行では、デバイスファイルに「TCGETS」リクエストを ioctl で送信しており、tty デバイスのドライバ層はこのリクエストに基づいて設定パラメータを返し、渡された &opt ポインタを介して出力します。

株式会社日昇テクノロジー

- 同様に、サンプルコードの第 73 行では、デバイスファイルに「TCSETS」リクエストを ioctl で送信しており、tty デバイスのドライバ層はこのリクエストに基づいて、&opt ポインタで渡された設定パラメータを設定し、デバイスの属性を変更します。

- 特に、ioctl を使用して設定パラメータを取得する際、ボーレートの値は termios 構造体の c_ispeed および c_ospeed メンバに直接書き込まれないため、c_cflag の値を計算して求める必要があります。そのため、第 43、44 行および 78、79 行では計算変換が追加されており、計算された値は B9600 や B115200 などの値になります。このような計算操作を行わなければ、c_ispeed と c_ospeed が正しく得られない可能性があります。

- コードの他の部分は、実験中に取得した情報が正しいかを確認するためのデバッグ情報の出力です。

ioctl の TCGETS および TCSETS パラメータについては、man マニュアルで確認できます。以下のコマンドを使用してください：

```
1 man ioctl_tty
```

```
IOCTL_TTY(2)                    Linux Programmer's Manual                    IOCTL_TTY(2)

NAME
       ioctl_tty - ioctls for terminals and serial lines

SYNOPSIS
       #include <termios.h>

       int ioctl(int fd, int cmd, ...);

DESCRIPTION
       The ioctl(2) call for terminals and serial ports accepts many possible command arguments.  Most require a third argument, of varying type, here called argp or arg.

       Use of ioctl makes for nonportable programs.  Use the POSIX interface described in termios(3) whenever possible.

   Get and set terminal attributes
       TCGETS     struct termios *argp
              Equivalent to tcgetattr(fd, argp).
              Get the current serial port settings.

       TCSETS     const struct termios *argp
              Equivalent to tcsetattr(fd, TCSANOW, argp).
              Set the current serial port settings.

       TCSETSW    const struct termios *argp
              Equivalent to tcsetattr(fd, TCSADRAIN, argp).
              Allow the output buffer to drain, and set the current serial port settings.
```

ioctl システムコールは非常に幅広く使用されています。すべてのデバイスが読み書き操作のみを持っているわけではないためです。たとえば、CD-ROM の排出や収納、特殊なデバイスの機械操作、または自分で LED ドライバプログラムを書いて上層に命令を提供して花式点灯を実現するなど、ドライバプログラムの作成時には、ioctl システムコール関連のインターフェースについてさらに学習します。

21.6.3 コンパイル

方法 1：

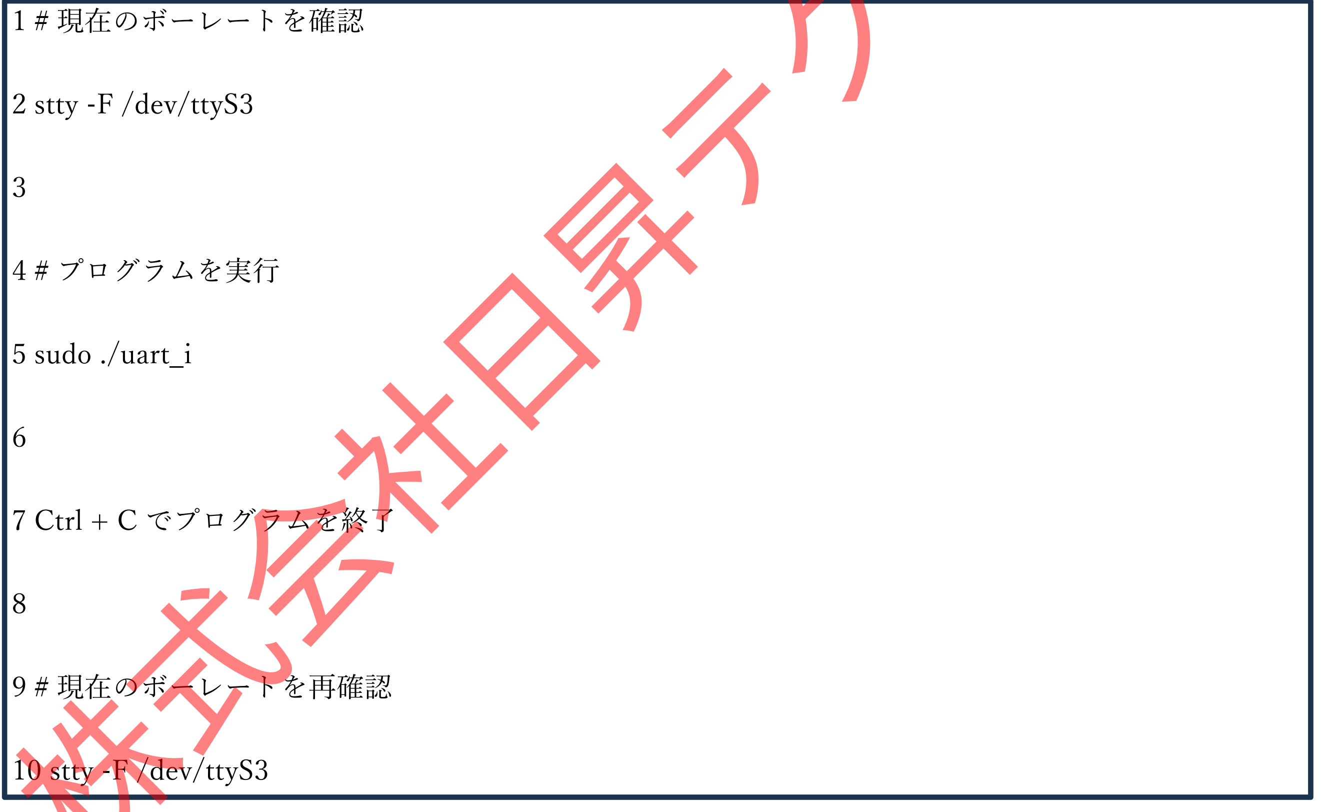
```
1 # コンパイル

2 make
```

方法 2：

```
1 # コンパイル

2 gcc uart_i.c -o uart_i
```

21.6.4 実行

```
1 # 現在のボーレートを確認

2 stty -F /dev/ttyS3

3

4 # プログラムを実行

5 sudo ./uart_i

6

7 Ctrl + C でプログラムを終了

8

9 # 現在のボーレートを再確認

10 stty -F /dev/ttyS3
```



実行結果

```
1 cat@lubancat:~/23/lubancat-test/base_linux/uart$ sudo stty -F /dev/ttyS3
2 speed 9600 baud; line = 0;
3 -brkint -imaxbel
4 cat@lubancat:~/23/lubancat-test/base_linux/uart$ sudo ./uart_i
5 This is tty/usart demo.
6 Macro B9600 = 015
7 Macro B115200 = 010002
8 ioctl TCGETS,opt.c_ospeed = 015
9 ioctl TCGETS,opt.c_ispeed = 015
10 ioctl TCGETS,opt.c_cflag = 0xcbd
11 ioctl TCGETS after TCSETS
12 ioctl TCGETS,opt.c_ospeed = 010002
13 ioctl TCGETS,opt.c_ispeed = 010002
14 ioctl TCGETS,opt.c_cflag = 0x1cb2
15 ^C
16
17 cat@lubancat:~/23/lubancat-test/base_linux/uart$ sudo stty -F /dev/ttyS3
18 speed 115200 baud; line = 0;
19 -brkint -imaxbel
20 cat@lubancat:~/23/lubancat-test/base_linux/uart$
```

株式会社日昇テクノロジー

21.7 glibc のソースコードを見る

tcgetattr.c と tcsetattr.c が ioctl システムコールを介して実装されていること、また c_ispeed と c_ospeed が c_cflag メンバを計算して得られる必要があることをどのように知ったかと聞かれた場合、答えはソースコードを見ることです。Linux はオープンソースであり、これらの宝をどのように掘り下げるかにかかっています。

それらがライブラリ関数であるため、glibc のソースコードを探しに行きます。glibc のソースコードは公式ウェブサイト http://www.gnu.org/software/libc/からダウンロードできます。ソースコードをダウンロードした後、VS Code エディタの検索機能を使用すれば、関連する内容を見つけることができます。

これら 2 つの関数の定義は、glibc ソースコードの glibc/sysdeps/unix/sysv/linux/ディレクトリにあります。このディレクトリ内の tcgetattr.c および tcsetattr.c ファイルは、これら 2 つの関数をそれぞれ定義しています。

tcsetattr.c ファイルの内容は以下の通りです。

リスト 11: glibc の tcsetattr のソースコード

株式会社日昇テクノロジー

(glibc/sysdeps/unix/sysv/linux/tcsetattr.c ファイル)

```
1 int
2 __tcsetattr (int fd, int optional_actions, const struct termios *termios_p)
3 {
4 struct __kernel_termios k_termios;
5 unsigned long int cmd;
6
7 switch (optional_actions)
8 {
9 case TCSANOW:
10 cmd = TCSETS;
11 break;
12 case TCSADRAIN:
13 cmd = TCSETSW;
14 break;
15 case TCSAFLUSH:
16 cmd = TCSETSF;
17 break;
18 default:
19 return INLINE_SYSCALL_ERROR_RETURN_VALUE (EINVAL);
20 }
21
```

株式会社日昇テクノロジー

```
22 k_termios.c_iflag = termios_p->c_iflag & ~IBAUD0;

23 k_termios.c_oflag = termios_p->c_oflag;

24 k_termios.c_cflag = termios_p->c_cflag;

25 k_termios.c_lflag = termios_p->c_lflag;

26 k_termios.c_line = termios_p->c_line;

27 #if _HAVE_C_ISPEED && _HAVE_STRUCT_TERMIOS_C_ISPEED

28 k_termios

29 #endif

30 #if _HAVE_C_OSPEED && _HAVE_STRUCT_TERMIOS_C_OSPEED

31 k_termios.c_ospeed = termios_p->c_ospeed;

32 #endif

33 memcpy (&k_termios.c_cc[0], &termios_p->c_cc[0],

34 __KERNEL_NCCS * sizeof (cc_t));

35

36 return INLINE_SYSCALL (ioctl, 3, fd, cmd, &k_termios);

37 }

38

39 libc_hidden_def (__tcgetattr)

40 weak_alias (__tcgetattr, tcgetattr)
```

株式会社日昇テクノロジー

初めて接触する場合、コードがやや複雑に見えるかもしれませんが、すべての詳細を完全に理解する必要はありません：

- 7 行目：TCGETS リクエストを送信し、&k_termios ポインタにパラメータを記録するために ioctl を使用しています。

- 10 行目以降：読み取った k_termios の内容を__tcgetattr が渡した termios_p ポインタが指す変数にコピーしています。c_ispeed と c_ospeed は 19 行目と 26 行目で c_cflag を計算して得られます。そのため、tcgetattr ライブラリ関数を使用する場合は、自分で計算して値を割り当てる必要はありませんが、ioctl を使用して設定を読み取る場合は、計算変換を加える必要があります。

glibc ソースコードでは、INLINE_SYSCALL、__glibc_likely、weak_alias などの特殊なマクロやラッパーを多用しています。興味のある読者は、glibc の公式文書でこれらについて学ぶことができます。

第 22 章 I2C 通信

第 22 章では、アプリケーション層で I2C バスを使用して外部デバイスと通信する方法、Linux システムのバスタイプデバイスドライバアーキテクチャの応用について説明します。Linux カーネルのドキュメントの Documentation/i2c ディレクトリには、I2C ドライバに関する非常に詳細な説明があります。この章のサンプルコードのディレクトリは base_linux/i2c です。

22.1 I2C 通信プロトコルの概要

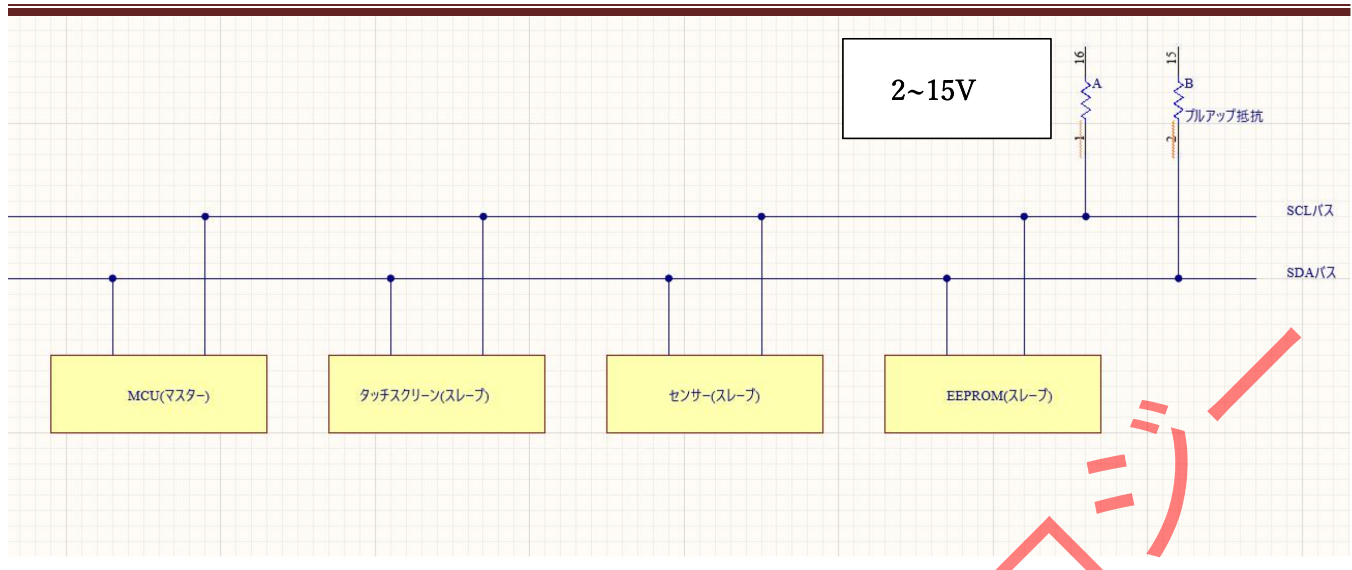
I2C 通信プロトコル（Inter-Integrated Circuit）は、Philips 社によって開発されました。ピン数が少なく、ハードウェアの実装が簡単で、拡張性が高いため、USART、CAN などの通信プロトコルに外部受信装置が不要で、多くの集積回路（IC）間の通信に広く使用されています。

以下では、I2C プロトコルの物理層とプロトコル層について説明します。

22.1.1 I2C の物理層

I2C 通信デバイス間の一般的な接続方法は以下の通りです。

株式会社日昇テクノロジー



その物理層には以下の特徴があります：

- 複数のデバイスをサポートするバスです。「バス」とは、複数のデバイスが共有する信号線を指します。I2C 通信バスでは、複数の I2C 通信デバイスを接続でき、複数の通信マスターと複数の通信スレーブをサポートします。

- I2C バスは、2 本のバスラインのみを使用します。1 本は双方向のシリアルデータライン（SDA）、もう 1 本はシリアルクロックライン（SCL）です。データラインはデータをリストし、クロックラインはデータ送受信の同期に使用されます。

- バスに接続された各デバイスには、独立したデバイスアドレスがあります。マスターはこのアドレスを使用して、異なるデバイス間でアクセスを行います。アドレスは 7 ビットまたは 10 ビットの数値です。

- バスはプルアップ抵抗を介して電源に接続されます。I2C デバイスがアイドル状態の場合、高インピーダンスを出力します。すべてのデバイスがアイドル状態で高インピーダンスを出力すると、プルアップ抵抗によってバスが高電圧に引き上げられます。

- 複数のマスターが同時にバスを使用する場合、データ衝突を防ぐためにアービトレーションを利用して、どのデバイスがバスを占有するかを決定します。

- 3 つの伝送モードがあります：標準モードの伝送速度は 100kbit/s、高速モードは 400kbit/s、高速

株式会社日昇テクノロジー

モードでは最大 3.4Mbit/s に達しますが、現在のところ多くの I2C デバイスは高速モードをサポートしていません。

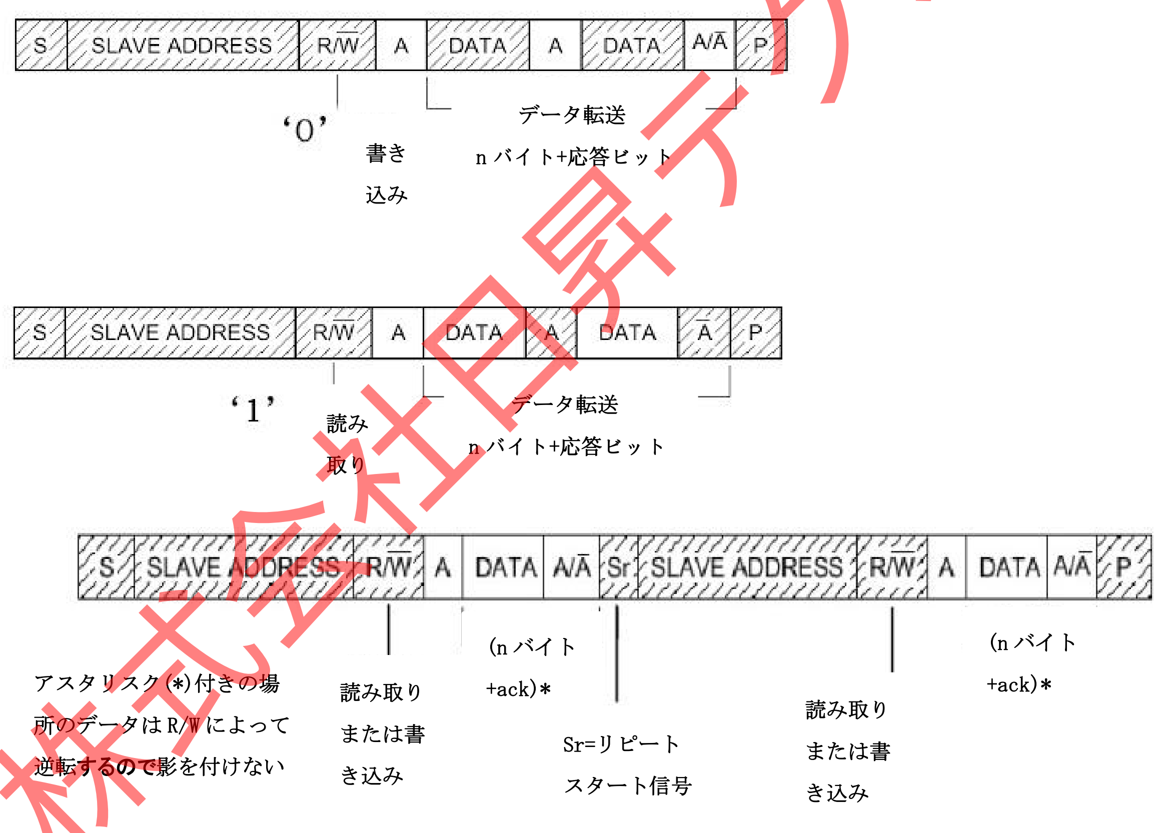
- 同じバスに接続された IC の数は、バスの最大容量 400pF によって制限されます。

22.1.2 プロトコル層

I2C のプロトコルは、通信の開始と停止信号、データの有効性、応答、アービトレーション、クロック同期、アドレスブロードキャストなどを定義しています。

22.1.2.1 I2C の基本的な読み書きプロセス

まず、I2C 通信プロセスの基本構造を見てみましょう。通常、以下の 3 つの方法があります。



図例：

- ：データがマスターからスレーブに伝達されます

- S ：伝送開始信号

株式会社日昇テクノロジー

- SLAVE_ADDRESS: スレーブアドレス

- ：データがスレーブからマスターに伝達されます

- R ：伝送方向選択ビット、1 は読み取り、0 は書き込み

- A ：応答（ACK）または非応答（NACK）信号

- P ：伝送停止信号

これらの図は、マスターとスレーブが通信する際に SDA 線のデータパケットシーケンスを示しています。

1. S は I2C インターフェースを持つホストによって生成されるトランスミッション開始信号を意味し、この信号は I2C バスに接続されているすべてのスレーブデバイスが受信します。

2. スタート信号が生成された後、すべてのスレーブデバイスはホストによって直後にブロードキャストされるスレーブアドレス信号(SLAVE_ADDRESS)を待ちます。I2C バス上では、各デバイスのアドレスはユニークであり、ホストがブロードキャストしたアドレスがあるデバイスのアドレスと一致した場合、そのデバイスが選択され、選択されなかったデバイスは後続のデータ信号を無視します。I2C プロトコルによると、このスレーブアドレスは 7 ビットまたは 10 ビットです。

3. アドレスビットの後には、データ伝送方向を選択するビットがあり、このビットが 0 の場合、後続のデータ伝送方向はホストからスレーブへ、つまりホストからスレーブへの書き込みを意味します。このビットが 1 の場合はその逆で、ホストがスレーブからデータを読み取ります。

4. スレーブが一致するアドレスを受信した後、ホストまたはスレーブは応答(ACK)または非応答(NACK)信号を返します。応答信号を受信した後のみ、ホストはデータの送信または受信を続けることができます。

書き込み方向:

データ伝送方向ビットが「書き込み」方向に設定されている場合、つまり最初の図の場合、アドレスをブロードキャストし、応答信号を受信した後、ホストは正式にスレーブへのデータ(DATA)伝送を開始し、データパケットのサイズは 8 ビットです。ホストは 1 バイトのデータを送信するごとに、スレーブの応答

株式会社日昇テクノロジー

信号(ACK)を待ち、このプロセスを繰り返してスレーブに N 個のデータを伝送できます。この N にはサイズ制限がありません。データ伝送が終了すると、ホストはスレーブに伝送停止信号(P)を送信し、これ以上データを伝送しないことを示します。

読み取り方向:

データ伝送方向ビットが「読み取り」方向に設定されている場合、つまり第二の図の場合、アドレスをブロードキャストし、応答信号を受信した後、スレーブはホストにデータ(DATA)を返し始めます。データパケットのサイズも 8 ビットで、スレーブが 1 つのデータを送信するごとに、ホストの応答信号(ACK)を待ち、このプロセスを繰り返して N 個のデータを返すことができます。この N にもサイズ制限はありません。ホストがデータの受信を停止したい場合は、スレーブに非応答信号(NACK)を返し、スレーブは自動的にデータ伝送を停止します。

複合形式:

基本的な読み書きの他に、I2C 通信でよく使用されるのは複合形式、つまり第三の図の状況です。この伝送プロセスには 2 回のスタート信号(S)があります。一般に、最初の伝送で、ホストは SLAVE_ADDRESS を通じてスレーブデバイスを見つけた後、「データ」を送信し、このデータは通常、スレーブデバイス内のレジスタまたはメモリアドレスを示すために使用されます（これが SLAVE_ADDRESS と異なることに注意してください）；二回目の伝送では、そのアドレスの内容を読み書きします。つまり、最初の通信はスレーブに読み書きのアドレスを伝え、二回目は実際の読み書きの内容です。

以上の通信プロセスに含まれるスタート、ストップ、データの有効性、アドレスとデータの方向、および応答に関する説明は、以下の小節で説明されています。

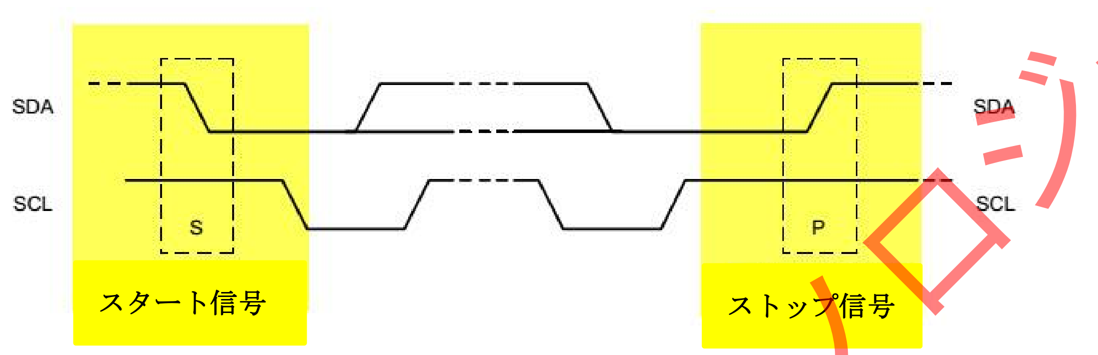
22.1.2.2 通信の開始と停止信号

前述のように、開始(S)と停止(P)信号は 2 種類の特別な状態で、通常はホストが生成します。

株式会社日昇テクノロジー

- SCL ラインが高電圧のときに SDA ラインが高電圧から低電圧に切り替わる場合、これは通信の開始を示します。

- SCL が高電圧のときに SDA ラインが低電圧から高電圧に切り替わる場合、これは通信の停止を示します。

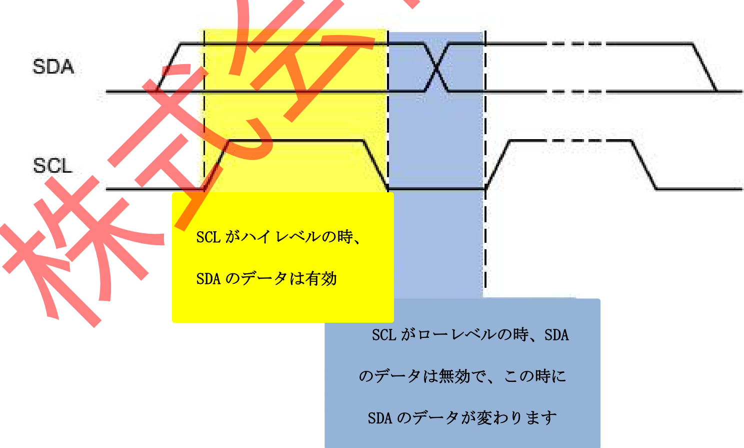


22.1.2.3 データの有効性

I2C は SDA 信号線を使用してデータを伝送し、SCL 信号線を使用してデータを同期します。SDA データ線は SCL の各クロックサイクルで 1 ビットのデータを伝送します。

- 伝送時、SCL が高電圧のとき、SDA が示すデータは有効で、このときの SDA が高電圧であればデータ「1」を、低電圧であればデータ「0」を示します。

- SCL が低電圧のとき、SDA のデータは無効で、通常この時に SDA は電圧の切り替えを行い、次のデータを示す準備をします。



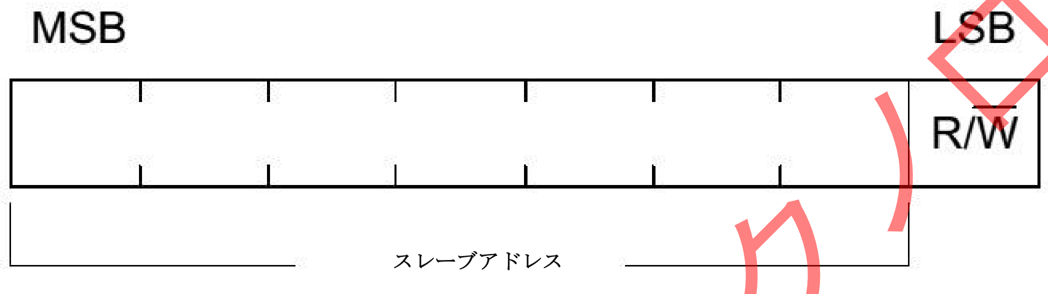
22.1.2.4 アドレスとデータの方向

I2C バス上の各デバイスには独立したアドレスがあり、ホストが通信を開始するときは SDA 信号線を通

株式会社日昇テクノロジー

じてデバイスアドレス(SLAVE_ADDRESS)を送信してスレーブを探します。I2C プロトコルでは、デバイスアドレスは 7 ビットまたは 10 ビットであることが規定されており、実際には 7 ビットアドレスが広く使用されています。

デバイスアドレスの直後にある 1 ビットのデータはデータ伝送の方向を示し、これはデータ方向ビット(R/)で、8 ビット目または 11 ビット目です。データ方向ビットが「1」の場合はホストがスレーブからデータを読み取ることを示し、「0」の場合はホストがスレーブにデータを書き込むことを示します。



- 読み取り方向の場合、ホストは SDA 信号線の制御を解放し、スレーブが SDA 信号線を制御し、ホストは信号を受信します。

- 書き込み方向の場合、SDA はホストによって制御され、スレーブは信号を受信します。

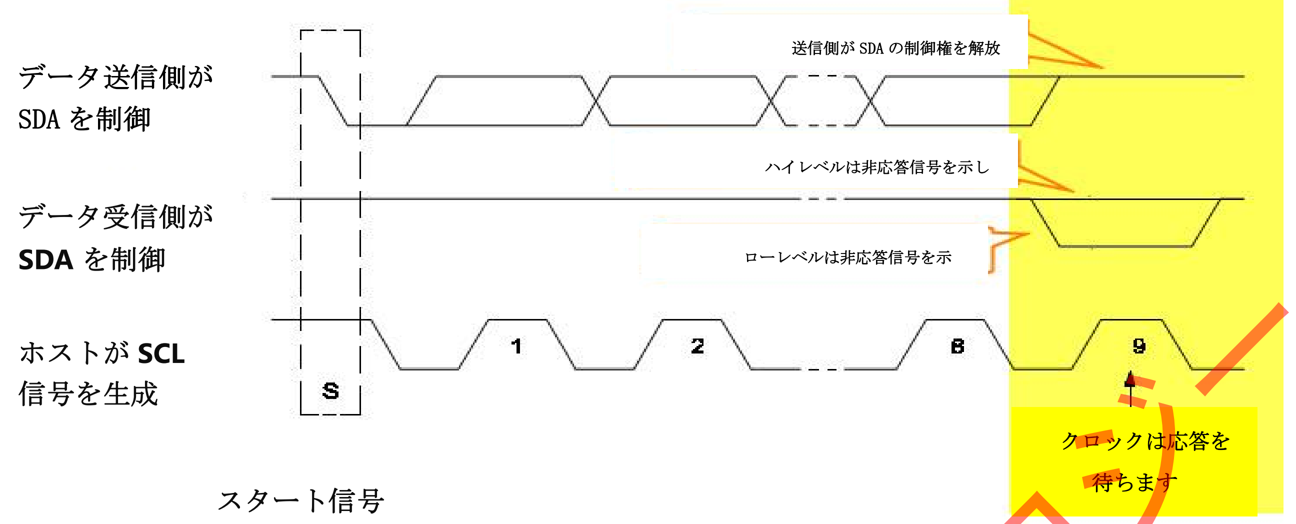
22.1.2.5 応答

I2C のデータおよびアドレス伝送は応答を伴います。応答には「応答(ACK)」と「非応答(NACK)」の 2 種類の信号があります。データ受信側として、デバイス(ホストまたはスレーブに関わらず)が I2C 伝送の 1 バイトのデータまたはアドレスを受信した後：

- 送信側にデータの継続送信を望む場合は、「応答(ACK)」信号を送信し、送信側は次のデータを続けて送信します。

- 受信側がデータ伝送を終了したい場合は、「非応答(NACK)」信号を送信し、送信側がこの信号を受信した後は停止信号を生成し、信号伝送を終了します。

株式会社日昇テクノロジー



伝送時、ホストがクロックを生成し、9 番目のクロック時に、データ送信側は SDA の制御を解放し、データ受信側が SDA を制御します。SDA が高電圧の場合は非応答信号(NACK)を、低電圧の場合は応答信号(ACK)を示します。

この章では、アプリケーションレイヤーで I2C バスを使用して外部デバイスと通信する方法と、Linux システムのバスタイプデバイスドライバーアーキテクチャの応用について説明します。

22.2 本ボードの I2C ピン

この章では、40Pin ピンを備えた LubanCat-RK のボードに焦点を当てます。以下の通りです。

LubanCat-RK のボードには、i2c-3 および i2c-5 の 2 つの I2C デバイスがあります。

I2C	PIN	機能
I2C3-SCL	5	i2c3 のクロック信号線
I2C3-SDA	3	i2c3 のデータ線
I2C5-SCL	28	i2c5 のクロック信号線
I2C5-SDA	27	i2c5 のデータ線

LUBANCAT RK PIN 機能			
機能	PIN		機能
3.3V	1	2	5V
I2C3_SDA	3	4	5V
I2C3_SCL	5	6	GND
GPIO	7	8	UART_TX
GND	9	10	UART_RX

株式会社日昇テクノロジー

GPIO	11	12	PWM
GPIO	13	14	GND
GPIO	15	16	GPIO
3.3V	17	18	GPIO
MOSI	19	20	GND
MISO	21	22	GPIO
SCLK	23	24	CSO
GND	25	26	CSI
I2C5_SDA	27	28	I2C5_SCL
GPIO	29	30	GND
GPIO	31	32	PWM
PWM	33	34	GND
PWM	35	36	GPIO
GPIO	37	38	GPIO
GND	39	40	GPIO

LUBANCAT4 の PIN 設定

					3.3V	1	2	5V					
			I2C5_SDA_M3	47	GPIO1_B7	3	4	5V					
			I2C5_SCL_M3	46	GPIO1_B6	5	6	GND					
	I2C2_SDA_M4	UART6_RX_M1		32	GPIO1_A0	7	8	GPIO4_A3	131	UART0_TX_M2			
					GND	9	10	GPIO4_A4	132	UART0_RX_M2			
	I2C2_SCL_M4	UART6_TX_M1		33	GPIO1_A1	11	12	GPIO1_D6	62	PWM14_M2	I2C8_SCL_M2		
	PDM1_SDI0_M1	PWM3_IR_M3		39	GPIO1_A7	13	14	GND					
	PDM1_SDI1_M1			40	GPIO1_B0	15	16	GPIO3_C1	113			UART7_RX_M1	SPI1_CLK_M1
					3.3V	17	18	GPIO3_D2	122				UART9_RTSN_M2
	PDM1_SDI3_M1	UART4_RX_M2	SPI0_MOSI_M2	42	GPIO1_B2	19	20	GND					
	PDM1_SDI2_M1		SPI0_MISO_M2	41	GPIO1_B1	21	22	GPIO3_D4	124		UART9_RX_M2		
	PDM1_CLK1_M1	UART4_TX_M2	SPI0_CLK_M2	43	GPIO1_B3	23	24	GPIO1_B4	44	SPI0_CS0_M2	UART7_RX_M2	PDM1_CLK0_M1	
					GND	25	26	GPIO1_B5	45	SPI0_CS1_M2	UART7_TX_M2		
			I2C6_SDA_M3	136	GPIO4_B0	27	28	GPIO4_B1	137	I2C6_SCL_M3			
				102	GPIO3_A6	29	30	GND					
SPI1_MOSI_M1		I2C3_SCL_M1		111	GPIO3_B7	31	32	GPIO1_D7	63	PWM15_IR_M3	I2C8_SDA_M2		
UART9_CTSN_M2			PWM10_M2	123	GPIO3_D3	33	34	GND					
		UART9_TX_M2	PWM11_IR_M3	125	GPIO3_D5	35	36	GPIO4_A0	128		SPI0_MISO_M1		UART9_RTSN_M1
SPI1_MISO_M1	UART7_TX_M1	I2C3_SDA_M1		112	GPIO3_C0	37	38	GPIO4_A1	129		SPI0_MOSI_M1		UART9_CTSN_M1
					GND	39	40	GPIO4_A2	130		SPI0_CLK_M1		

22.2.1 IIC 通信インターフェースの有効化

デフォルトでは IIC インターフェースは無効になっており、使用するには有効にする必要があります。

/boot/uEnv/board.txt を開いて（board は使用しているボードの名前です）、i2c 関連のデバイストリープラグインが有効になっているかを確認します。

ここでは I2C-3 を有効にする例を示します。I2C-3 が含まれる行のコメントアウト記号を削除します。

株式会社日昇テクノロジー

```
cat@lubancat:~$ cat /boot/uEnv/uEnvLubanCat1.txt
uname_r=4.19.232
size=0x1000000
#dtb=rk3566-lubancat1.dtb

enable_uboot_overlays=1
#overlay_start

dtoverlay=/dtb/overlay/lubancat-i2c3-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-i2c5-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm10-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm14-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm8-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm9-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-spi3-m1-gpio-cs-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-spi3-m1-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-uart3-m1-overlay.dtbo

#overlay_end

cat@lubancat:~$
```

その後、デバイスを再起動して有効にします。

注意: 電源を直接抜いて再起動する方法では、ファイルが変更されない可能性があります（理由：ファイルがメモリからストレージデバイスにタイムリーに同期されないため。解決策として、ターミナルで「sync」コマンドを入力してから電源を切る）。

22.3 IIC デバイスの確認

以下のコマンドで i2c バスが開始されているかを確認できます。

```
1.  ls /dev/i2c-*
```

```
cat@lubancat:~$ ls /dev/i2c-*
/dev/i2c-0  /dev/i2c-1  /dev/i2c-3  /dev/i2c-6
cat@lubancat:~$
```

22.4 デバイスの接続

注意：例として mpu6050 を使用します。

mpu6050 を i2c-3 バスに接続する方法は以下のとおりです。

```
# ボードと mpu6050 の接続

ボード------mpu6050

3.3V(1) ------ VCC

GND(6) ------ GND
```

株式会社日昇テクノロジー

SCL(5) ------ SCL

SDA(3) ------ SDA

22.5 IIC 第三者ツール - i2c-tools

i2c-tools パッケージは、システムの I2C バスをデバッグするための非常に便利なツールを提供します。

以下のコマンドをボードのターミナルで直接実行してインストールできます：

```
1. sudo apt install i2c-tools wget gcc -y
```

インストール後、i2cdetect、i2cdump、i2cset、i2cget といったコマンドが使用できます。これらは、I2C バス上のデバイスをスキャンしたり、指定されたデバイスのレジスタを読み書きするために使用されます。

22.5.1 i2cdetect その他のコマンド

- i2cdetect -F i2cbus：i2c バスの機能を照会します。引数 i2cbus は i2c バスを示します。

- i2cdetect -V：ソフトウェアバージョンを表示します。

- i2cdetect -l：現在のシステムにいくつの i2c バスがあるかを検出します。

22.5.1.1 i2cget コマンド

i2cget：指定された IIC デバイスの特定のレジスタの値を読み取ります。

関連コマンド構文：

i2cget [-f] [-y] i2cbus chip-address [data-address [mode]]

パラメータ説明：

- パラメータ f：デバイスへの強制アクセス。

- パラメータ y：対話モードをオフにします。このパラメータを使用すると、警告メッセージが表示されません。

- パラメータ i2cbus：i2c バスの番号を指定します。

株式会社日昇テクノロジー

- パラメータ chip-address：i2c デバイスアドレス。

- パラメータ data-address：デバイスのレジスタアドレス。

- パラメータ mode：i2cdump コマンドを参照してください。

22.5.1.2 i2cset コマンド

i2cset：指定された IIC デバイスの特定のレジスタに値を書き込みます。

関連コマンド構文：

i2cset [-f] [-y] [-m mask] [-r] i2cbus chip-address data-address [value] ...[mode]

パラメータ説明：

- パラメータ f：デバイスへの強制アクセス。

- パラメータ y：対話モードをオフにします。このパラメータを使用すると、警告メッセージが表示されません。

- パラメータ m：マスク。

- パラメータ r：書き込み後にレジスタ値を即時に読み取り、書き込まれた値と比較します。

- パラメータ i2cbus：i2c バスの番号を指定します。

- パラメータ chip-address：i2c デバイスアドレス。

- パラメータ data-address：デバイスのレジスタアドレス。

- パラメータ value：レジスタに書き込む値。

- パラメータ mode：i2cdump コマンドを参照してください。

22.5.1.3 i2cdump コマンド

i2cdump：指定されたデバイスのすべてのレジスタの値を読み取ります。

関連コマンド構文：

i2cdump [-f] [-r first-last] [-y] i2cbus address [mode [bank [bankreg]]]

株式会社日昇テクノロジー

パラメータ説明：

- パラメータ r：レジスタ範囲を指定し、first から last までのエリアのみをスキャンします。

- パラメータ f：デバイスへの強制アクセス。

- パラメータ y：人間と機械の対話モードをオフにします。

・パラメータ i2cbus：I2C バスの番号を指定

・パラメータ address：デバイスのアドレスを指定

・パラメータ mode：読み取りサイズを指定。b, w, s, i のいずれかで、それぞれバイト、ワード、SMBus ブロック、I2C ブロックに対応

i2cdump -V：ソフトウェアのバージョン番号を表示

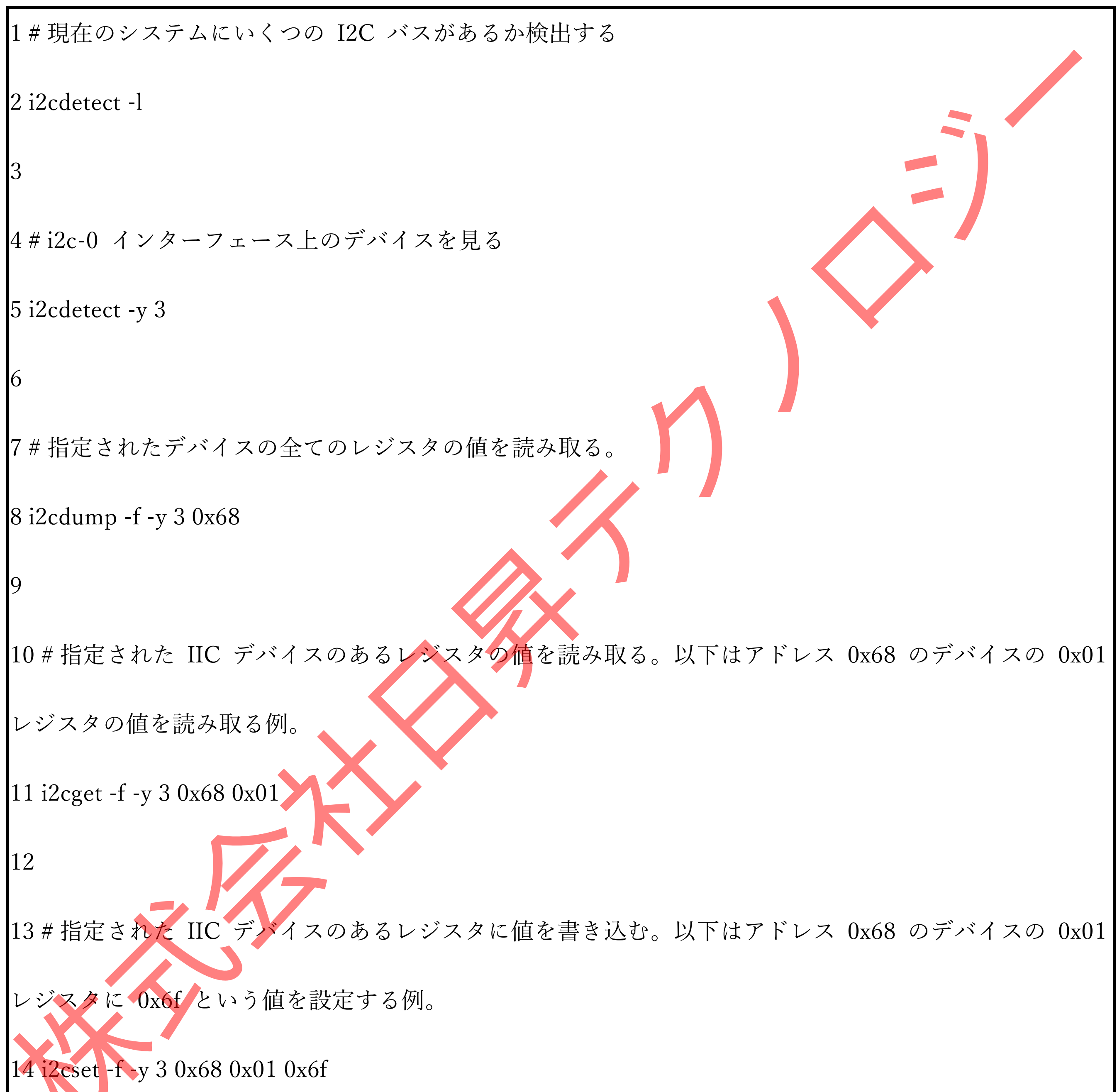
次に、i2c-3 にマウントされているデバイスの状況を確認し、出力内容は以下の通りです：

```
1 root@lubancat:~# i2cdetect -a 3
2 WARNING! This program can confuse your I2C bus, cause data loss and worse!
3 I will probe file /dev/i2c-3.
4 I will probe address range 0x00-0x7f.
5 Continue? [Y/n] y
6 0 1 2 3 4 5 6 7 8 9 a b c d e f
7 00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
8 10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
9 20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10 30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
11 40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
12 50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
13 60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
```

株式会社日昇テクノロジー

```
14 70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

15 root@lubancat:~# ^C
```

「68」は MPU6050 のデバイスアドレスであり、以下は一般的に使用されるいくつかのコマンドです。



```
1 # 現在のシステムにいくつの I2C バスがあるか検出する

2 i2cdetect -l

3

4 # i2c-0 インターフェース上のデバイスを見る

5 i2cdetect -y 3

6

7 # 指定されたデバイスの全てのレジスタの値を読み取る。

8 i2cdump -f -y 3 0x68

9

10 # 指定された IIC デバイスのあるレジスタの値を読み取る。以下はアドレス 0x68 のデバイスの 0x01
レジスタの値を読み取る例。

11 i2cget -f -y 3 0x68 0x01

12

13 # 指定された IIC デバイスのあるレジスタに値を書き込む。以下はアドレス 0x68 のデバイスの 0x01
レジスタに 0x6f という値を設定する例。

14 i2cset -f -y 3 0x68 0x01 0x6f
```

株式会社日昇テクノロジー

22.6 ジャイロスコープセンサーデータの読み取り実験

22.6.1 実験説明

このチュートリアルでは、IIC インターフェースを通じてジャイロスコープ(MPU6050)の生データを読み

取ります。この実験では i2c-3 を例にしますが、i2c-5 の操作も i2c-3 と同様です。もちろん、mpu6050 モジュールを持っていない場合でも、mpu6050 の操作方法を学ぶことで、希望する i2c デバイスを操作することができます。テストプログラムでは、大体 1 秒ごとに MPU6050 の

生データを読み取って表示します。

IIC デバイスファイルを確認し、IIC 3 インターフェイスが有効になっていることを確認します。

```
1. root@lubancat:~# ls /dev/i2c-*
2. /dev/i2c-0 /dev/i2c-1 /dev/i2c-3 /dev/i2c-6
3. root@lubancat:~#
```

ここで「i2c-3」は MPU6050 が使用する IIC 3 インターフェイスバスです。

22.6.2 ioctl 関数

アプリケーションを書く際には、ioctl 関数を使用して i2c 関連の設定を行う必要があります。関数のプロトタイプは以下の通りです。

```
1. #include <sys/ioctl.h>
2.
3. int ioctl(int fd, unsigned long request, ...);
```

端末で request の値として一般的に使用されるものには以下があります。

I2C_RETRIES	ACK を受信できない時の再試行回数を設定します。デフォルトは 1 です。
I2C_TIMEOUT	タイムアウトの jiffies を設定します。
I2C_SLAVE	スレーブアドレスを設定します。
I2C_SLAVE_FORCE	スレーブアドレスを強制的に設定します。
I2C_TENBIT	アドレスの長さを選択します。0 は 7 ビットアドレス、非 0 は 10 ビットアドレスです。

株式会社日昇テクノロジー

22.6.3 アプリケーションの作成

ioctl 関連のパラメータを基に、i2c 関連のインターフェイス関数を作成し、mpu6050 の生データを読み取るプログラムは以下の通りです。

リスト 1: base_linux/i2c/i2c_mpu6050/i2c_mpu6050.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/stat.h>
7 #include <fcntl.h>
8 #include <linux/i2c.h>
9 #include <linux/i2c-dev.h>
10 #include <sys/ioctl.h>
11
12 /* レジスタアドレス*/
13 #define SMPLRT_DIV 0x19
14 #define PWR_MGMT_1 0x6B
15 #define CONFIG 0x1A
16 #define ACCEL_CONFIG 0x1C
17
18 #define ACCEL_XOUT_H 0x3B
```

株式会社日昇テクノロジー

```
19 #define ACCEL_XOUT_L 0x3C

20 #define ACCEL_YOUT_H 0x3D

21 #define ACCEL_YOUT_L 0x3E

22 #define ACCEL_ZOUT_H 0x3F

23 #define ACCEL_ZOUT_L 0x40

24 #define GYRO_XOUT_H 0x43

25 #define GYRO_XOUT_L 0x44

26 #define GYRO_YOUT_H 0x45

27 #define GYRO_YOUT_L 0x46

28 #define GYRO_ZOUT_H 0x47

29 #define GYRO_ZOUT_L 0x48

30

31 //スレーブアドレス MPU6050 アドレス

32 #define Address 0x68

33

34 // MPU6050 操作関連関数

35 static int mpu6050_init(int fd,uint8_t addr);

36 static int i2c_write(int fd, uint8_t addr,uint8_t reg,uint8_t val);

37 static int i2c_read(int fd, uint8_t addr,uint8_t reg,uint8_t * val);

38 static short GetData(int fd,uint8_t addr,unsigned char REG_Address);

39

40 int main(int argc,char *argv[] )

41 {
```

株式会社日昇テクノロジー

```
42 int fd;

43 fd = I2C_SLAVE;

44

45 if(argc < 2){

46 printf("Wrong use !!!!¥n");

47 printf("Usage: %s [dev]¥n",argv[0]);

48 return -1;

49 }

50

51 fd = open(argv[1], O_RDWR); // open file and enable read and write

52 if (fd < 0){

53 printf("Can't open %s ¥n",argv[1]); // open i2c dev file fail

54 exit(1);

55 }

56

57 // MPU6050 の初期化

58 mpu6050_init(fd,Address);

59 while(1){

60 usleep(1000 * 10);

61 printf("ACCE_X:%6d¥n ", GetData(fd,Address,ACCEL_XOUT_H));

62 usleep(1000 * 10);

63 printf("ACCE_Y:%6d¥n ", GetData(fd,Address,ACCEL_YOUT_H));

64 usleep(1000 * 10);
```

株式会社日昇テクノロジー

```
65 printf("ACCE_Z:%6d¥n ", GetData(fd,Address,ACCEL_ZOUT_H));

66 usleep(1000 * 10);

67 printf("GYRO_X:%6d¥n ", GetData(fd,Address,GYRO_XOUT_H));

68 usleep(1000 * 10);

69 printf("GYRO_Y:%6d¥n ", GetData(fd,Address,GYRO_YOUT_H));

70 usleep(1000 * 10);

71 printf("GYRO_Z:%6d¥n¥n ", GetData(fd,Address,GYRO_ZOUT_H));

72 sleep(1);

73 }

74

75 close(fd);

76

77 return 0;

78 }

79

80 static int mpu6050_init(int fd,uint8_t addr)

81 {

82 i2c_write(fd, addr,PWR_MGMT_1,0x00); //電源管理を設定、0x00、通常起動

83 i2c_write(fd, addr,SMPLRT_DIV,0x07); // MPU6050 の出力分割を設定し、サンプリングを設定

84 i2c_write(fd, addr,CONFIG,0x06); //デジタルローパスフィルターとフレーム同期ピンを設定

85 i2c_write(fd, addr,ACCEL_CONFIG,0x01); //測定範囲と X、Y、Z 軸の加速度自己テストを設定

86

87 return 0;
```

株式会社日昇テクノロジー

```
88 }

89

90 static int i2c_write(int fd, uint8_t addr,uint8_t reg,uint8_t val)

91 {

92 int retries;

93 uint8_t data[2];

94

95 data[0] = reg;

96 data[1] = val;

97

98 //アドレス長を設定：0 は 7 ビットアドレス

99 ioctl(fd,I2C_TENBIT,0);

100

101 //スレーブアドレスを設定

102 if (ioctl(fd,I2C_SLAVE,addr) < 0){

103 printf("fail to set i2c device slave address!¥n");

104 close(fd);

105 return -1;

106 }

107

108 // ACK を受信できない場合の再試行回数を設定

109 ioctl(fd,I2C_RETRIES,5);
```

株式会社日昇テクノロジー

```
110

111 if (write(fd, data, 2) == 2){

112 return 0;

113 }

114 else{

115 return -1;

116 }

117

118 }

119

120 static int i2c_read(int fd, uint8_t addr,uint8_t reg,uint8_t * val)

121 {

122 int retries;

123

124 //アドレス長を設定：0 は 7 ビットアドレス

125 ioctl(fd,I2C_TENBIT,0);

126

127 //スレーブアドレスを設定

128 if (ioctl(fd,I2C_SLAVE,addr) < 0){

129 printf("fail to set i2c device slave address!¥n");

130 close(fd);

131 return -1;
```

株式会社日昇テクノロジー

```
132 }
133
134 // ACK  を受信できない場合の再試行回数を設定
135 ioctl(fd,I2C_RETRIES,5);
136
137 if (write(fd, &reg, 1) == 1){
138 if (read(fd, val, 1) == 1){
139 return 0;
140 }
141 }
142 else{
143 return -1;
144 }
145 }
146
147 static short GetData(int fd,uint8_t addr,unsigned char REG_Address)
148 {
149 char H, L;
150
151 i2c_read(fd, addr,REG_Address, &H);
152 usleep(1000);
153 i2c_read(fd, addr,REG_Address + 1, &L);
```

株式会社日昇テクノロジー

```
154 return (H << 8) +L;

155 }
```

22.6.3.1 コンパイル

方法 1：

```
1. # コンパイル

2. make
```

方法 2：

```
1. # コンパイル

2. gcc i2c_mpu6050.c -o mpu6050
```

22.6.3.2 実行

```
1. # 実行

2. sudo ./mpu6050 /dev/i2c-3

3. # /dev/i2c-3 を/dev/i2c-5 に変更することもできます。これはあなたのハードウェアに依存します。
```

効果は以下の通りです。

```
root@lubancat:~# gcc -o mpu6050 mpu6050.c
root@lubancat:~# ./mpu6050
ACCE_X: -3989
 ACCE_Y:   -248
 ACCE_Z:   2089
 GYRO_X:   -329        1
 GYRO_Y:      4
 GYRO_Z:    120

 ACCE_X: -7728
 ACCE_Y:  -284
 ACCE_Z:   2185
 GYRO_X:  -333         2
 GYRO_Y:     22
 GYRO_Z:   -102

 ACCE_X: -7704
 ACCE_Y:  -287
 ACCE_Z:   2210
 GYRO_X:  -260         3
 GYRO_Y:     40
 GYRO_Z:    -47
```

株式会社日昇テクノロジー

22.6.4 コード分析

リスト 2: i2c 書き込み関数

```
1 static int i2c_write(int fd, uint8_t addr,uint8_t reg,uint8_t val)
2 {
3 int retries;
4 uint8_t data[2];
5
6 data[0] = reg;
7 data[1] = val;
8
9 //アドレスの長さを設定：0 は 7 ビットアドレス
10 ioctl(fd,I2C_TENBIT,0);
11
12 //スレーブアドレスを設定
13 if (ioctl(fd,I2C_SLAVE,addr) < 0){
14 printf("スレーブアドレスの設定に失敗しました！¥n");
15 close(fd);
16 return -1;
17 }
18
19 //ACK が受信できない場合の再試行回数を設定
20 ioctl(fd,I2C_RETRIES,5);
```

株式会社日昇テクノロジー

```
21
22 if (write(fd, data, 2) == 2){
23 return 0;
24 }
25 else{
26 return -1;
27 }
28
29 }
```

- MPU6050 書き込み関数。成功すると 0 を返し、失敗すると-1 を返します。関数のパラメータは 4 つあります。1. fd: ファイルディスクリプタ。2. addr: i2c デバイスのアドレス。3. reg: 書き込む MPU6050 のレジスタアドレス。4. val: 書き込む値。

- 第 2-6 行で、レジスタとその変更値を配列に入れ、送信時に一括で送信します。

- 第 8-16 行で、i2c アドレスの長さを設定し、送信する i2c アドレスを設定します。

- 第 18-29 行で、i2c の再送回数を設定し、配列をデバイスに送信します。

リスト 3: i2c 読み取り関数

- MPU6050 読み取り関数。成功すると 0 を返し、失敗すると-1 を返します。関数のパラメータは 4 つあります。1. fd: ファイルディスクリプタ。2. addr: i2c デバイスのアドレス。3. reg: 読み取る MPU6050 のレジスタア

- MPU6050 読み取り関数。成功した場合は 0 を返し、失敗した場合は-1 を返します。関数の引数は合計 4 つあります。1. fd: ファイル記述子。2. addr: i2c デバイスのアドレス。3. REG_Address: 読み取りたい MPU6050 のレジスタアドレス。

株式会社日昇テクノロジー

- 第 5-7 行で、i2c アドレスとレジスタを使用して該当する値を取得します。データは 16 ビットなので、i2c の転送形式が 8 ビットであるため、2 回読み取る必要があります。

- 第 8 行で、16 ビットの数値を返します。

リスト 5: MPU6050 の初期化

- MPU6050 の初期化関数。初期化のプロセスは実際にはデバイスを開いて設定パラメータを書き込むことです。addr は i2c で送信するアドレスで、第 3 行から第 7 行まで、MPU6050 のサンプリング精度や範囲などを設定します。

リスト 6: メイン関数

- 第 5-16 行で、i2c バスインターフェースを開き、バスを取得します。

- 第 19 行で、MPU6050 を初期化します。

- 第 20-34 行で、データを取得して表示します。

22.7 OLED ディスプレイ実験

この実験では、以下のような OLED ディスプレイを使用します。

この実験の付随するプログラムは、IIC_1 通信インターフェースの OLED ディスプレイに適応しており、解像度は 128*64 です。他の OLED ディスプレイを使用する場合は、IIC インターフェースをサポートしていることを確認する必要があります。

株式会社日昇テクノロジー

22.7.1 ハードウェア接続

- 画面———ボードカード

- VCC ———3.3v

- GND ———GND

- SDA ———SDA

- SCL ———SCL

22.7.2 コンパイル&実行

リスト 7: コードの場所

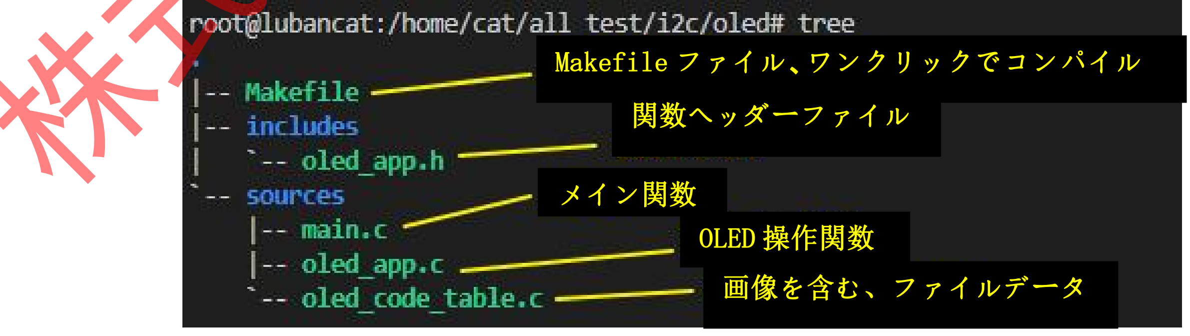
```
1 base_linux/i2c/i2c_oled

1 # コンパイル

2 make

3

4 # 実行

5 ./i2c_oled /dev/i2c-3
```

実物展示図:

株式会社日昇テクノロジー

22.7.3 コード分析

コードの構造は以下の通りです。



oled_app.c の関数は以下の通りです。

株式会社日昇テクノロジー

```
static int i2c_write(int fd, unsigned char addr,unsigned char reg,unsigned char val);
void OLED_Init(int fd,unsigned char addr);
void OLED_Fill(unsigned char addr,unsigned char fill_Data);
void OLED_CLS(unsigned char addr);
void oled_set_Pos(unsigned char addr,unsigned char x, unsigned char y);
void OLED_ShowStr(unsigned char addr,unsigned char x, unsigned char y, unsigned char ch[], unsigned char TextSize);
void OLED_ShowCN(unsigned char addr,unsigned char x, unsigned char y, unsigned char N);
void OLED_DrawBMP(unsigned char addr,unsigned char x0,unsigned char y0,unsigned char x1,unsigned char y1,unsigned char BMP[]);

#endif
```

関数を書く、関数の内容は前述と全く同じ

リスト 9: oled_app.h

```
1 #define OLED_COMMEND_ADDR 0x00

2 #define OLED_DATA_ADDR 0x40
```

reg = 0x00 は、命令を送信することを意味し、より正確には OLED の設定パラメータや制御パラメータです。reg = 0x40 は、データを送信することを意味します。val は、送信する内容を指定します。関数の実装は 2 部分に分かれています。第一部分では、関数の入口パラメータをローカル変数 data[]配列に保存し、後で送信を実行するために便利です。ioctl 関数を呼び出して IIC のスレーブアドレス、つまり oled のアドレスを設定します。oled が自分に対応するアドレスを検出すると、通信が可能になります。oled のアドレスは以下のように定義されています：

```
1 #define Address 0x3c // 0R 抵抗を調整することにより、スクリーンは 0x78 と 0x7A の 2 つのアドレ
```

スを持つことができますが、デフォルトは 0x78 です。

OLED ディスプレイはデフォルトで IIC のスレーブアドレスが 0X78 で、抵抗を調整することで 0X7A に設定できます。ここでのアドレスは 8 ビットのアドレスで、最後のビットは読み書きを示します。しかし、ここで送信するのは IIC デバイスの 7 ビットアドレスで、上記のコードで示されているように、マクロ定義で設定した IIC アドレスは 0x78 を左に 1 ビットシフトして得られます。

全画面塗りつぶし関数 OLED_Fill は、全画面塗りつぶし関数とクリア関数が似ており、前者は各ピクセルを点灯させ、後者は各ピクセルを消灭します。プログラムでは、前者は 0xff を書き込み、後者は 0x00 を書き込みます。関数の実装は以下の通りです：

株式会社日昇テクノロジー

リスト 11: OLED 填充-oled_app.c

```
1 void OLED_Fill(unsigned char addr,unsigned char fill_Data) //全画面塗りつぶし
2 {
3 unsigned char m, n;
4 for (m = 0; m < 8; m++)
5 {
6 i2c_write(fd, addr, OLED_COMMEND_ADDR, 0xb0 + m); //ページ 0-ページ 1
7 i2c_write(fd, addr, OLED_COMMEND_ADDR, 0x00); //低い列の開始アドレス
8 i2c_write(fd, addr, OLED_COMMEND_ADDR, 0x10); //高い列の開始アドレス
9
10 for (n = 0; n < 128; n++)
11 {
12 // WriteDat(fill_Data);
13 i2c_write(fd, addr, OLED_DATA_ADDR, fill_Data); //高い列の開始アドレス
14 }
15 }
16 }
```

全画面塗りつぶし関数は 2 つのネストされた for ループで構成されており、デフォルトでは 8 行のピクセルが「一行」を構成しています。oled ディスプレイは一列に 64 ピクセルがありますので、外側のループは 0 から 7 まで取ることができます。内側のループは「一行」を設定するために使用され、oled の一行は 128 ピクセルがありますので、内側のループは 128 回実行されます。i2c_write 関数は、一度に 8 ビットのデータを書き込むことで 8 ピクセルの状態をリストします。正確には、内側のループが完了する（128 回ループする）と、実際には 8 行のピクセルが書き込まれます。

株式会社日昇テクノロジー

oled 文字列表示関数 OLED_ShowStr は、F6x8 と F8X16 の 2 種類のフォントのみを表示できます。F6x8 は、各文字の幅が 6 ピクセル、高さが 8 ピクセルで、F6x8 と F8X16 はいずれもフォント生成ソフトウェアを手動で生成したものです。興味があれば、他のフォントも自分で作成することができます。文字列表示関数は以下の通りです：

リスト 12: OLED で英文を表示する

```
1 void OLED_ShowStr(unsigned char addr,unsigned char x, unsigned char y,␣
,→unsigned char ch[], unsigned char TextSize)
2 {
3 unsigned char c = 0,i = 0,j = 0;
4 switch(TextSize)
5 {
6 case 1:
7 {
8 while(ch[j] != '¥0')
9 {
10 c = ch[j] - 32;
11 if(x > 126)
12 {
13 x = 0;
14 y++;
15 }
16 oled_set_Pos(addr,x,y);
```

株式会社日昇テクノロジー

```
17 for(i=0;i<6;i++)

18 i2c_write(fd, addr,OLED_DATA_ADDR,

,→F6x8[c][i]);

19 x += 6;

20 j++;

21 }

22 }break;

23 case 2:

24 {

25 while(ch[j] != '¥0')

26 {

27 c = ch[j] - 32;

28 if(x > 120)

29 {

30 x = 0;

31 y++;

32 }

33 oled_set_Pos(addr,x,y);

34 for(i=0;i<8;i++)

35 i2c_write(fd,addr, OLED_DATA_ADDR,

,→F8X16[c*16+i]);

36 oled_set_Pos(addr,x,y+1);
```

株式会社日昇テクノロジー

```
37 for(i=0;i<8;i++)

38 i2c_write(fd, addr,OLED_DATA_ADDR,F8X16[c*16+i+8]);

39 x += 8;

40 j++;

41 }

42 }break;

43 }

44 }
```

関数には合計 5 つのパラメータがあります。x、y は文字の表示位置を設定するために使用されます。文字コードの違いにより、x と y の取りうる範囲は固定ではありません。F8X16 を例にとると、文字を一つ書き込むごとに x は 8 増加する必要があり、OLED の解像度から x の最大値は 128-8-1（ピクセルはゼロから始まる）となります。文字を一つ書き込むごとに Y は 16 ピクセル増加する必要があります。8 行のピクセルが「一行」を形成することを考えると、実際の y は 0 から 7 まで取ることができますが、文字を一つ書き込むごとに y が 16 ピクセル増加するため、y は 0 から 6 まで取ることができます。

ch[]は、書き込む文字列を指定します。TextSize は文字サイズを指定し、この関数は現在 2 種類のフォントのみをサポートしています。TextSize = 1 の場合は F6x8 フォント形式を使用し、TextSize = 2 の場合は F8X16 フォント形式を使用します。

他の表示関数も同様ですのでここでは詳細は述べません。

株式会社日昇テクノロジー

リスト 13: oled/sources/main.c

```
1 #include <stdio.h>

2 #include <stdlib.h>

3 #include <unistd.h>

4 #include <sys/ioctl.h>

5 #include <sys/types.h>

6 #include <sys/stat.h>

7 #include <fcntl.h>

8 #include <sys/select.h>

9 #include <sys/time.h>

10 #include <errno.h>

11 #include <linux/i2c.h>

12 #include <linux/i2c-dev.h>

13 #include <sys/ioctl.h>

14

15 #include "oled_app.h"

16

17 #define アドレス 0x3c

18

19 extern int fd;

20 extern const unsigned char BMP1[];

21
```

株式会社日昇テクノロジー

```
22 /* メイン関数 */
23 int main(int argc, char *argv[])
24 {
25 int i = 0; // ループ用
26
27 fd = open("/dev/i2c-3", O_RDWR); // ファイルを開いて読み書き可能にする
28
29 if (fd < 0){
30 perror("/dev/i2c-3 を開けません ¥n"); // i2c デバイスファイルを開けなかった場合のエラー
31 exit(1);
32 }
33
34 OLED_Init(fd, アドレス); // OLED を初期化
35 usleep(1000 * 100);
36 OLED_Fill(アドレス, 0xff); // 全画面を塗りつぶし
37
38 while (1)
39 {
40 OLED_Fill(アドレス, 0xff); // 全画面を塗りつぶし
41 sleep(1);
42
43 OLED_CLS(アドレス); // 画面をクリア
```

株式会社日昇テクノロジー

```
44 sleep(1);

45

46 OLED_ShowStr(アドレス, 0, 3, (unsigned char *)"Wildfire Tech", 1); // 6*8 の文字をテスト

47 OLED_ShowStr(アドレス, 0, 4, (unsigned char *)"Hello wildfire", 2); // 8*16 の文字をテスト

48 sleep(1);

49 OLED_CLS(アドレス); // 画面をクリア

50

51 for (i = 0; i < 4; i++)

52 {

53 OLED_ShowCN(アドレス, 22 + i * 16, 0, i); // 中文表示をテスト

54 }

55 sleep(1);

56 OLED_CLS(アドレス); // 画面をクリア

57

58 OLED_DrawBMP(アドレス, 0, 0, 128, 8, (unsigned char *)BMP1); // BMP ビットマップ表示をテスト

59 sleep(1);

60 OLED_CLS(アドレス); // 画面をクリア

61 }

62

63 close(fd);

64 }
```

メイン関数の実装は比較的シンプルで、前述の関数を直接呼び出すだけです。while(1)の無限ループの中で、全画面塗りつぶし、画面クリア、英字表示、漢字表示、画像テスト関数を順に実行します。

株式会社日昇テクノロジー

第 23 章 SPI 通信

この章では、アプリケーション層で SPI バスを使用して外部デバイスと通信する方法について説明し、Linux システムのバスタイプデバイスドライバアーキテクチャの適用について説明します。これは前章の I2C バス操作方法と非常に似ており、比較学習が可能です。

Linux カーネルのドキュメントにある Documentation/SPI ディレクトリには、SPI ドライバに関する非常に詳細な説明があります。

この章のサンプルコードのディレクトリは以下のとおりです：base_linux/spi

23.1 SPI 通信プロトコルの紹介

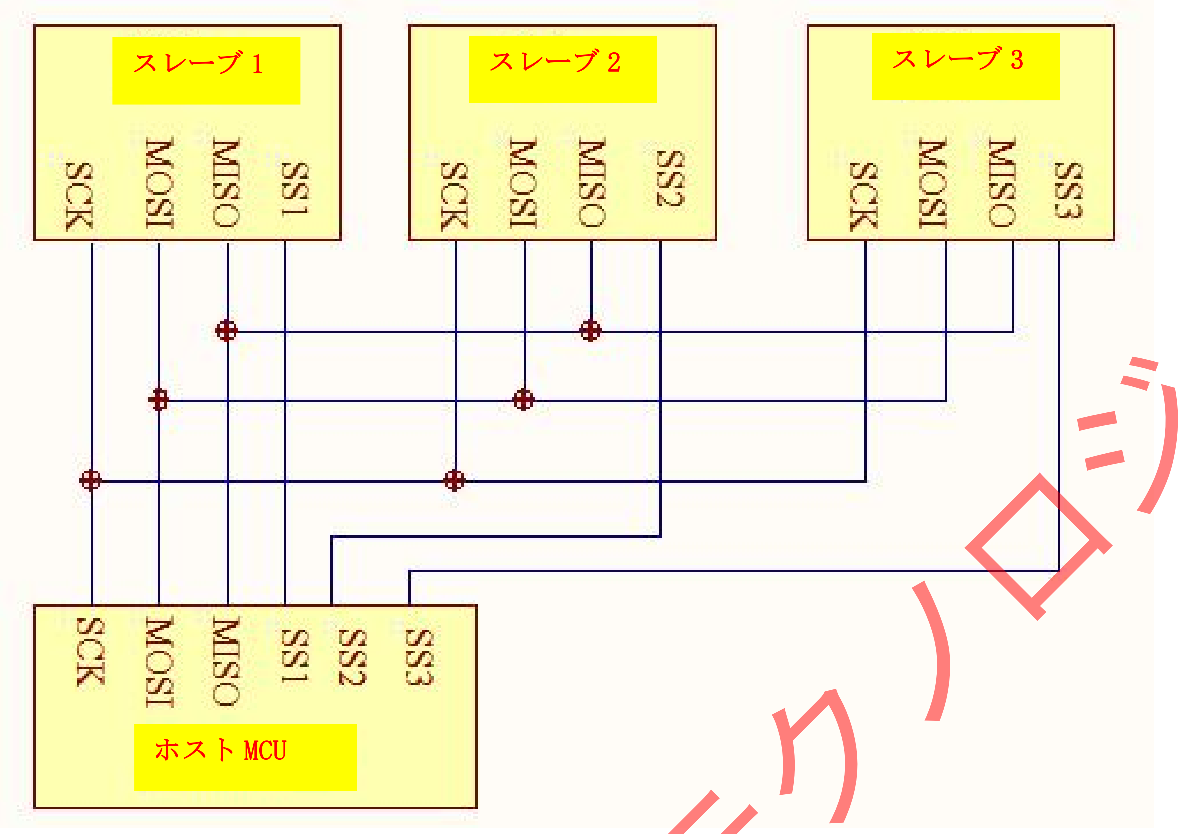
SPI プロトコルは、モトローラ社によって提案された通信プロトコル（Serial Peripheral Interface）であり、シリアル周辺機器インターフェースとは、高速な全二重通信バスです。ADC、LCD などのデバイスと MCU 間で広く使用されており、通信速度が要求される場合に学習します。この章では、I2C 章と比較して読むことで、二つの通信バスの違いを理解することができます。以下、SPI プロトコルの物理層とプロトコル層について説明します。

23.1.1 SPI の物理層

SPI 通信デバイス間の一般的な接続方法は以下の図に示されます。

スレーブスレーブ

株式会社日昇テクノロジー



SPI 通信には、3 つのバスとチップセレクトラインが使用されます。3 つのバスはそれぞれ SCK、MOSI、MISO で、チップセレクトラインは、以下の役割を紹介します：

(1) (Slave Select) ：スレーブデバイス選択信号線、通常はチップセレクト信号線とも呼ばれ、NSS、CS とも呼ばれます。以下では NSS とリスト記します。複数の SPI スレーブデバイスが SPI マスターに接続されている場合、デバイスの他の信号線 SCK、MOSI、MISO は同じ SPI バスに並列接続されます。つまり、いくつかのスレーブデバイスがあっても、これら 3 つのバスを共有します。一方、各スレーブデバイスには独立した NSS 信号線があり、NSS 信号線はマスターのピンを専有します。つまり、スレーブデバイスがいくつあるかに応じて、同じ数のチップセレクト信号線があります。I2C プロトコルではデバイスアドレスを使用してバス上のデバイスを選択し、通信しますが、SPI プロトコルにはデバイスアドレスがありません。NSS 信号線を使用してアドレス指定を行います。マスターがスレーブデバイスを選択する場合、そのスレーブデバイスの NSS 信号線をローレベルに設定します。つまり、そのスレーブデバイスが選択され、SPI 通信が開始されます。したがって、SPI 通信は NSS

株式会社日昇テクノロジー

ラインがローレベルに設定されたときに開始され、NSS ラインがハイレベルに引き上げられたときに終了します。

(2) SCK（Serial Clock）：クロック信号線で、通信データの同期に使用されます。通信の速度はこれによって決まり、異なるデバイスがサポートする最高のクロック周波数は異なります。例えば、RT1052 の SPI クロック周波数の最大値は fpclk/2 です。二つのデバイス間で通信する場合、通信速度は低速デバイスに制限されます。

(3) MOSI（Master Output, Slave Input）：マスター出力/スレーブ入力ピン。マスターのデータはこの信号線から出力され、スレーブはこの信号線からマスターが送信したデータを読み取ります。つまり、この線上のデータの方向はマスターからスレーブへです。

(4) MISO（Master Input, Slave Output）：マスター入力/スレーブ出力ピン。マスターはこの信号線からデータを読み取り、スレーブのデータはこの信号線からマスターへ出力されます。つまり、この線上のデータの方向はスレーブからマスターへです。

23.1.2 プロトコル層

I2C と同様に、SPI プロトコルは通信の開始と停止の信号、データの有効性、クロック同期などを定義しています。

23.1.2.1 SPI の基本的な通信プロセス

まず、SPI 通信のタイミングを見てみましょう、以下の図に示されています。

株式会社日昇テクノロジー

これはマスターの通信タイミングです。NSS、SCK、MOSI の信号はすべてマスターによって制御され、生成されますが、MISO の信号はスレーブによって生成され、マスターはこの信号線を通じてスレーブのデータを読み取ります。MOSI と MISO の信号は、NSS がローレベルのときにのみ有効です。SCK の各クロックサイクルで MOSI と MISO は 1 ビットのデータを転送します。

上記の通信プロセスに含まれる各信号の分解は以下のとおりです：

23.1.2.2 通信の開始と停止の信号

上図の番号付き部分で、NSS 信号線は高から低に変わり、SPI 通信の開始信号です。NSS は各スレーブが独自に持つ信号線で、スレーブが自身の NSS 線で開始信号を検出すると、自分がマスターに選ばれたことを知り、マスターとの通信の準備を始めます。図の番号付き部分で、NSS 信号が低から高に変わるのは、SPI 通信の停止信号であり、この通信が終了し、スレーブの選択状態が解除されます。

23.1.2.3 データの有効性

SPI は、データの転送に MOSI および MISO 信号線を使用し、SCK 信号線を使用してデータを同期します。MOSI および MISO データ線は、SCK の各クロックサイクルで 1 ビットのデータを転送し、デー

株式会社日昇テクノロジー

タの入出力は同時に行われます。データの転送では、MSB が先行するか LSB が先行するかは厳密には定められていませんが、二つの SPI 通信デバイス間で同じ規則を使用する必要があります。一般的には、上図の MSB 先行モードが採用されます。

図の番号付き部分を観察すると、MOSI および MISO のデータは SCK の上昇エッジの間に変化し、SCK の下降エッジでサンプリングされます。つまり、SCK の下降エッジの時点で、MOSI および MISO のデータが有効であり、高電位はデータ「1」を、低電位はデータ「0」をリストします。他の時点では、データは無効で、MOSI および MISO は次のデータをリストするための準備をします。

SPI のデータ転送は、8 ビットまたは 16 ビット単位で行うことができ、転送する単位数に制限はありません。

23.1.2.4 CPOL/CPHA および通信モード

上述した図のタイミングは、SPI の中の一つの通信モードに過ぎません。SPI には四つの通信モードがあり、それらの主な違いは、バスがアイドル状態の時の SCK のクロック状態およびデータサンプリングのタイミングです。説明を容易にするために、ここでは「クロック極性 CPOL」と「クロック相位 CPHA」の概念を導入します。

クロック極性 CPOL は、SPI 通信デバイスがアイドル状態にある時の SCK 信号線の電位信号を指します（つまり、SPI 通信が開始する前、NSS 線がハイレベルの時の SCK の状態）。CPOL=0 の時、SCK はアイドル状態でローレベルです。CPOL=1 の時はその逆です。

クロック相位 CPHA は、データのサンプリングタイミングを指します。CPHA=0 の時、MOSI または MISO データ線上の信号は、SCK クロック線の「奇数エッジ」でサンプリングされます。CPHA=1 の時は、データ線は SCK の「偶数エッジ」でサンプリングされます。

以下の図のように：

株式会社日昇テクノロジー

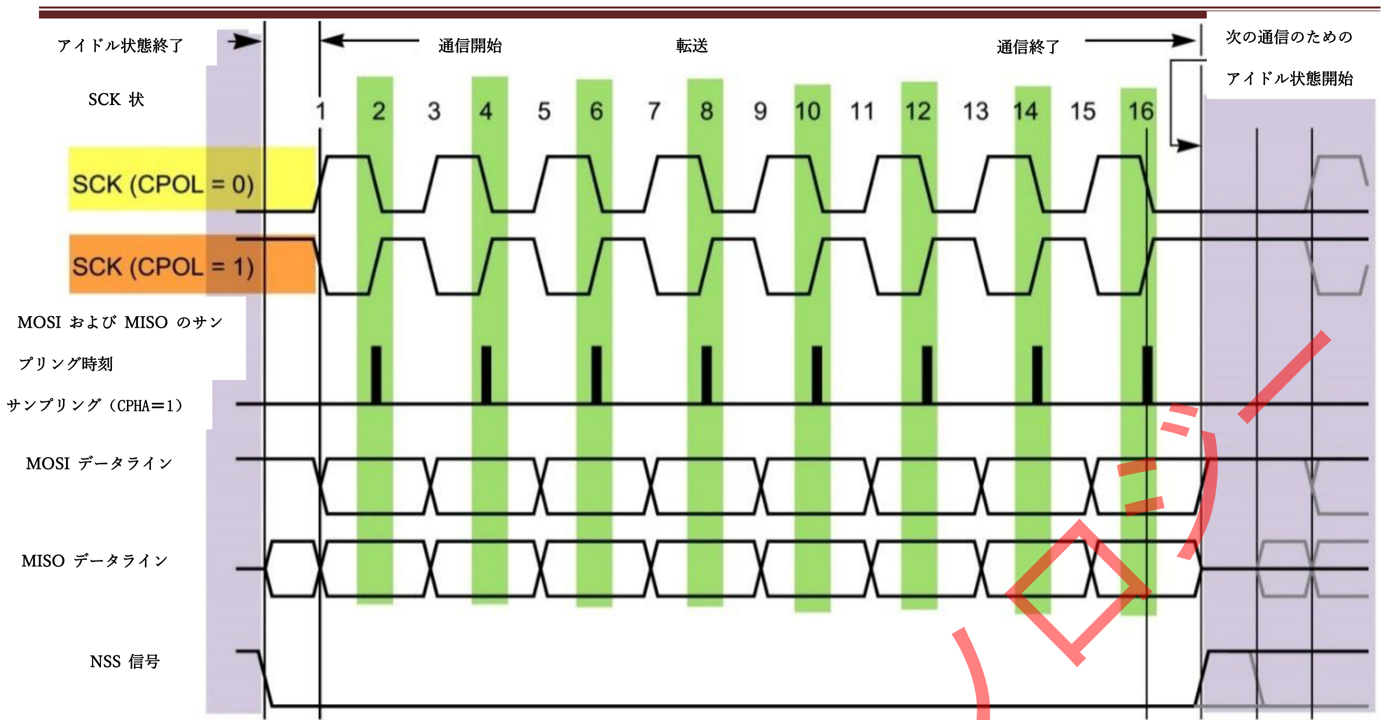
CPHA=0 のタイミング図を分析してみましょう。まず、SCK がアイドル状態での電平に基づき、2 つのケースに分けられます。SCK 信号線がアイドル状態で低電位の場合は CPOL=0、アイドル状態で高電位の場合は CPOL=1 です。

CPOL=0 でも CPOL=1 でも、設定されたクロック相位 CPHA=0 であるため、図で見ることができるように、サンプリング時刻は SCK の奇数エッジになります。CPOL=0 の時は、クロックの奇数エッジが上昇エッジであり、CPOL=1 の時は、クロックの奇数エッジが下降エッジになります。したがって、SPI のサンプリング時刻は上昇/下降エッジによって決定されるわけではありません。MOSI と MISO のデータ線の有効信号は SCK の奇数エッジで変化せず、データ信号は SCK の奇数エッジでサンプリングされ、非サンプリング時に MOSI と MISO の有効信号が切り替わります。

同様に、CPHA=1 の場合、CPOL の影響を受けず、データ信号は SCK の偶数エッジでサンプリングされます。

以下の図のように：

株式会社日昇テクノロジー



CPOL および CPHA の異なる状態により、SPI は 4 つのモードに分かれます。マスターとスレーブは同じモードで動作する必要があり、実際には「モード 0」と「モード 3」がよく使用されます。

リスト SPI の 4 つのモード

SPI モード	CPOL	CPHA	アイドル時 SCK クロック	サンプリング時刻
0	0	0	ローレベル	奇数エッジ
1	0	1	ローレベル	偶数エッジ
2	1	0	ハイレベル	奇数エッジ
3	1	1	ハイレベル	偶数エッジ

23.1.3 拡張 SPI プロトコル（シングル/デュアル/クアッド/オクタル SPI）

上述したのは、クラシック SPI プロトコルの内容であり、標準 SPI プロトコル（Standard SPI）またはシングル SPI プロトコル（Single SPI）とも呼ばれます。ここでの「シングル」とは、SPI プロトコルでデータ送信に MOSI を 1 本、データ受信に MISO を 1 本使用することを指します。

より高速な通信ニーズに対応するため、半導体メーカーは SPI プロトコルを拡張し、デュアル/クアッド/オクタル SPI プロトコルを主に開発しました。標準 SPI プロトコル（Single SPI）に加えて、これら 4

株式会社日昇テクノロジー

つのプロトコルの主な違いは、データラインの数と通信方式です。具体的には以下のリストに示します。

リスト 4 つの SPI プロトコルの違い

プロトコル	データラインの数と機能	通信方式
シングル SPI（標準 SPI）	1 本送信、1 本受信	フルデュプレックス
デュアル SPI（2 線式 SPI）	送受信に 2 本のデータラインを共用	ハーフデュプレックス
クアッド SPI（4 線式 SPI）	送受信に 4 本のデータラインを共用	ハーフデュプレックス
オクタル SPI（8 線式 SPI）	送受信に 8 本のデータラインを共用	ハーフデュプレックス

拡張された 3 つの SPI プロトコルはすべてハーフデュプレックスの通信方式です。つまり、データラインは送受信に時間分割して使用されます。例えば、標準 SPI（Single SPI）とデュアル SPI（Dual SPI）は 2 本のデータラインを持ちますが、標準 SPI（Single SPI）は 1 本が送信専用、もう 1 本が受信専用です、すなわちフルデュプレックスです。一方、デュアル SPI（Dual SPI）の 2 本の線は送受信機能を持ちますが、同時に送信または受信のみ可能で、ハーフデュプレックスです。クアッド SPI（Quad SPI）とオクタル SPI（Octal SPI）はデュアル SPI（Dual SPI）に似ていますが、データラインの数が異なります。

23.1.3.1 SDR と DDR モード

拡張された SPI プロトコルには、SDR モード（シングルデータレート Single Data Rate）と DDR モード（ダブルデータレート Double Data Rate）が追加されました。例えば、標準 SPI プロトコルの SDR モードでは、SCK の単一エッジでデータ転送が行われ、つまり 1 つの SCK クロックで 1 ビットのデータが転送されます。一方、DDR モードでは、SCK の上昇エッジと下降エッジの両方でデータ転送が行われ、つまり 1 つの SCK クロックで 2 ビットのデータが転送され、転送速度が倍増します。

23.2 SPI 関連データ構造と ioctl 関数

アプリケーションプログラムを作成するには、spi_ioc_transfer 構造体を使用する必要があります。以下に示します。

株式会社日昇テクノロジー

リスト 1: linux/spi/spidev.h

```
1 struct spi_ioc_transfer {

2 __u64 tx_buf; // 送信データバッファ

3 __u64 rx_buf; // 受信データバッファ

5 __u32 len; // データの長さ

6 __u32 speed_hz; // 通信速度

7

8 __u16 delay_usecs; // 2 つの spi_ioc_transfer 間の遅延、マイクロ秒

9 __u8 bits_per_word; // ワードあたりのビット数

10 __u8 cs_change; // チップセレクトの選択解除

11 __u8 tx_nbits; // 送信時のデータ幅（マルチラインモード）

12 __u8 rx_nbits; // 受信時のデータ幅（マルチラインモード）

13 __u16 pad;

14

15 };
```

アプリケーションプログラムを作成する際には、ioctl 関数を使用して SPI 関連の設定を行う必要があります。その関数プロトタイプは以下の通りです。

```
1 #include <sys/ioctl.h>

2

3 int ioctl(int fd, unsigned long request, ...);
```

ここでの端末リクエストの値として一般的に使用されるのは以下のいくつかです。

SPI_IOC_RD_MODE32 SPI	モードの読み取りを設定します（上記の SPI の 4 つのモードのリス

株式会社日昇テクノロジー

	トに対応、SPI_MODE_x)
SPI_IOC_WR_MODE32	SPI モードの書き込みを設定します（上記の SPI の 4 つのモードのリストに対応、SPI_MODE_x)
SPI_IOC_RD_LSB_FIRST	SPI のデータ読み取りモードを設定します（LSB が先、1 を返す）
SPI_IOC_WR_LSB_FIRST	SPI のデータ書き込みモードを設定します（0:MSB、非 0：LSB）
SPI_IOC_RD_BITS_PER_WORD	SPI の読み取り時のワード長を設定します
SPI_IOC_WR_BITS_PER_WORD	SPI の書き込み時のワード長を設定します
SPI_IOC_RD_MAX_SPEED_HZ	SPI デバイスの最大通信周波数を読み取ります。
SPI_IOC_WR_MAX_SPEED_HZ	SPI デバイスの最大通信速度を設定します
SPI_IOC_MESSAGE(N)	一度に双方向/複数回の読み書き操作を行います

ヒント: SPI の読み取りと書き込みは異なるパラメータを設定できます。

23.3 LubanCat ボードの SPI インターフェース

この章では、40Pin ピンを持つ LubanCat-RK のボードを中心に説明します。

40 ピンの中には 1 組の SPI インターフェース SCK、MOSI、MISO があり、2 つのデフォルトチップセレクトシグナル CS0、CS1 があります。

SPI3 のピン配置は以下のリストの通りです。

SPI	ピン	機能
MOSI	19	マスター出力/スレーブ入力
MISO	21	マスター入力/スレーブ出力
CLOCK	23	クロック信号線
CS0	24	チップセレクト信号線 0
CS1	26	チップセレクト信号線 1

spidev3.0 が CS0 を制御し、spidev3.1 が CS1 を制御します。

以下の図のように：

機能	PIN		機能
3.3V	1	2	5V
I2C3_SDA	3	4	5V
I2C3_SCL	5	6	GND
GPIO	7	8	UART_TX
GND	9	10	UART_RX

株式会社日昇テクノロジー

GPIO	11	12	PWM
GPIO	13	14	GND
GPIO	15	16	GPIO
3.3V	17	18	GPIO
MOSI	19	20	GND
MISO	21	22	GPIO
SCLK	23	24	CSO
GND	25	26	CSI
I2C5_SDA	27	28	I2C5_SCL
GPIO	29	30	GND
GPIO	31	32	PWM
PWM	33	34	GND
PWM	35	36	GPIO
GPIO	37	38	GPIO
GND	39	40	GPIO

対応する実物の 40 ピンインターフェース

23.3.1 SPI 通信インターフェースの有効化

SPI インターフェースはデフォルトでオフになっており、使用するには有効化する必要があります。SPI を有効にするには、SPI 通信ピンと spi-cs ピンの 2 つを設定する必要があります。

/boot/uEnv/board.txt を開いて、SPI 関連のデバイスツリープラグインが有効になっているか確認できます。

ファイルを編集し、spi と spi-cs に関する 2 行のコメントアウト記号を削除します。以下の図のように：

株式会社日昇テクノロジー

```
cat@lubancat:~$ cat /boot/uEnv/uEnvLubanCat1.txt
uname_r=4.19.232
size=0x1000000
#dtb=rk3566-lubancat1.dtb

enable_uboot_overlays=1
#overlay_start

#dtoverlay=/dtb/overlay/lubancat-i2c3-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-i2c5-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm10-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm14-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm8-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm9-m0-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-spi3-m1-gpio-cs-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-spi3-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-uart3-m1-overlay.dtbo

#overlay_end

cat@lubancat:~$
```

その後、デバイスを再起動して有効にします。

注意: 電源を直接抜いて再起動する場合、ファイルが正しく変更されない可能性があります（理由：ファイルがメモリからストレージデバイスにタイムリーに同期されなかったため。解決策：ターミナルで「sync」を入力してから電源を切る）。

23.3.2 SPI デバイスの確認

SPI デバイスツリープラグインを有効にしてボードを再起動した後、SPI デバイスファイルを通じて SPI ドライバが正常にロードされたかどうかを判断できます。SPI_3 に対応するデバイスファイルは spidev3.0 と spidev3.1 です。

以下のように表示されます。

```
1 root@lubancat:~# ls /dev/spi*
2 /dev/spidev3.0 /dev/spidev3.1
3 root@lubancat:~#
```

spidev3.0 と spidev3.1 の違いはチップセレクト信号が異なることで、spidev3.0 は CS0 を使用し、spidev3.1 は CS1 を使用します。

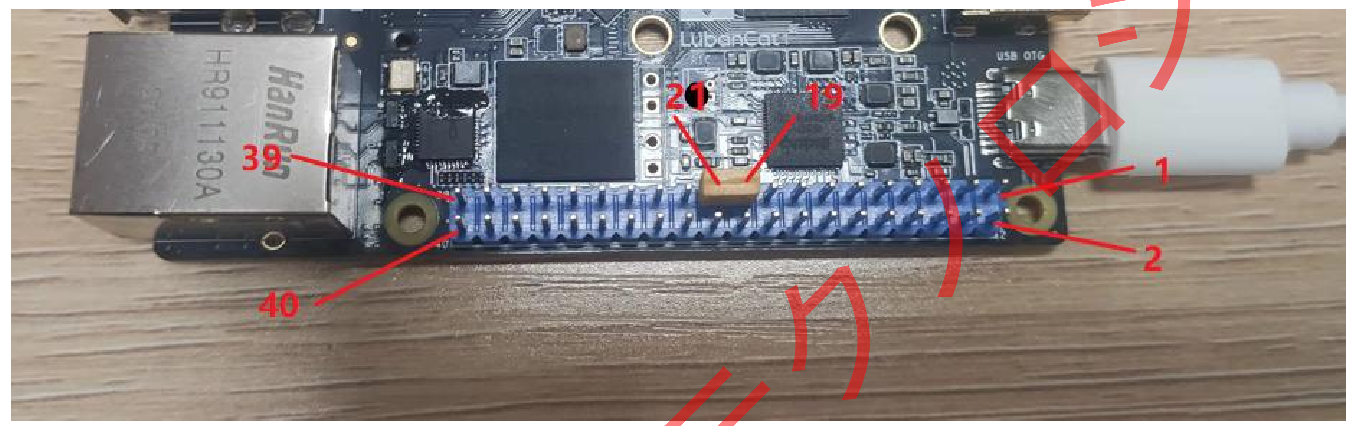
株式会社日昇テクノロジー

23.4 SPI ループバック通信テスト実験

23.4.1 ハードウェア接続：

SPI3 の MISO と MOSI ピン（ボード上の 19 と 21）をジャンパーキャップでショートさせるだけです。

以下の図のように：



23.4.2 プログラムの作成

リスト 2: base_linux/spi/spi_selftest/spi_selftest.c

```
1 #include <sys/ioctl.h>

2 #include <sys/types.h>

3 #include <sys/stat.h>

4 #include <fcntl.h>

5 #include <unistd.h>

6 #include <stdio.h>

7 #include <stdlib.h>

8 #include <stdint.h>

9 #include <linux/spi/spidev.h>

10
```

株式会社日昇テクノロジー

```
11 #define SPI_DEV_PATH "/dev/spidev3.0"

12

13 /* SPI 受信・送信バッファ */

14 unsigned char tx_buffer[100] = "hello the world !";

15 unsigned char rx_buffer[100];

16

17

18 int fd; // SPI 制御ピンのデバイスファイル記述子

19 static unsigned mode = SPI_MODE_2; // SPI 動作モードを保存するために使用

20 static uint8_t bits = 8; // 受信・送信データのビット数

21 static uint32_t speed = 10000000; // 送信速度

22 static uint16_t delay; // 遅延時間を保存

23

24 void transfer(int fd, uint8_t const *tx, uint8_t const *rx, size_t len)

25 {

26 int ret;

27

28 struct spi_ioc_transfer tr = {

29 .tx_buf = (unsigned long)tx,

30 .rx_buf = (unsigned long)rx,

31 .len = len,

32 .delay_usecs = delay,

33 .speed_hz = speed,
```

株式会社日昇テクノロジー

```
34 .bits_per_word = bits,

35 .tx_nbits = 1,

36 .rx_nbits = 1

37 };

38

39 ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);

40

41 if (ret < 1)

42 printf("spi メッセージを送信できません¥n");

43 }

44

45 void spi_init(void)

46 {

47 int ret = 0;

48 // SPI デバイスを開く

49 fd = open(SPI_DEV_PATH, O_RDWR);

50 if (fd < 0)

51 printf("%s を開けません¥n", SPI_DEV_PATH);

52

53 // spi mode SPI 動作モードの設定

54 ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);

55 if (ret == -1)

56 printf("spi モードを設定できません¥n");
```

株式会社日昇テクノロジー

```
57

58 // bits per word 1 バイトのビット数の設定

59 ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);

60 if (ret == -1)

61 printf("ビット数を設定できません¥n");
```

```
62

63 // max speed hz SPI の最高動作周波数の設定

64 ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);

65 if (ret == -1)

66 printf("最高速度を設定できません¥n");

67

68 // 印刷

69 printf("spi モード: 0x%x¥n", mode);

70 printf("ビット数: %d¥n", bits);

71 printf("最高速度: %d Hz (%d KHz)¥n", speed, speed / 1000);

72 }

73

74 int main(int argc, char *argv[])
```

株式会社日昇テクノロジー

23.4.3 コンパイル＆実行

23.4.3.1 コンパイル

方法 1:

```
1 make
```

方法 2：

```
1 gcc -o spi_selftest spi_selftest.c
```

23.4.3.2 実行

```
1 # 管理者権限で実行する必要があります
2 sudo ./spi_selftest
```

以下の図のように:

```
cat@lubancat:~/all_test/spi/spi_test/sources$ sudo ./a.out
spi mode: 0x2
bits per word: 8
max speed: 10000000 Hz (10000 KHz)
tx_buffer:
 hello the world !
 rx_buffer:
hello the world !
```

mosi と miso を接続していない場合、以下の状況が発生します

```
cat@lubancat:~/all_test/spi/spi_test/sources$ sudo ./a.out
spi mode: 0x2
bits per word: 8
max speed: 10000000 Hz (10000 KHz)
tx_buffer:
 hello the world !
 rx_buffer:
 ������������������������������������������������������������������������������������������
```

23.4.4 コード分析

SPI コードは前章の IIC と非常に似ています。まず SPI 対応のデバイスファイルを開き、次に設定パラメータを書き込むことで、SPI の初期化が完了します。その後の具体的な機能実装は、対応する API 関数を呼び出すだけです。SPI の初期化コードは以下の通りです：

23.4.4.1 SPI 初期化コード

リスト 3: SPI 初期化コード

```
1 void spi_init(void)
2 {
```

株式会社日昇テクノロジー

```
3 int ret = 0;
4 // SPI デバイスを開く
5 fd = open(SPI_DEV_PATH, O_RDWR);
6 if (fd < 0)
7 printf("can't open %s¥n", SPI_DEV_PATH);
8
9 // spi mode SPI 動作モードの設定
10 ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);
11 if (ret == -1)
12 printf("spi モードを設定できません¥n");
13
14 // bits per word 1 バイトあたりのビット数を設定
15 ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
16 if (ret == -1)
17 printf("ビット数を設定できません¥n");
18
19 // max speed hz SPI 通信の最大速度を設定
20 ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
21 if (ret == -1)
22 printf("最大速度を設定できません¥n");
23
24 // 印刷
25 printf("spi モード: 0x%x¥n", mode);
```

株式会社日昇テクノロジー

26 printf("ビット数: %d¥n", bits);

27 printf("最大速度: %d Hz (%d KHz)¥n", speed, speed / 1000);

28 }

29

コードを組み合わせて簡単に説明すると：

- 3-6 行目、SPI バスのデバイスファイルを開きます。デバイスファイルのパスは「/dev/spidev3.0」で、開けない場合はパスが正しいかデバイスファイルが存在するかをまず確認する必要があります。開く方法は「O_RDWR」で、SPI ループバック通信テストを行うために読み書き方式で開きます。

- 9-12 行目、SPI の動作モードを設定します。以前の説明によると、SPI はフェーズとポラリティの違いによって 4 種類の動作モードに分かれています。ここでは 4 種類の動作モードは SPI_MODE_x (x = 0、1、2、3)です。これはループバックテストなので、どの動作モードでも設定できます。SPI の読み書き動作モードを分けて設定できることに注意してください。

- 14-17 行目、SPI 通信中の 1 バイトあたりのビット数を設定します。通常は 8 ビットで設定します。ここでも、読み書きは分けて設定されます。

- 19-22 行目、SPI 通信のボーレートを設定します。ここでは 1M に設定されています。

- 以上の四つの初期化を経て、SPI は通信できるようになります。

23.4.4.2 SPI 送信関数

SPI は SPI 送信構造体 spi_ioc_transfer を利用して送信を実現します。プログラム中で、送信したいデータと必要な設定パラメータを構造体に入力し、次に ioctl 関数を呼び出して送信を実行します。送信関数は以下の通りです：

株式会社日昇テクノロジー

リスト 4: SPI 送信関数

```
1 void transfer(int fd, uint8_t const *tx, uint8_t const *rx, size_t len)
2 {
3 int ret;
4
5 struct spi_ioc_transfer tr = {
6 .tx_buf = (unsigned long)tx,
7 .rx_buf = (unsigned long)rx,
8 .len = len,
9 .delay_usecs = delay,
10 .speed_hz = speed,
11 .bits_per_word = bits,
12 .tx_nbits = 1,
13 .rx_nbits = 1
14 };
15
16 ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
```

株式会社日昇テクノロジー

17

18 if (ret < 1)

- 関数には四つの引数があります。fd は、SPI デバイスファイルを開いた際に得られる SPI デバイスファイルのディスクリプタ、tx は送信するデータのアドレス、rx は双方向転送の場合に受信バッファのアドレスを指定するために使用します。len は、この転送で送信するデータの長さをバイト単位で指定します。

- 関数の実装は非常にシンプルです。以下のコードで説明します：

- 5 行目から 14 行目では、SPI 転送構造体を定義し初期化しています。SPI 転送構造体の完全な定義は以下の通りです：

- コメントを組み合わせると、理解は簡単です。簡単に説明すると以下の通りです：

1. tx_buf と rx_buf は、それぞれ送信バッファと受信バッファのアドレスを指します。データ型は「__u64」で、64 ビットシステムと互換性があります。64 ビットまたは 32 ビットはシステムによって自動的に処理されるため、気にする必要はありません。

2. len は一回の転送でのデータ長です。speed_hz は SPI 通信のビットレートを指定します。

3. delay_usecs はゼロでなければ、二回の転送間の遅延時間を設定するために使用されます。

4. bits_per_word はバイト長を指定します、つまり一バイトが何ビットを占めるかを指定します。

5. cs_change は選択解除で、真に設定されている場合、次の転送前に現在の SPI デバイスの選択を解除し、片選択を更新します。

6. tx_nbits は「書き込み」データ幅を指定します。SPI は 1、2、4 ビット幅をサポートしています。特別なデータ要件がない場合、一般的に 1 または 0 に設定されます（0 に設定するとデフォルトの幅、つまり幅 1 を使用することを意味します）。

7. pad は使用していないパラメータで、設定する必要はありません。

- 16 行目で、ioctl を呼び出して送信を実行します。引数 fd は SPI デバイスファイルのディスクリプ

株式会社日昇テクノロジー

タ、引数 SPI_IOC_MESSAGE(1)は転送回数を指定するために使用されます。ここでは転送構造体 tr を一つだけ定義して初期化しているので、転送回数は 1 です。tr は最初の部分で設定した転送構造体変数です。

23.4.4.3 メイン関数:

SPI の初期化関数と送信関数があれば、SPI のループバックテストは非常に簡単です。SPI を初期化してから送信関数を呼び出し、転送結果を以下のように表示する

SPI MAIN 関数のリスト：

```
1. #include <sys/ioctl.h>
2. #include <sys/types.h>
3. #include <sys/stat.h>
4. #include <fcntl.h>
5. #include <unistd.h>
6. #include <stdio.h>
7. #include <stdlib.h>
8. #include <stdint.h>
9. #include <linux/spi/spidev.h>
10
11 #define SPI_DEV_PATH "/dev/spidev3.0"
12
13 /* SPI 受信・送信バッファ */
14 unsigned char tx_buffer[100] = "hello the world !";
15 unsigned char rx_buffer[100];
```

株式会社日昇テクノロジー

```
16

17

18 int fd; // SPI 制御ピンのデバイスファイルディスクリプタ

19 static unsigned mode = SPI_MODE_2; // SPI 動作モードを保存するための変数

20 static uint8_t bits = 8; // 受信・送信データのビット数

21 static uint32_t speed = 10000000; // 送信速度

22 static uint16_t delay; // 遅延時間を保存する変数

23

24 void transfer(int fd, uint8_t const *tx, uint8_t const *rx, size_t len)

25 {

26 int ret;

27

28 struct spi_ioc_transfer tr = {

29 .tx_buf = (unsigned long)tx,

30 .rx_buf = (unsigned long)rx,

31 .len = len,

32 .delay_usecs = delay,

33 .speed_hz = speed,

34 .bits_per_word = bits,

35 .tx_nbits = 1,

36 .rx_nbits = 1

37 };

38
```

株式会社日昇テクノロジー

```
39 ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);

40

41 if (ret < 1)

42 printf("SPI メッセージを送信できません¥n");

43 }

44

45 void spi_init(void)

46 {

47 int ret = 0;

48 // SPI デバイスを開く

49 fd = open(SPI_DEV_PATH, O_RDWR);

50 if (fd < 0)

51 printf("%s を開けません¥n", SPI_DEV_PATH);

52

53 // SPI モードの設定

54 ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);

55 if (ret == -1)

56 printf("SPI モードを設定できません¥n");

57

58 // ビット数の設定

59 ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);

60 if (ret == -1)

61 printf("ビット数を設定できません¥n");
```

株式会社日昇テクノロジー

```
62

63 // SPI の最高動作周波数の設定

64 ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);

65 if (ret == -1)

66 printf("最高速度を設定できません¥n");

67

68 // 設定内容の表示

69 printf("spi mode: 0x%x¥n", mode);

70 printf("bits per word: %d¥n", bits);

71 printf("max speed: %d Hz (%d KHz)¥n", speed, speed / 1000);

72 }

73

74 int main(int argc, char *argv[])

75 {

76 /* SPI の初期化 */

77 spi_init();

78

79 /* 送信を実行 */

80 transfer(fd, tx_buffer, rx_buffer, sizeof(tx_buffer));

81

82 /* tx_buffer と rx_buffer を表示 */

83 printf("tx_buffer: ¥n %s¥n ", tx_buffer);

84 printf("rx_buffer: ¥n %s¥n ", rx_buffer);
```

株式会社日昇テクノロジー

85

86 close(fd);

87 return 0;

88 }

メイン関数では、関数 spi_init を呼び出して SPI を初期化し、関数 transfer を呼び出して送信を実行します。最後に tx_buffer と rx_buffer の内容をそれぞれ表示します。通常、プログラムを実行すると、コントロールターミナルで tx_buffer と rx_buffer の内容が一致していることが確認できます。

23.5 SPI_OLED ディスプレイ実験

前のセクションでは SPI ループバック通信を実装しました。このセクションでは SPI をドライブして SPI_OLED ディスプレイを表示することを実現します。このセクションは前章の IIC で IIC_OLED をドライブするのと非常に似ており、送信関数が異なるだけです。

リスト 6: コード位置

```
1 # コード位置
2 base_linux/spi_oled
```

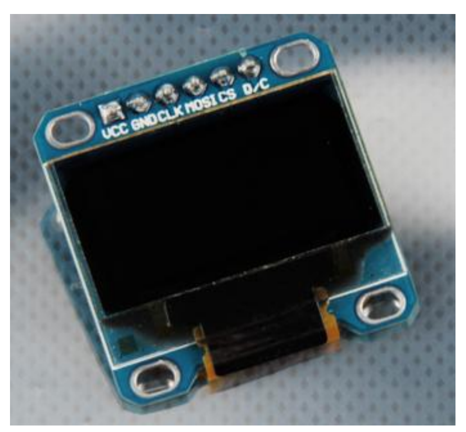
リスト 7: コード構造

```
1 .
2 |-- Makefile
3 |-- includes
4 | `-- spi_oled_app.h
5 `-- sources
6 |-- main.c
7 |-- oled_code_table.c
8 `-- spi_oled_app.c
```

株式会社日昇テクノロジー

23.5.1 ハードウェア説明：

OLED は SPI インターフェース 0.96 インチのモノクロディスプレイを使用しており、実物は以下の通りです：



SPI_OLED ディスプレイとボードの接続関係は以下のリストの通りです

SPI_OLED ディスプレイ	ボードピン番号
MOSI	19(MOSI)
なし	21(MISO)
CLK	23(SCLK)
D/C	自分で設定
CS	24(CS0)
GND	25(GND)
VCC	17(VCC)

23.5.1.1 コンパイルと実行

```
1 # ファイルディレクトリ下で

2 make

3 # spi_oled ファイルディレクトリ下に test ファイルが生成され、生成された実行ファイルにソフトリン

クが作成されます

4 # ソフトリンクファイルを直接実行するだけです

5 sudo ./test /dev/spidev3.0 42
```

株式会社日昇テクノロジー

6

7 #/dev/spidev3.0 は CS0 でスクリーンをドライブします

8 #42 は D/C ピンで、実際の状況に応じて変更する必要があります

23.5.2 コード解析

SPI_OLED ディスプレイ実験は、SPI ループバックテストプログラムを修正して得られました。OLED ディスプレイ関連のコードは、IIC で OLED ディスプレイをドライブするプログラムを参照しています。

23.5.2.1 SPI 初期化関数

ここでの SPI 初期化関数は前のセクションと 2 点異なります。第一に、SPI_OLED の D/C ピンを制御するための GPIO デバイスを追加しました。第二に、SPI の動作モードをモード 2 に設定し、読み取りを設定する必要がない（SPI_OLED は書き込みのみのデバイスです）。

リスト 8: spi_oled/sources/spi_oled_app.c

```
1 void spi_and_gpio_init(char *name)

2 {

3 int ret = 0;

4 gpio_init(name);

5 /*

6 * spi mode SPI 動作モードの設定

7 */

8 ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);

9 if (ret == -1)

10 pabort("can't set spi mode");

11
```

株式会社日昇テクノロジー

12 /*

13 * bits per word 1 バイトあたりのビット数の設定

14 */

```
15 ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
16 if (ret == -1)
17 pabort("can't set bits per word");
18
19 /*
20 * max speed hz SPI の最高動作周波数の設定
21 */
22 ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
23 if (ret == -1)
24 pabort("can't set max speed hz");
25
26
27 printf("spi mode: 0x%x¥n", mode);
28 printf("bits per word: %d¥n", bits);
29 printf("max speed: %d Hz (%d KHz)¥n", speed, speed / 1000);
30
```

23.5.2.2 SPI_OLED 命令送信とデータ送信関数

以前の説明によると、SPI_OLED の D/C ピンはディスプレイが受け取るのが命令かデータかを示します。

D/C が低電位の場合は命令を送信し、D/C が高電位の場合はデータを送信します。命令送信関数とデー

株式会社日昇テクノロジー

タ送信関数は、送信を実行する前に D/C 対応のピンを高/低電位に設定する点のみが異なります。以下のように示されます。

リスト 9: spi_oled/sources/spi_oled_app.c

```
1 /*
2 * SPI_OLED に制御命令を送信
3 * cmd、送信する命令。
4 */
5 void spi_oled_send_commend(unsigned char cmd)
6 {
7 uint8_t tx = cmd;
8 uint8_t rx;
9
10 gpio_low(); //SPI_OLED の D/C を低電位に設定
11 transfer(fd, &tx, &rx, 1); //制御命令を送信
12 }
13
14 /*
15 * SPI_OLED にデータを送信
16 * cmd、送信するデータ。
17 */
18 void spi_oled_send_data(unsigned char dat)
19 {
```

株式会社日昇テクノロジー

```
20 uint8_t tx = dat;

21 uint8_t rx;

22

23 gpio_high(); //SPI_OLED の D/C を高電位に設定

24 transfer(fd, &tx, &rx, 1); //データを送信

25 }
```

23.5.2.3 SPI_OLED 初期化コード

SPI_OLED の初期化は 2 つの部分に分けられます。最初に SPI を初期化し、その後 SPI を通じて OLED に設定パラメータを送信します。初期化コードは以下の通りです：

リスト 10: spi_oled/sources/spi_oled_app.c

```
1 /*

2 * oled 初期化関数

3 */

4 void oled_init(char *name)

5 {

6

7 spi_and_gpio_init(name);

8

9 spi_oled_send_commend(0xae);

10 spi_oled_send_commend(0xae); //--turn off oled panel

11 spi_oled_send_commend(0x00); //---set low column address

12 spi_oled_send_commend(0x10); //---set high column address
```

株式会社日昇テクノロジー

13 spi_oled_send_commend(0x40); //--set start line address Set Mapping RAM Display Start Line (0x00~0x3F)

14 spi_oled_send_commend(0x81); //--set contrast control register

```
15 spi_oled_send_commend(0xcf); // Set SEG Output Current Brightness
16 spi_oled_send_commend(0xa1); //--Set SEG/Column Mapping 0xa0,0xa1
17 spi_oled_send_commend(0xc8); //Set COM/Row Scan Direction 0xc0,0xc8
18 spi_oled_send_commend(0xa6); //--set normal display
19 spi_oled_send_commend(0xa8); //--set multiplex ratio(1 to 64)
20 spi_oled_send_commend(0x3f); //--1/64 duty
21 spi_oled_send_commend(0xd3); //-set display offset Shift Mapping RAM Counter (0x00~0x3F)
22 spi_oled_send_commend(0x00); //-not offset
23 spi_oled_send_commend(0xd5); //--set display clock divide ratio/oscillator frequency
24 spi_oled_send_commend(0x80); //--set divide ratio, Set Clock as 100 Frames/Sec
25 spi_oled_send_commend(0xd9); //--set pre-charge period
26 spi_oled_send_commend(0xf1); //Set Pre-Charge as 15 Clocks & Discharge as 1 Clock
27 spi_oled_send_commend(0xda); //--set com pins hardware configuration
28 spi_oled_send_commend(0x12);
29 spi_oled_send_commend(0xdb); //--set vcomh
```

コードを組み合わせて簡単に説明すると、以下の通りです：

- 7 行目、spi_and_gpio_init 初期化関数を呼び出し、以前の説明によると、この関数は SPI と GPIO を初期化します。初期化が完了すると、SPI を介して OLED と通信すると同時に GPIO を介して送信するのが命令かデータかを制御できます。
- 9-37 行目、OLED 初期化パラメータを送信します。

株式会社日昇テクノロジー

• 39 行目、画面をクリアする関数（画面に表示されないようにする）と OLED カーソル設定関数（カーソルを開始位置に設定する）を順に呼び出します。

• プログラム中の OLED 画面クリア関数、文字表示関数、漢字表示関数、画像表示関数は、IIC で IIC_OLED をドライブするのと非常に似ており、関数の単純な置換だけであり、ここでは詳しく説明しません。

23.5.2.4 main 関数の実装

main 関数では、OLED の基本的な表示機能をテストするために OLED 表示関数を呼び出します。完全なコードは以下の通りです：

リスト 11: spi_oled/sources/main.c

```
1 int main(int argc, char *argv[])
2 {
3 int i = 0; //ループ用
4 if(argc < 3){
5 printf("使用方法が間違っています!!!!¥n");
6 printf("使用法: %s [dev] [D/C PinNum]¥n",argv[0]);
7 return -1;
8 }
9 printf("%s¥n",argv[1]);
10 /* SPI デバイスを開く*/
11 fd = open(argv[1], O_RDWR); // ファイルを開いて読み書きを有効にする
12 if (fd < 0){
13 printf("%s を開けません¥n",argv[1]); // i2c dev ファイルを開けない
```

株式会社日昇テクノロジー

```
14 exit(1);

15 }

16

17 oled_init(argv[2]);

18 printf("oled_init¥n");

19 OLED_Fill(0xFF);

20 while (1){

21 OLED_Fill(0xff); //画面をクリア

22 sleep(1);

23 OLED_Fill(0x00); //画面をクリア

24 OLED_ShowStr(0, 3, (unsigned char *)"Wildfire Tech", 1); // 6*8 文字のテスト

25 OLED_ShowStr(0, 4, (unsigned char *)"Hello wildfire", 2); // 8*16 文字のテスト

26 sleep(1);

27 OLED_Fill(0x00); //画面をクリア

28

29 for (i = 0; i < 4; i++)

30 OLED_ShowCN(22 + i * 16, 0, i); //中国語表示のテスト

31

32 sleep(1);

33 OLED_Fill(0x00); //画面をクリア

34

35 OLED_DrawBMP(0, 0, 128, 8, (unsigned char *)BMP1); //BMP ビットマップ表示のテスト

36 sleep(1);
```

株式会社日昇テクノロジー

```
37 OLED_Fill(0x00); //画面をクリア
38 }
39
40 close(fd);
41 return 0;
42 }
43
```

コードの各部分を簡単に説明します：

第 17-19 行、oled を初期化して全画面表示します。使用している oled の解像度は 128*64（64 行、128 列）です。各ピクセル点はオンまたはオフの 2 つの状態（0 または 1）のみです。

第 21 行、画面をクリアします。画面クリアと全画面塗りつぶしは関数のパラメータが異なるだけで、画面クリアは各ピクセル点を 0 に設定し、画面が表示されないようにし、全画面塗りつぶしはすべてのピクセル点を 1 に設定し、画面を全て明るくします。

第 23-26 行、文字列を表示します。文字列表示関数については、IIC で OLED をドライブする章で詳しく説明されていますが、ここではもう一度簡単に説明します。関数の最初の 2 つのパラメータはそれぞれ文字列の開始位置の x 座標と y 座標を設定するために使用され、選択したフォントによって（第 4 パラメータ）、x、y の取得範囲も異なります。6*8 フォントを例にすると、x は 0 から（128-1-6）まで取ることができます。1 を引くのはゼロベースでカウントするため、6 を引くのは 1 文字の幅が 6 ピクセルであり、行の残りのピクセルが 6 未満の場合は完全に表示されないためです。y は 0 から 7 まで取ることができます。OLED ディスプレイには 64 行があり、8 行ごとに 1 グループに分けられるため、合計 8 グループがあり、文字の開始位置の y 座標は 0 から 7 まで取ることができます。

第 27-32 行、中国語を表示します。中国語を使用するには、対応するドットマトリクスのフォントラ

株式会社日昇テクノロジー

イブラリが必要です。フォントライブラリの作成ツールおよび作成方法については、SPI_OLED モジュールの資料を参照してください。ここでは詳しく説明しません。

第 33-37 行、画像を表示します。中国語を表示するのと同様に、画像表示関数を使用する前に、画像をドットマトリクスデータに変換する必要があります。制作ツールおよび制作方法については、SPI_OLED モジュールの資料を参照してください。

第 24 章 PWM（パルス幅変調）

この章では Linux PWM に関連するアプリケーションレベルのプログラム制御について説明します。

この章のサンプルコードディレクトリは：base_linux/pwm

24.1 パルス幅変調

24.1.1 PWM とは

PWM（Pulse Width Modulation）は、パルス幅変調のことで、マイクロプロセッサのデジタル出力を利用してアナログ回路を制御する非常に効果的な技術です。測定、通信、工業制御などの分野で広く応用されています。

24.1.2 PWM の周波数

1 秒間に信号が高電位から低電位へ、そして再び高電位へ戻る回数を指し、つまり 1 秒間に PWM が何周期あるかを示します。単位は Hz です。

24.1.3 PWM の周期

T=1/f、T は周期、f は周波数です。

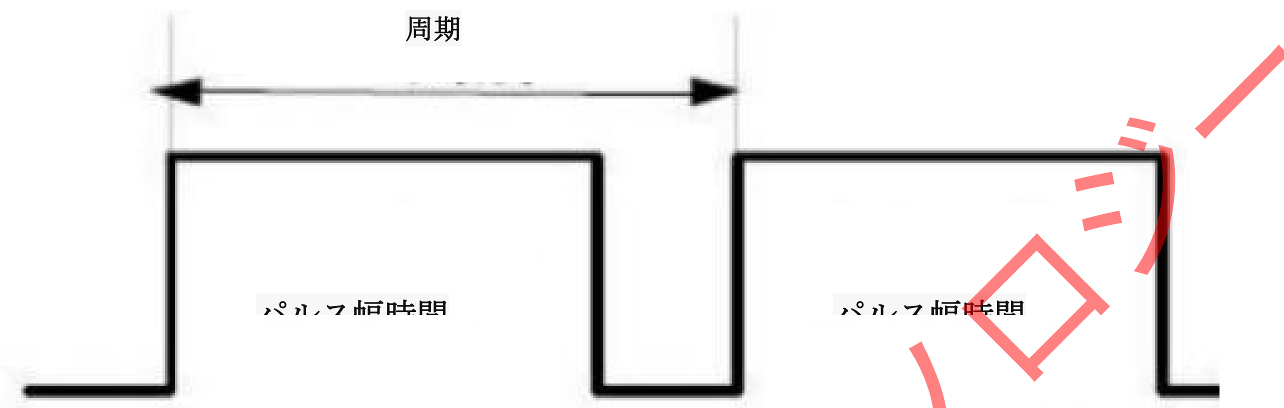
例えば、周波数が 50Hz の場合、1 周期は 20ms で、1 秒間に 50 回の PWM 周期があります。

株式会社日昇テクノロジー

24.1.4 占有率

パルス周期内の高電位の時間と全周期時間の比率を指し、単位は%（0%-100%）です。

周期の長さは下図のように示されます。



ここで、周期は 1 秒間に T 回の周波数 f で発生するパルス信号の時間で、パルス幅は高電位の時間を指します。

図示されているように、パルス幅の時間が全周期時間の比率、つまり占有率です。

例えば、周期の時間が 10ms で、パルス幅の時間が 8ms の場合、占有率は 8/10=80%となり、これが80%の占有率を持つパルス信号です。

PWM はパルス幅を変調することで、パルス幅を調整することができます。

24.1.5 PWM の原理

高電位が 5V、低電位が 0V とすると、異なるアナログ電圧を出力するために PWM が必要になります。IO ポートが出力する方形波の占有率を変更することで、デジタル信号をアナログ電圧信号として模倣することができます。

電圧はパルスシーケンスとしてアナログ負荷に加えられ、接続時は高電位 1、切断時は低電位 0 です。接続時は直流電源が出力され、切断時は直流電源が切断されます。接続と切断の時間を制御することで、理論上、最大電圧値 5V 以下の任意のアナログ電圧を出力することができます。

例えば、占有率が 50%の場合、高電位時間と低電位時間が半分ずつになり、一定の周波数下で 2.5V の模

株式会社日昇テクノロジー

倣出力電圧を得ることができます。75%の占有率では、得られる電圧は 3.75V になります。

24.2 PWM ピン

LubanCat ボードには PWM 機能を持つ GPIO が 4 つ統合されています。

機能	PIN		機能
3.3V	1	2	5V
I2C3_SDA	3	4	5V
I2C3_SCL	5	6	GND
GPIO	7	8	UART_TX
GND	9	10	UART_RX
GPIO	11	12	PWM
GPIO	13	14	GND
GPIO	15	16	GPIO
3.3V	17	18	GPIO
MOSI	19	20	GND
MISO	21	22	GPIO
SCLK	23	24	CSO
GND	25	26	CSI
I2C5_SDA	27	28	I2C5_SCL
GPIO	29	30	GND
GPIO	31	32	PWM
PWM	33	34	GND
PWM	35	36	GPIO
GPIO	37	38	GPIO
GND	39	40	GPIO

株式会社日昇テクノロジー

24.2.1 PWM インターフェース機能の有効化

デフォルトでは PWM インターフェースはオフになっており、使用する前に有効化する必要があります。

各ボードには 4 つのハードウェア PWM があり、以下は PWM8 と PWM9 を例にした有効化操作です。

24.2.1.2

/boot/uEnv/board.txt を開いて、pwm 関連のデバイスツリープラグインが有効になっているか確認できます。

ファイルを編集し、pwm8 と pwm9 の行のコメントアウトを解除します。

```
root@lubancat:~# cat /boot/uEnv/uEnvLubanCat1.txt
uname_r=4.19.232
size=0x1000000
#dtb=rk3566-lubancat1.dtb

enable_uboot_overlays=1
#overlay_start

dtoverlay=/dtb/overlay/lubancat-i2c3-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-i2c5-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm10-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm14-m0-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-pwm8-m0-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-pwm9-m0-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-spi3-m1-gpio-cs-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-spi3-m1-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-uart3-m1-overlay.dtbo

#overlay_end

root@lubancat:~#
```

その後、デバイスを再起動して有効化します。

注: 電源を抜いて再起動する方法では、ファイルが変更されない可能性があります（理由：ファイルがメモリからストレージデバイスに同期されなかった場合）。解決方法として、ターミナル上で「sync」コマンドを入力してから電源を切ってください。

24.3 PWM デバイスの確認

PWM 通信インターフェースを有効にした後、以下のコマンドで PWM が開始されたか確認できます。

```
ls /sys/class/pwm/
```

株式会社日昇テクノロジー

```
root@lubancat:/home/cat# ls /sys/class/pwm/
pwmchip0  pwmchip1  pwmchip2
root@lubancat:/home/cat#
root@lubancat:/home/cat#
root@lubancat:/home/cat#
root@lubancat:/home/cat#                PWM9
root@lubancat:/home/cat#
root@lubancat:/home/cat# PWM8
root@lubancat:/home/cat#
root@lubancat:/home/cat#
root@lubancat:/home/cat#
```

pwmchip0 は画面のバックライト用で、システムデフォルトで開始されます。複数の PWM デバイスツリープラグインを開始すると、PWM コントローラの値が小さいほど、システムに割り当てられる pwmchip が小さくなります。

```
1 たとえば、pwm8、pwm9、pwm14 を同時に開始した場合、以下の対応関係が生じます
2
3 pwm8 -> pwmchip1
4 pwm9 -> pwmchip2
5 pwm14 -> pwmchip3
```

24.4 PWM 制御方法(shell)

以下の操作は pwm8 を例にしています。

注: 操作前に必ずデバイスツリープラグインを開始し、pwm8 ピンを有効にして再起動してください。

```
1 # pwm3 をユーザースペースにエクスポートします
2 echo 0 > /sys/class/pwm/pwmchip1/export
3
4 # pwm の周期を ns 単位で設定します
5 echo 1000000 > /sys/class/pwm/pwmchip1/pwm0/period
6
7 # デューティサイクルを設定します
8 echo 500000 > /sys/class/pwm/pwmchip1/pwm0/duty_cycle
```

株式会社日昇テクノロジー

```
9

10 # pwm の極性を設定します

11 echo "normal" > /sys/class/pwm/pwmchip1/pwm0/polarity

12

13 # pwm を有効にします

14 echo 1 > /sys/class/pwm/pwmchip1/pwm0/enable

15

16 # pwm3 のユーザースペースへのエクスポートを解除します

17 echo 0 > /sys/class/pwm/pwmchip1/unexport
```

提示: period と duty_cycle の値を設定する際には、どの状況でも period の値が duty_cycle の値以上であることを確認してください。

24.5 PWM 制御方法（システムコール）

リスト 1: base_linux/pwm/pwm_test.c

```
1 #include <stdio.h>

2 #include <stdlib.h>

3 #include <sys/types.h>

4 #include <sys/stat.h>

5 #include <fcntl.h>

6 #include <unistd.h>

7 #include <string.h>

8

9

10 static char pwm_path[75];
```

株式会社日昇テクノロジー

```
11 static int pwm_config(const char *attr, const char *val) // PWM の設定

12 {

13 char file_path[100];

14 int len;

15 int fd;

16 sprintf(file_path, "%s/%s", pwm_path, attr);

17 if ((fd = open(file_path, O_WRONLY)) < 0) {

18 perror("open error");

19 return fd;

20 }

21 len = strlen(val);

22 if (len != write(fd, val, len)) {

23 perror("write error");

24 close(fd);

25 return -1;

26 }

27 close(fd); // ファイルを閉じる

28 return 0;

29 }

30

31

32 int main(int argc, char *argv[])

33 {
```

株式会社日昇テクノロジー

```
34 /* 引数の確認 */
35 if (4 != argc) {
36 fprintf(stderr, "使用法: %s <id> <period> <duty>¥n",argv[0]);
37 exit(-1);
38 }
39 /* 設定情報の表示 */
40 printf("PWM 設定: id<%s>, period<%s>, duty<%s>¥n",argv[1], argv[2], argv[3]);
41
42 /* pwm をエクスポート */
43 sprintf(pwm_path, "/sys/class/pwm/pwmchip%s/pwm0", argv[1]);
44 // pwm0 ディレクトリが存在しない場合、エクスポートする
45 if (access(pwm_path, F_OK)) {
46 char temp[100];
47 int fd;
48 sprintf(temp, "/sys/class/pwm/pwmchip%s/export", argv[1]);
49 if (0 > (fd = open(temp, O_WRONLY))) {
50 perror("open error");
51 exit(-1);
52 }
53 // pwm をエクスポート
54 if (1 != write(fd, "0", 1)) {
55 perror("write error");
56 close(fd);
```

株式会社日昇テクノロジー

```
57 exit(-1);
58 }
59 close(fd); // ファイルを閉じる
60 }
61
62 /* PWM 周期の設定 */
63 if (pwm_config("period", argv[2]))
64 exit(-1);
65 /* デューティサイクルの設定 */
66 if (pwm_config("duty_cycle", argv[3]))
67 exit(-1);
68 /* pwm を有効にする */
69 pwm_config("enable", "1");
70 getchar();
71 /* プログラムを終了 */
72 exit(0);
73 }
```

方法 1：

```
1. make
```

株式会社日昇テクノロジー

方法 2：

```
gcc pwm_test.c -o pwm_test
```

```
1 # PWM の周期とデューティサイクルを設定

2

3 # ./pwm_test pwmchip2 周期 デューティサイクル

4 sudo ./pwm_test 2 1000000 500000
```

第 25 章 カメラ

LubanCat-RK のボードで使用されているカメラインターフェースは、24 ピンの mipi インターフェースで、現在は imx415 カメラが対応しています。インターフェースの近くには通常「mipi csi」という言葉があります。インターフェースの図は以下の通りです。

注意: フラットケーブルの金属ピンはボード側を向いている必要があります。

接続図は以下の通りです。

株式会社日昇テクノロジー

25.1 カメラ設定

LubanCat-4 ボードは現在、imx415 カメラのみをサポートしており、最大で 3 台の imx415 カメラを同時に動作させることができます。

ここでは Debian 11 システムを使用して説明します。

25.1.1. シングルカメラ

CAM0、CAM1、CAM2 はそれぞれ独立してカメラを接続できます。カメラはデフォルトでオフになっており、/boot/uEnv/uEnv.txt ファイルで設定します。

株式会社日昇テクノロジー

25.1.1.1. 有効化方法

ここでは CAM0 を有効にする例を示します。CAM1、CAM2 の有効化方法も同様です。

1. 設定ファイルを開きます

```
vi /boot/uEnv/uEnv.txt
```

2. CAM0 のデバイスツリープラグインを見つけ、プラグインの前の#を削除してファイルを保存し、再起動します。変更内容は以下の通りです。

```
# CAM0
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam0-imx415-overlay.dtbo
# CAM1
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam1-imx415-overlay.dtbo
# CAM2
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam2-imx415-overlay.dtbo
```

25.1.1.2. 無効化方法

ここでは CAM0 を無効にする例を示します。CAM1、CAM2 の無効化方法も同様です。

1. 設定ファイルを開きます

```
vi /boot/uEnv/uEnv.txt
```

2. CAM0 のデバイスツリープラグインを見つけ、プラグインの前に#を追加してファイルを保存し、再起動します。変更内容は以下の通りです。

```
# CAM0
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam0-imx415-overlay.dtbo
# CAM1
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam1-imx415-overlay.dtbo
# CAM2
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam2-imx415-overlay.dtbo
```

25.1.1.3. カメラノード

シングルカメラのノードは/dev/video11 です。

株式会社日昇テクノロジー

カメラのプレビュー

```
gst-launch-1.0 v4l2src device=/dev/video11 ! video/x-raw,format=NV12,width=3840,height=2160,framerate=30/1 ! videoconvert ! autovideosink
```

25.1.2. デュアルカメラ

CAM0、CAM1、CAM2 はそれぞれ独立してカメラを接続できます。カメラはデフォルトでオフになっており、/boot/uEnv/uEnv.txt ファイルで設定します。

デュアルカメラは任意の組み合わせで使用できます。

- CAM0 と CAM1
- CAM0 と CAM2
- CAM1 と CAM2

25.1.2.1. 有効化方法

ここでは CAM0 と CAM1 を有効にする例を示します。他の方法も同様です。

1. 設定ファイルを開きます

vi /boot/uEnv/uEnv.txt

2. CAM0 と CAM1 のデバイスツリープラグインを見つけ、プラグインの前の#を削除してファイルを保存し、再起動します。変更内容は以下の通りです。

```
# CAM0
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam0-imx415-overlay.dtbo
# CAM1
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam1-imx415-overlay.dtbo
# CAM2
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam2-imx415-overlay.dtbo
```

25.1.2.2. 無効化方法

ここでは CAM0 と CAM1 を無効にする例を示します。他の方法も同様です。

株式会社日昇テクノロジー

1. 設定ファイルを開きます

```
vi /boot/uEnv/uEnv.txt
```

2. CAM0 と CAM1 のデバイスツリープラグインを見つけ、プラグインの前に#を追加してファイルを保存し、再起動します。変更内容は以下の通りです。

```
# CAM0
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam0-imx415-overlay.dtbo
# CAM1
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam1-imx415-overlay.dtbo
# CAM2
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam2-imx415-overlay.dtbo
```

25.1.2.3. カメラノード

デュアルカメラのノードは/dev/video22 と/dev/video31 です。

CAM0 と CAM1 を有効にした場合、それぞれのノードは/dev/video22 と/dev/video31 です。

デュアルカメラのプレビュー

```
gst-launch-1.0 v4l2src device=/dev/video22 ! video/x-raw,format=NV12,width=3840,height=2160,framerate=30/1 ! videoconvert ! autovideosink &
gst-launch-1.0 v4l2src device=/dev/video31 ! video/x-raw,format=NV12,width=3840,height=2160,framerate=30/1 ! videoconvert ! autovideosink &
```

25.1.3. トリプルカメラ

CAM0、CAM1、CAM2 はそれぞれ独立してカメラを接続できます。カメラはデフォルトでオフになっており、/boot/uEnv/uEnv.txt ファイルで設定します。

25.1.3.1. 有効化方法

ここでは CAM0、CAM1、CAM2 を有効にする例を示します。他の方法も同様です。

株式会社日昇テクノロジー

1. 設定ファイルを開きます

vi /boot/uEnv/uEnv.txt

2. CAM0、CAM1、CAM2 のデバイスツリープラグインを見つけ、プラグインの前の#を削除してファイルを保存し、再起動します。変更内容は以下の通りです。

```
# CAM0
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam0-imx415-overlay.dtbo
# CAM1
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam1-imx415-overlay.dtbo
# CAM2
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam2-imx415-overlay.dtbo
```

25.1.3.2. 無効化方法

ここでは CAM0、CAM1、CAM2 を無効にする例を示します。他の方法も同様です。

1. 設定ファイルを開きます

vi /boot/uEnv/uEnv.txt

2. CAM0、CAM1、CAM2 のデバイスツリープラグインを見つけ、プラグインの前に#を追加してファイルを保存し、再起動します。変更内容は以下の通りです。

```
# CAM0
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam0-imx415-overlay.dtbo
# CAM1
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam1-imx415-overlay.dtbo
# CAM2
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam2-imx415-overlay.dtbo
```

25.1.3.3. カメラノード

トリプルカメラのノードは/dev/video33、/dev/video42、/dev/video51 です。

CAM0、CAM1、CAM2 を有効にした場合、それぞれのノードは/dev/video33、/dev/video42、/dev/video51 です

トリプルカメラのプレビュー

株式会社日昇テクノロジー

```
gst-launch-1.0 v4l2src device=/dev/video33 ! video/x-raw,format=NV12,width=3840,height=2160,framerate=30/1 ! videoconvert ! autovideosink &
gst-launch-1.0 v4l2src device=/dev/video42 ! video/x-raw,format=NV12,width=3840,height=2160,framerate=30/1 ! videoconvert ! autovideosink &
gst-launch-1.0 v4l2src device=/dev/video51 ! video/x-raw,format=NV12,width=3840,height=2160,framerate=30/1 ! videoconvert ! autovideosink &
```

25.2 カメラ情報を表示する

25.2.1 カメラデバイスを一覧表示する

```
1 v4l2-ctl --list-devices
```

単一のカメラは次の図のようになります

```
root@lubancat:/home/cat# v4l2-ctl --list-devices
rkisp-statistics (platform: rkisp):
        /dev/video18
        /dev/video19

rkcif-mipi-lvds (platform:rkcif):
        /dev/media0

rkisp_mainpath (platform:rkisp0-vir0):
        /dev/video11
        /dev/video12
        /dev/video13
        /dev/video14
        /dev/video15
        /dev/video16
        /dev/video17
        /dev/media1
```

Rkisp_mainpath の一番目の/dev/video11 はいま使っているカメラです。

株式会社日昇テクノロジー

25.2.2 カメラのフォーマットと解像度を確認する

1. サポートされているフォーマットを確認する

```
v4l2-ctl --list-formats-ext --device=/dev/video11
```

コマンドの出力は以下の通りです：

```
root@lubancat:/home/cat# v4l2-ctl --list-formats-ext --device=/dev/video11
ioctl: VIDIOC_ENUM_FMT
        Type: Video Capture Multiplanar

        [0]: 'RG10' (10-bit Bayer RGRG/GBGB)
                Size: Stepwise 64x64 - 3864x2192 with step 8/8
        [1]: 'BA10' (10-bit Bayer GRGR/BGBG)
                Size: Stepwise 64x64 - 3864x2192 with step 8/8
        [2]: 'GB10' (10-bit Bayer GBGB/RGRG)
                Size: Stepwise 64x64 - 3864x2192 with step 8/8
        [3]: 'BG10' (10-bit Bayer BGBG/GRGR)
                Size: Stepwise 64x64 - 3864x2192 with step 8/8
        [4]: 'Y10 ' (10-bit Greyscale)
                Size: Stepwise 64x64 - 3864x2192 with step 8/8
```

現在のカメラは RG10、Y10 などのフォーマットをサポートし、最大解像度は 3864x2192 です。

重要: カメラが接続されているかどうかに関わらず、システムには/dev/video0 が存在します。カメラが接続されている場合、最大解像度が 800x600 であり、実際のパラメータがこれと異なる場合、現在のカメラは video0 ではないか、カメラのハードウェア接続に問題がある可能性があります。

25.2.3 カメラがサポートする設定パラメータを確認する

1. カメラがサポートする設定パラメータを確認する

v4l2-ctl --all --device /dev/video11

株式会社日昇テクノロジー

```
root@lubancat:/home/cat# v4l2-ctl --all --device /dev/video11
Driver Info:
        Driver name      : rkcif
        Card type        : rkcif
        Bus info         : platform:rkcif-mipi-lvds2
        Driver version   : 5.10.160
        Capabilities     : 0x84201000
                Video Capture Multiplanar
                Streaming
                Extended Pix Format
                Device Capabilities
        Device Caps      : 0x04201000
                Video Capture Multiplanar
                Streaming
                Extended Pix Format
Media Driver Info:
        Driver name      : rkcif
        Model            : rkcif-mipi-lvds2
        Serial           :
        Bus info         :
        Media version    : 5.10.160
        Hardware revision: 0x00000000 (0)
        Driver version   : 5.10.160
Interface Info:
        ID               : 0x03000002
        Type             : V4L Video
Entity Info:
        ID               : 0x00000001 (1)
        Name             : stream_cif_mipi_id0
        Function         : V4L2 I/O
        Pad 0x01000004   : 0: Sink
          Link 0x02000043: from remote pad 0x100002f of entity 'rockchip-mipi-csi2': Data, Enabled
          Link 0x02000059: from remote pad 0x1000030 of entity 'rockchip-mipi-csi2': Data
          Link 0x0200006f: from remote pad 0x1000031 of entity 'rockchip-mipi-csi2': Data
          Link 0x02000085: from remote pad 0x1000032 of entity 'rockchip-mipi-csi2': Data
          Link 0x0200009b: from remote pad 0x1000033 of entity 'rockchip-mipi-csi2': Data
          Link 0x020000b1: from remote pad 0x1000034 of entity 'rockchip-mipi-csi2': Data
          Link 0x020000c7: from remote pad 0x1000035 of entity 'rockchip-mipi-csi2': Data
          Link 0x020000dd: from remote pad 0x1000036 of entity 'rockchip-mipi-csi2': Data
          Link 0x020000f3: from remote pad 0x1000037 of entity 'rockchip-mipi-csi2': Data
```

```
          Link 0x0200009b: from remote pad 0x1000033 of entity 'rockchip-mipi-csi2': Data
          Link 0x020000b1: from remote pad 0x1000034 of entity 'rockchip-mipi-csi2': Data
          Link 0x020000c7: from remote pad 0x1000035 of entity 'rockchip-mipi-csi2': Data
          Link 0x020000dd: from remote pad 0x1000036 of entity 'rockchip-mipi-csi2': Data
          Link 0x020000f3: from remote pad 0x1000037 of entity 'rockchip-mipi-csi2': Data
          Link 0x02000109: from remote pad 0x1000038 of entity 'rockchip-mipi-csi2': Data
          Link 0x0200011f: from remote pad 0x1000039 of entity 'rockchip-mipi-csi2': Data
Priority: 2
Format Video Capture Multiplanar:
        Width/Height      : 3840/2160
        Pixel Format      : 'GB10' (10-bit Bayer GBGB/RGRG)
        Field             : None
        Number of planes  : 1
        Flags             : premultiplied-alpha, 0x000000fe
        Colorspace        : Unknown (0x099ebc38)
        Transfer Function : Default
        YCbCr/HSV Encoding: Unknown (0x000000ff)
        Quantization      : Default
        Plane 0           :
           Bytes per Line : 4864
           Size Image     : 10506240
Selection Video Capture: crop, Left 12, Top 16, Width 3840, Height 2160, Flags:
Selection Video Capture: crop_bounds, Left 12, Top 16, Width 3840, Height 2160, Flags:
Selection Video Output: crop, Left 12, Top 16, Width 3840, Height 2160, Flags:
Selection Video Output: crop_bounds, Left 12, Top 16, Width 3840, Height 2160, Flags:
```

株式会社日昇テクノロジー

25.3 ビデオ

```
1 # ビデオ撮影
2 v4l2-ctl --verbose -d /dev/video11 --set-fmt-video=width=800,height=600,pixelformat='NV12' --stream-mmap=4 --set-selection=target=crop,flags=0,top=0,left=0,width=800,height=600 --stream-to=out.yuv
3
4 # ビデオ再生
5
6 ffplay -f rawvideo -video_size 800x600 -pixel_format nv12 out.yuv
7
8 # out.yuv が撮影されたビデオです
```

25.4 写真

```
1 # 写真を撮る
2 sudo gst-launch-1.0 v4l2src num-buffers=1 device=/dev/video11 ! jpegenc ! filesink location=picture.jpg
3
4 # picture.jpg が撮影された写真です
```

25.5 デスクトップでの撮影

guvcview は非常に強力なカメラプレビューおよび撮影ソフトウェアです。

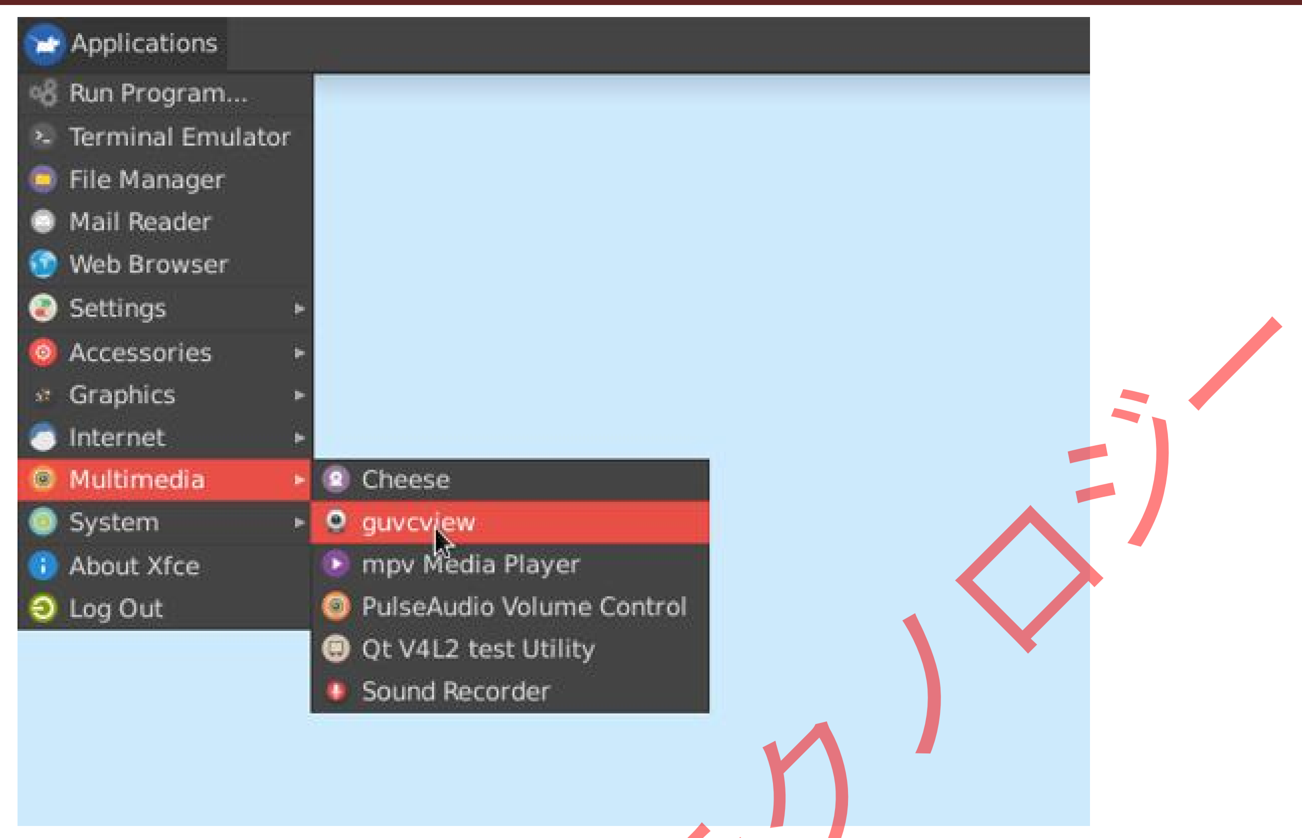
ソフトウェアのインストール

```
sudo apt update
sudo apt install guvcview
```

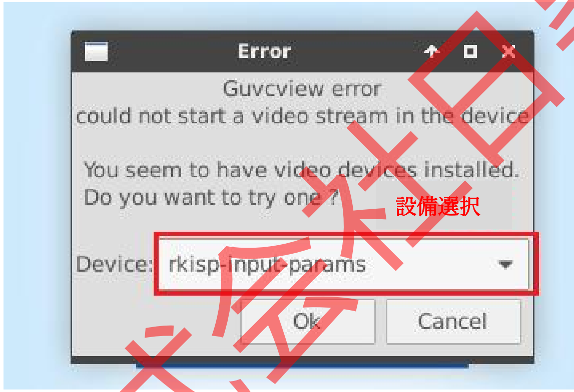
25.5.1 使用方法

デスクトップで guvcview を開きます。以下の図のように表示されます。

株式会社日昇テクノロジー



デバイスが見つからない場合は、デバイスオプションをクリックして自動選択します。以下の図のように表示されます。



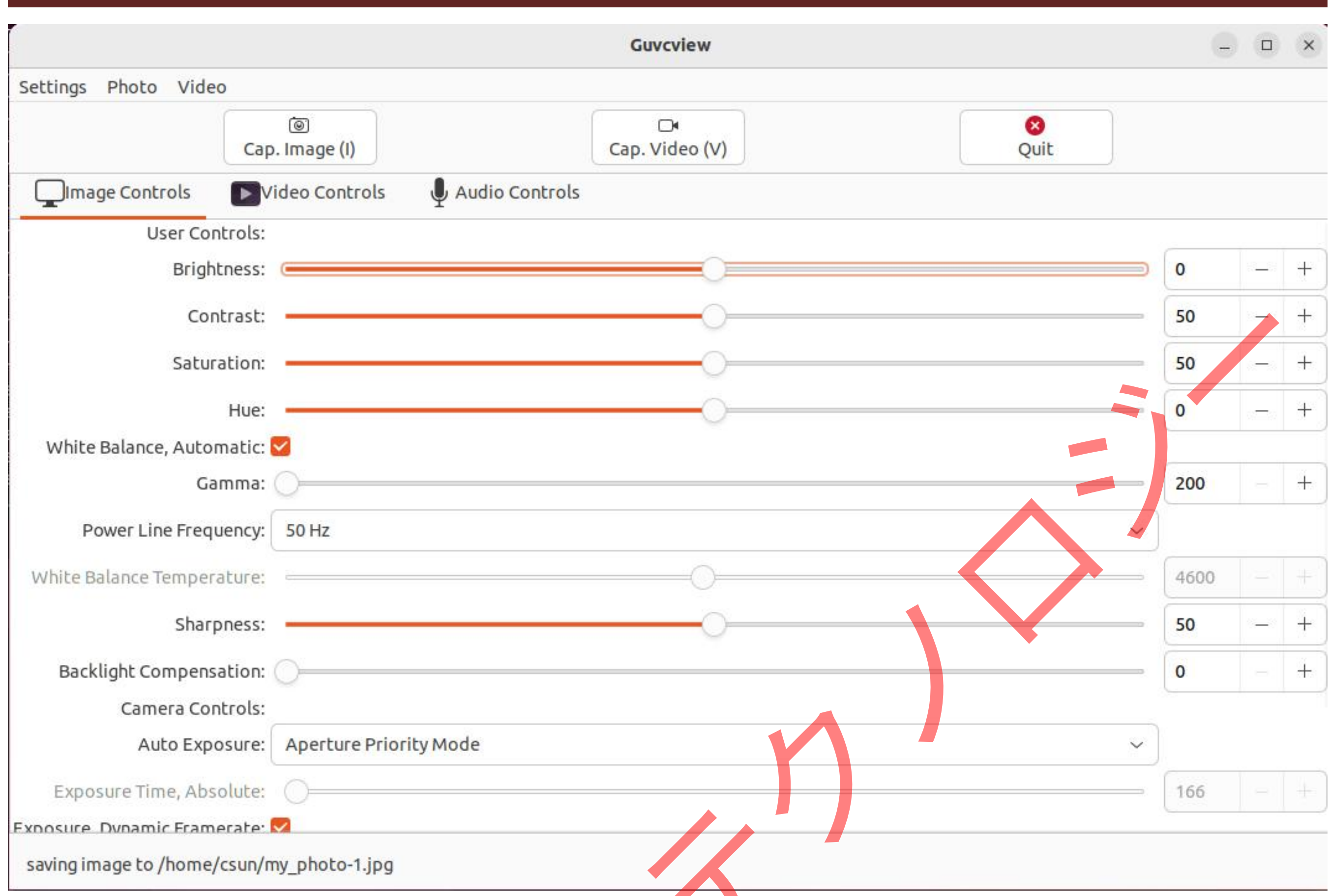
リストのデバイスの中に、実際のカメラデバイスが 2 つあり、それらは rkisp_mainpath という名前です。順番により、どのデバイスがどちらのカメラかを判断します。前にあるものが CAM0、後にあるものが CAM1 です。どちらかのデバイスを選択して開きます。

株式会社日昇テクノロジー

デバイスを開いた後、以下の図のように表示されます。

image_control では、プレビュー画像の表示状態を調整し、間接的にカメラのドライバを制御できます。

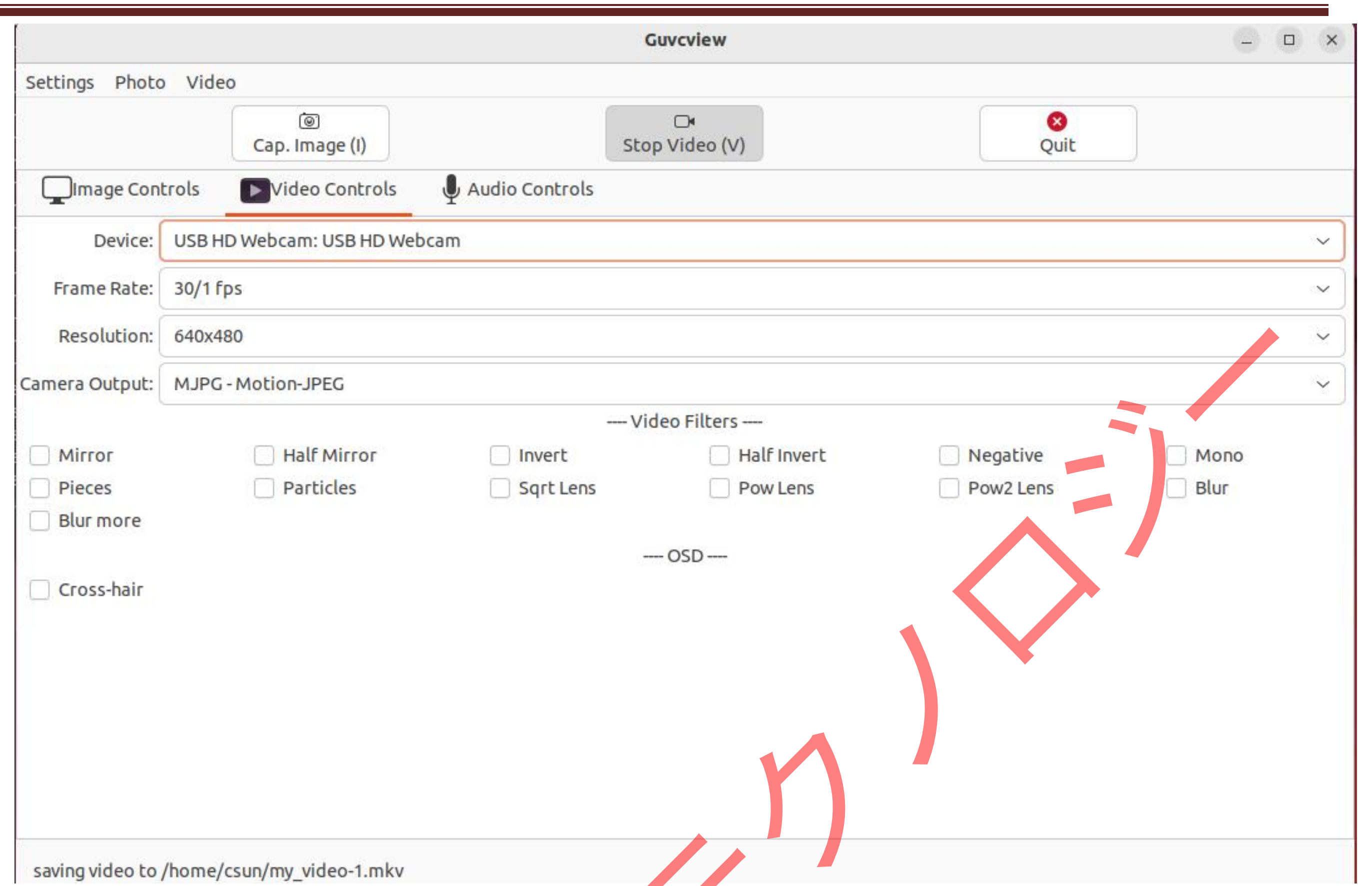
株式会社日昇テクノロジー



初めてデバイスを開いたとき、プレビューウィンドウの解像度は非常に低いため、video_control で変更する必要があります。

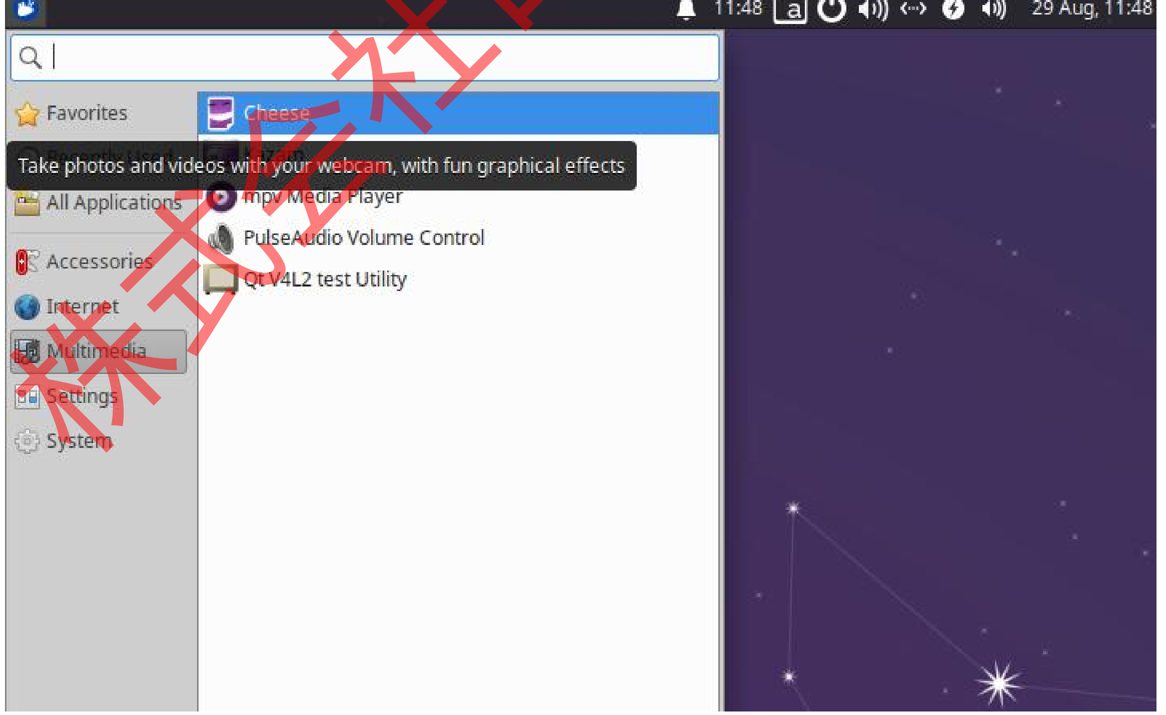
video_control の使用方法は以下の図の通りです。

株式会社日昇テクノロジー



調整が完了したら、1 つ目のカメラの設定が終了します。同じ操作を繰り返してもう 1 つのカメラを開きます。

デスクトップシステムに入り、左上のボタンをクリックして「cheese」を探します。

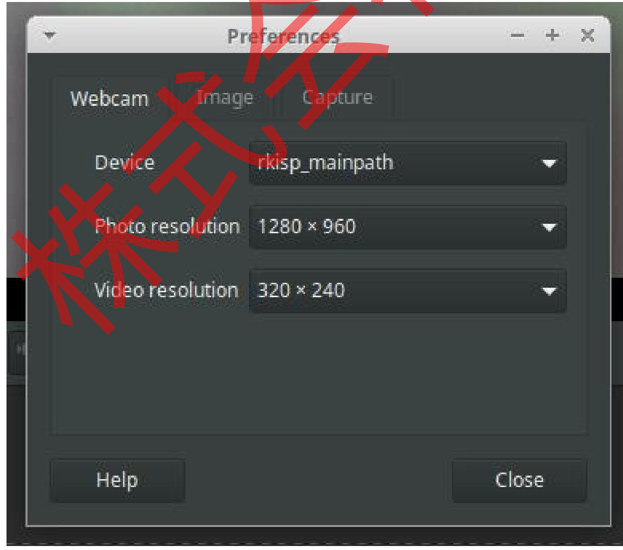


株式会社日昇テクノロジー

「cheese」のインターフェイスは以下の画像のようになっています。

Take a Photo
設定
撮影した画面
連写
撮影
録画
Photo Video Burst
撮影キー
Effects
メディアアライブラリ、撮影した写真や動画はここに保存されます

設定を変更することで撮影の解像度を変更できます。



撮影サンプルを表示します。

株式会社日昇テクノロジー

第 26 章 音声

音声とビデオは日常生活で広く利用される情報取得手段で、その本質は音声と画像情報の収集、保存、再生です。この章では、音声とビデオの再生に関する基本概念を紹介し、ボード上で音声とビデオを再生する方法をデモンストレーションします。

26.1 サウンドカードの設定

26.1.1 デバイス設定とツールのインストール

サウンドカードドライバディレクトリを確認する

```
1 ls /dev/snd/
```

株式会社日昇テクノロジー

```
1 root@lubancat:~# ls /dev/snd/
2 by-path controlC0 pcmC0D0c pcmC0D0p seq timer
3 root@lubancat:~#
```

- controlC0：サウンドカードの制御用、C0 はサウンドカード 0 を意味します。
- pcmC0D0c：録音用の pcm デバイス、最後の「c」は capture の略で、録音を意味します。
- pcmC0D0p：再生用の pcm デバイス、最後の「p」は playback の略で、再生を意味します。
- timer：タイマー

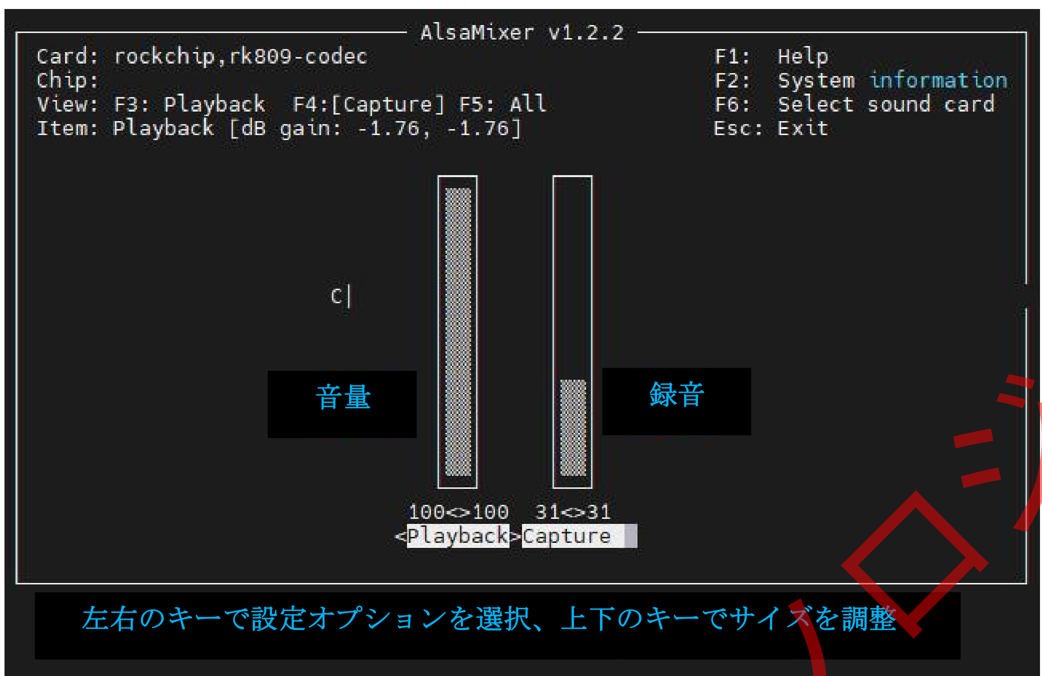
26.1.2 サウンドカードの音量設定

26.1.2.1 alsamixer

これはコマンドラインのグラフィカルな設定ツールです。

```
1 alsamixer
```

- F4 を押すと録音設定モードに入ります。



26.1.2.2 amixer

音量を設定する前に、amixer を使用して再生の id 番号を取得する必要があります。

```
1 # サウンドカードの取得
2 amixer controls
```

```
1 cat@lubancat:~$ amixer controls
2 numid=2,iface=MIXER,name='Capture MIC Path'
3 numid=4,iface=MIXER,name='Capture Volume'
4 numid=1,iface=MIXER,name='Playback Path'
5 numid=3,iface=MIXER,name='Playback Volume'
6 cat@lubancat:~$
```

numid=3,iface=MIXER,name='Playback Volume'は、再生音量を設定するデバイスです。以下のコマンドで音量を制御できます。

株式会社日昇テクノロジー

```
1 # 音量の大きさを取得する

2 amixer cget numid=3

3 # 音量の大きさを設定する

4 amixer cset numid=3 120
```

現在の音量現在の音量

```
cat@lubancat:~$ amixer cget numid=3
numid=3,iface=MIXER,name='Playback Volume'
  ; type=INTEGER,access=rw---R--,values=2,min=0,max=252,step=0
  : values=200,200
  | dBscale-min=-95.00dB,step=0.37dB,mute=0
cat@lubancat:~$
cat@lubancat:~$
cat@lubancat:~$
cat@lubancat:~$ amixer cset numid=3 120
numid=3,iface=MIXER,name='Playback Volume'
  ; type=INTEGER,access=rw---R--,values=2,min=0,max=252,step=0
  : values=120,120
  | dBscale-min=-95.00dB,step=0.37dB,mute=0
cat@lubancat:~$
```

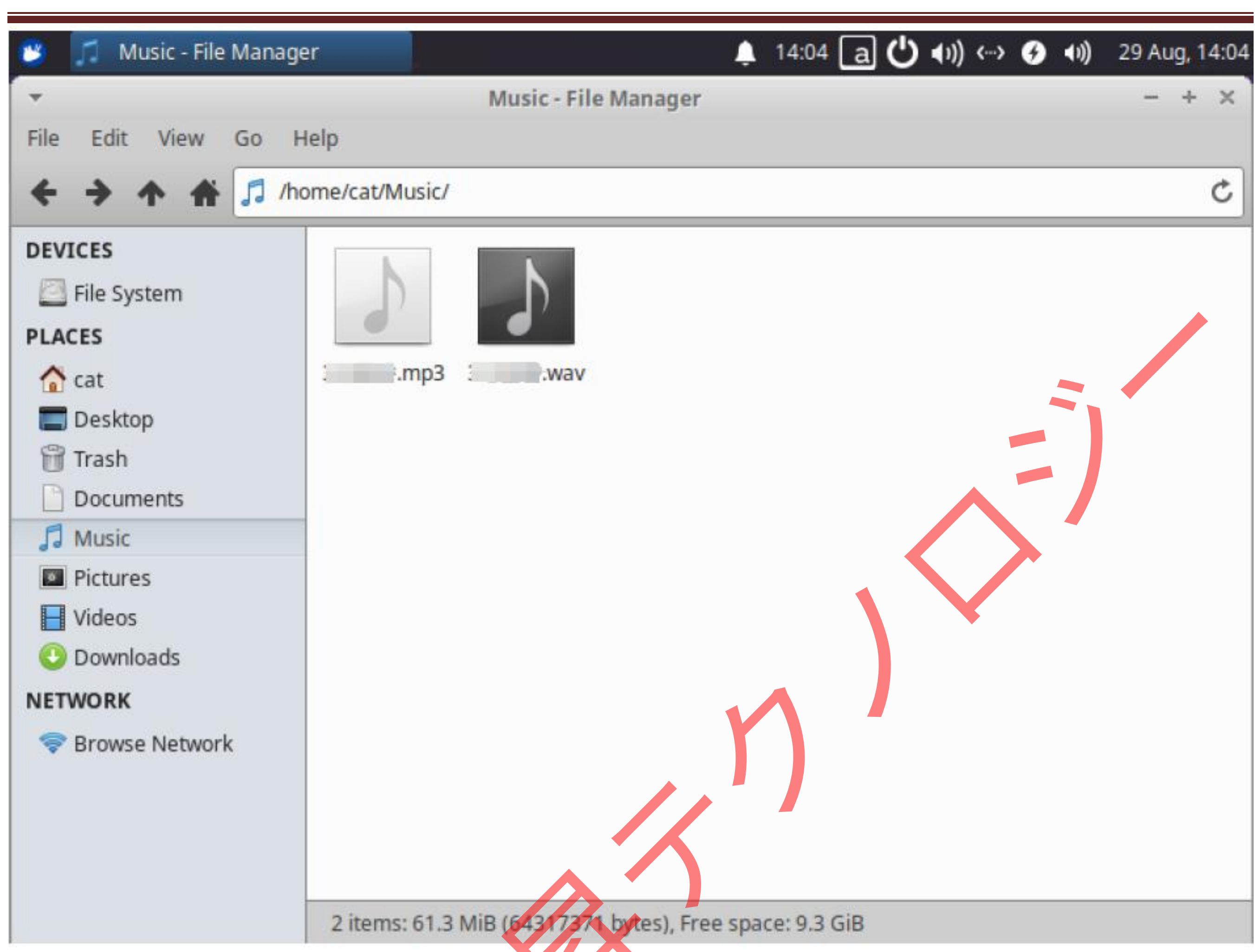
現在の音量
当前音量
設定後の音量
设置后的音量

26.2 デスクトップでの音声再生

26.2.1 音楽の再生

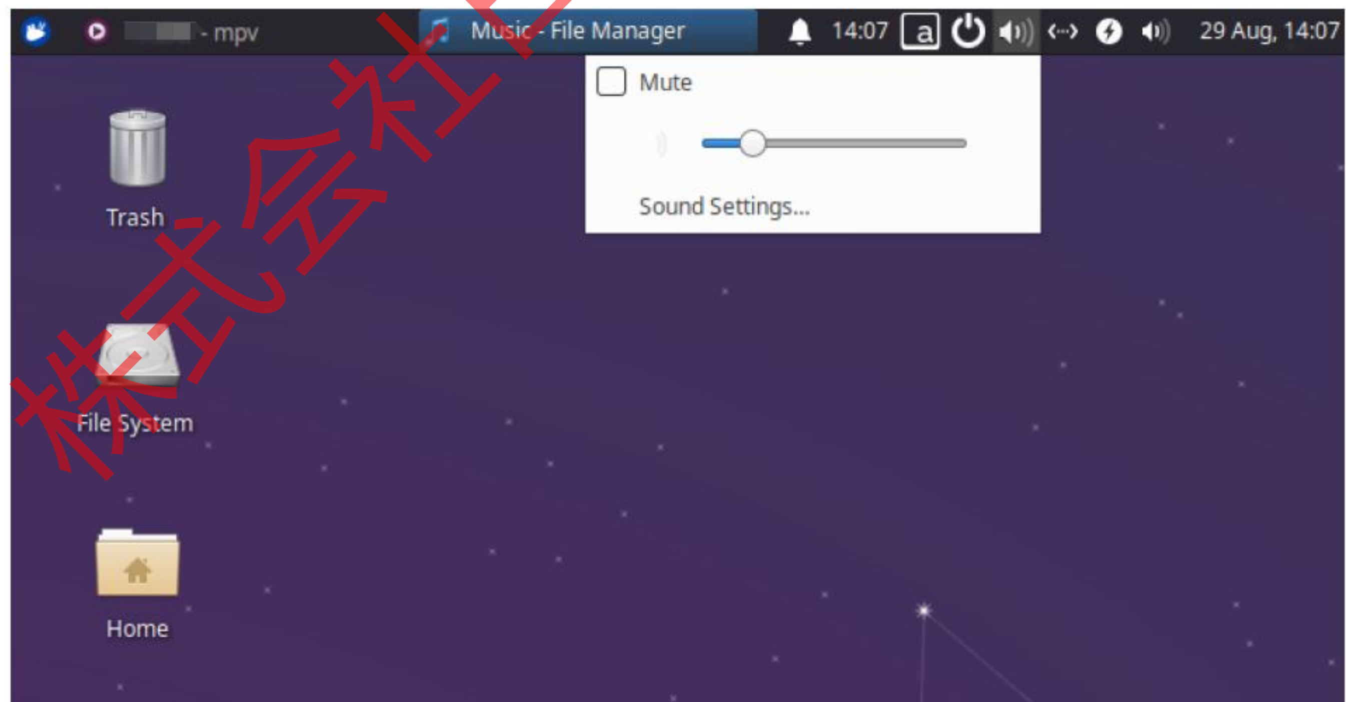
音楽ファイルがあるフォルダに入り、ダブルクリックすると音楽が再生されます。デフォルトの音楽プレーヤーは mpv Media Player ですが、他の音楽プレーヤーを使用したい場合は、そのプレーヤーのインストール方法に従ってインストールしてください。

株式会社日昇テクノロジー



株式会社日昇テクノロジー

右上隅の音量ボタンをクリックして調整することができます。さらに多くのオプションを調整するためには、さらに多くのオプションを選択します。

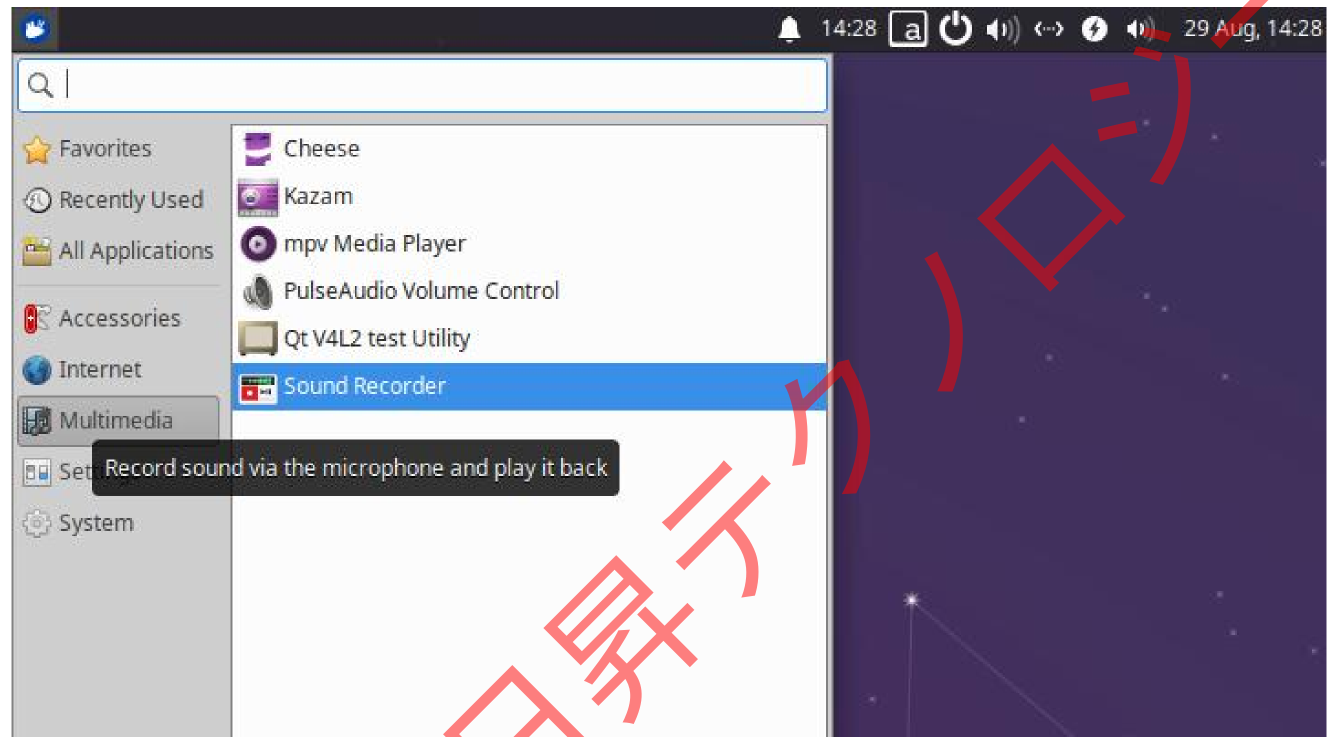


株式会社日昇テクノロジー

26.2.2 録音

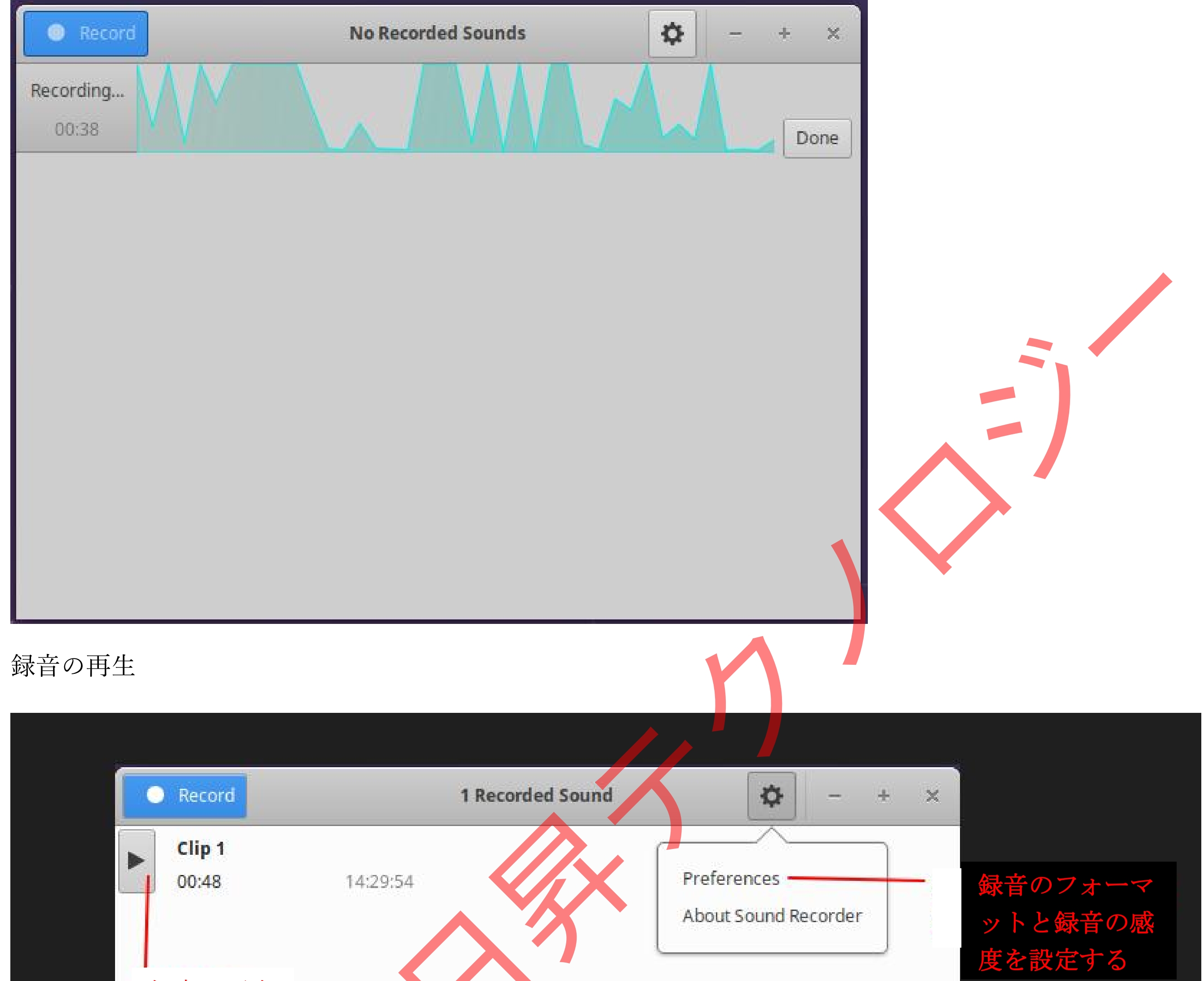
```
1 # ソフトウェアのインストール

2 sudo apt install gnome-sound-recorder
```

ソフトウェアを開く



左上角の record を押すと録音が始まり、done を押すと録音が終了します。録音されたファイルは /home/cat/Recordings に保存されます。録音中に波形が表示されない場合は、イヤホンが正しく接続されているか確認する必要があります。

株式会社日昇テクノロジー



録音の再生

もちろん、他にもっと専門的でパワフルな録音ソフトウェアをインストールすることもできます。

26.3 コマンドラインでの音声再生

26.3.1 WAV 形式の音楽の再生

インターネットから好きな WAV 形式の曲を探してダウンロードし、デバイスにコピーした後、以下のコマンドで再生できます。

株式会社日昇テクノロジー

```
1 # 注意：aplay は WAV 形式の音楽のみ再生可能で、MP3 形式は再生できません

2 aplay xxx.wav
```

```
1 # デモンストレーション

2 cat@lubancat:~$ aplay 02 lemon.wav 120

3 '02 Lemon.wav' を再生中: Signed 16 bit Little Endian, Rate 44100 Hz, Stereo
```

インターネット環境がない場合は、先に録音して WAV ファイルを生成してから再生できます。

26.3.2 録音

arecord を使用して録音します。

```
1 arecord -f cd -d 10 test.wav
```

- -f は録音品質を設定します。cd は CD 品質の録音を意味します。
- d は録音時間を指定します。単位は秒です。
- test.wav は生成される録音ファイル名です。

26.3.3 複数形式の再生

alsa-utils は WAV 形式の音声のみ再生可能です。他の形式の音声を再生したい場合は、SoX プレーヤーの使用を推奨します。

SoX は、様々な形式の音声ファイルを必要な他の形式に変換できるクロスプラットフォームのコマンドラインユーティリティです。SoX は、入力された音声ファイルに様々なエフェクトを適用することもでき、ほとんどのプラットフォームでの音声の再生と録音もサポートしています。

このセクションでは、簡単なテスト再生についてのみ説明します。詳細は公式ドキュメントを参照してください：http://sox.sourceforge.net/

以下のコマンドで sox および libsox-fmt-all をインストールします。libsox-fmt-all には様々な形式のデコーダが含まれています。

株式会社日昇テクノロジー

```
1 sudo apt install sox libsox-fmt-all
```

ソフトウェアをインストールした後、再生したい音声ファイルをデバイスにコピーし、play コマンドで様々な形式の音声ファイルを再生できます。

26.3.3.1 音声テストの再生

MP3 音声形式のファイルを再生

```
1 cat@lubancat:~$ play 01 Lemon.mp3

2

3 01 Lemon.mp3:

4

5 ファイルサイズ: 11.9M ビットレート: 320k

6 エンコーディング: MPEG オーディオ 情報: 2004

7 チャンネル: 2 @ 16 ビット トラック: 02

8 サンプルレート: 44100Hz アルバム: Lemon

9 Replaygain: off アーティスト: 米津玄師

10 持続時間: 00:04:57.29 タイトル: Lemon

11

12 入力:5.47% 00:00:16.25 [00:04:41.04] 出力:717k [-=====|=====-] Hd:0.0 Clip:0
```

FLAC 音声形式のファイルを再生

株式会社日昇テクノロジー

```
1 cat@lubancat:play 03 Lemon.flac

2

3 03 Lemon.flac:

4

5 ファイルサイズ: 36.2M ビットレート: 974k

6 エンコーディング: FLAC

7 チャンネル: 2 @ 16  ビット トラック: 0

8 サンプルレート: 44100Hz アルバム: Lemon

9 Replaygain: off アーティスト:

10 持続時間: 00:04:57.20 タイトル: Lemon

11

12 入力:3.59% 00:00:10.68 [00:04:46.52] 出力:471k [ ====|===- ] Hd:0.0 Clip:0
```

WAV 音声形式のファイルを再生

```
1 cat@lubancat:play 02 Lemon.wav

2

3 02 Lemon.wav:

4

5 ファイルサイズ: 52.4M ビットレート: 1.41M

6 エンコーディング: Signed PCM

7 チャンネル: 2 @ 16  ビット

8 サンプルレート: 44100Hz

9 Replaygain: off
```

株式会社日昇テクノロジー

```
10 持続時間: 00:04:57.20

11

12 入力:3.81% 00:00:11.33 [00:04:45.87] 出力:500k [ =====|======] Hd:0.0 Clip:0
```

26.3.3.2 ffplay

ffplay は FFMpeg に付属するプレーヤーで、ffmpeg のデコーディングライブラリとビデオレンダリング表示用の sdl ライブラリを使用しており、業界のプレーヤーの設計基準として最初に参考にされたものです。

ffplay は音声だけでなくビデオも再生できるため、音声・映像関連の開発に携わる専門家にとっては基本的なツールの一つです。

LubanCat-RK のボードには、システム上に ffmpeg ツールキットがプリインストールされているため、インストールする必要はありません。

ffplay を使用するにはグラフィカルインターフェースが必要で、X server を搭載したグラフィカルインターフェースを使用していない SSH 接続で接続した場合、正常に再生することはできません。（例：mobaxterm）ここでは、mobaxterm を使用して LubanCat ボードに SSH 接続し、X server サービスを起動することを推奨します。そうすることで、映像を視聴したり音楽を再生したりすることができます。

```
1 # ffplay は多種多様な音声フォーマットに対応しています

2 ffplay xxx.mp3
```

以下のように表示されます

株式会社日昇テクノロジー

```
^Ccat@lubancat:~$ ffplay  01.mp3
ffplay version 4.2.4-1ubuntu0.1 Copyright (c) 2003-2020 the FFmpeg developers
  built with gcc 9 (Ubuntu 9.3.0-17ubuntu1~20.04)
  configuration: --prefix=/usr --extra-version=1ubuntu0.1 --toolchain=hardened --libdir=/usr/lib/aarch64-linux-gnu --incdir=/usr/include/aarch64-linux-gnu --arch=arm64 --enable-gpl --disable-stripping --enable-avresample --disable-filter=resample --enable-avisynth --enable-gnutls --enable-ladspa --enable-libaom --enable-libass --enable-libbluray --enable-libbs2b --enable-libcaca --enable-libcdio --enable-libcodec2 --enable-libflite --enable-libfontconfig --enable-libfreetype --enable-libfribidi --enable-libgme --enable-libgsm --enable-libjack --enable-libmp3lame --enable-libmysofa --enable-libopenjpeg --enable-libopenmpt --enable-libopus --enable-libpulse --enable-librsvg --enable-librubberband --enable-libshine --enable-libsnappy --enable-libsoxr --enable-libspeex --enable-libssh --enable-libtheora --enable-libtwolame --enable-libvidstab --enable-libvorbis --enable-libvpx --enable-libwavpack --enable-libwebp --enable-libx265 --enable-libxml2 --enable-libxvid --enable-libzmq --enable-libzvbi --enable-lv2 --enable-omx --enable-openal --enable-opencl --enable-opengl --enable-sdl2 --enable-libdc1394 --enable-libdrm --enable-libiec61883 --enable-chromaprint --enable-frei0r --enable-libx264 --enable-libdrm --enable-librga --enable-rkmpp --enable-version3 --disable-libopenh264 --disable-vaapi --disable-vdpau --disable-decoder=h264_v4l2m2m --disable-decoder=vp8_v4l2m2m --disable-decoder=mpeg2_v4l2m2m --disable-decoder=mpeg4_v4l2m2m --enable-shared --disable-doc
  libavutil      56. 31.100 / 56. 31.100
  libavcodec     58. 54.100 / 58. 54.100
  libavformat    58. 29.100 / 58. 29.100
  libavdevice    58.  8.100 / 58.  8.100
  libavfilter     7. 57.100 /  7. 57.100
  libavresample   4.  0.  0 /  4.  0.  0
  libswscale      5.  5.100 /  5.  5.100
  libswresample   3.  5.100 /  3.  5.100
  libpostproc    55.  5.100 / 55.  5.100
Input #0, mp3, from '01.mp3':
  Metadata:
    title           :
    album           :
    artist          :
    genre           : Pop
    encoder         : Lavf59.6.100
    track           : 02
    date            : 2004
  Duration: 00:04:57.25, start: 0.025057, bitrate: 320 kb/s
    Stream #0:0: Audio: mp3, 44100 Hz, stereo, fltp, 320 kb/s
    Metadata:
      encoder         : Lavc59.12
  15.75 M-A: -0.000 fd=   0 aq=   43KB vq=    0KB sq=    0B f=0/0
```

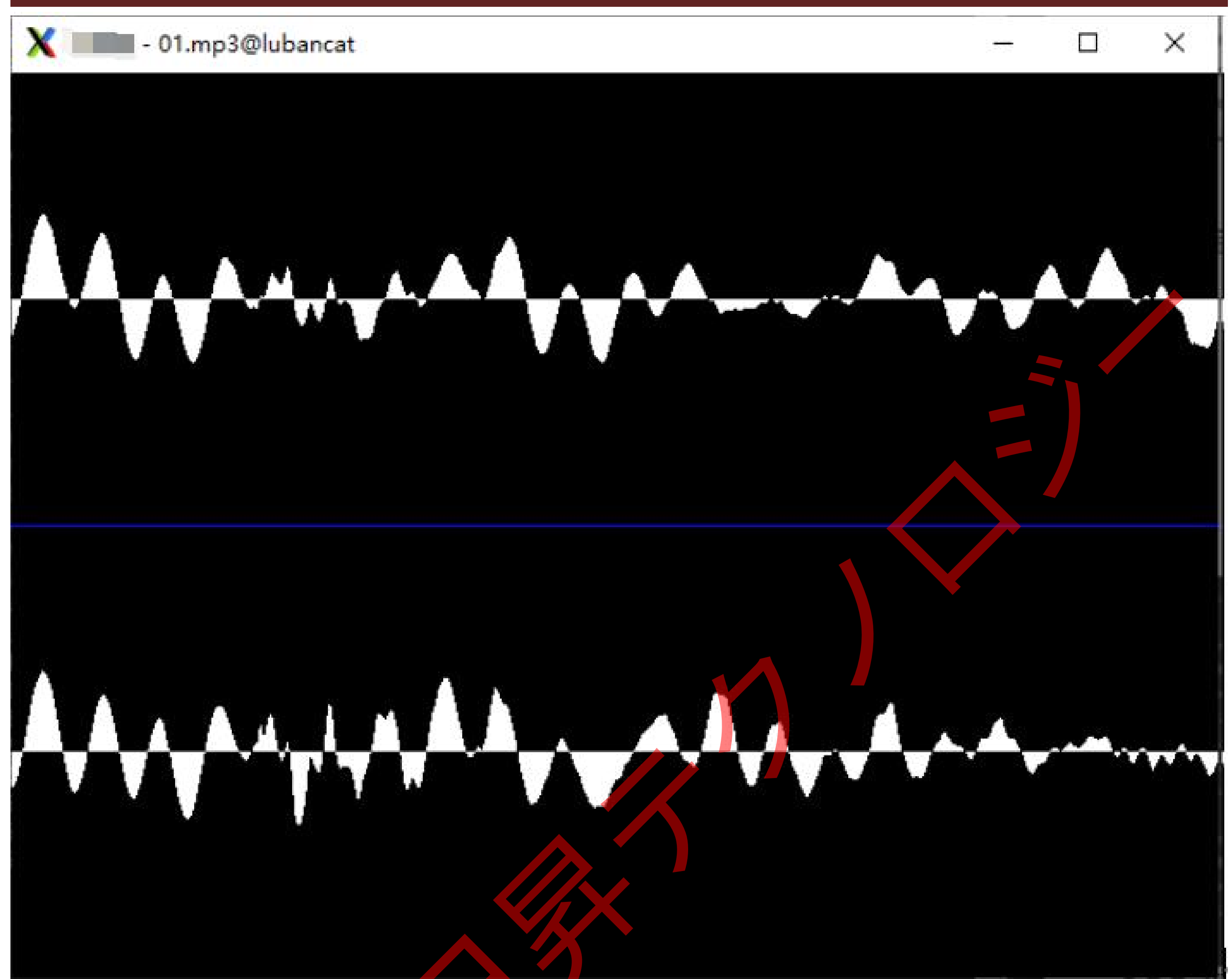
SSH で LubanCat-RK ボードに接続する場合、X server サービスを備えたリモート接続クライアントは音楽のスペクトルグラフを自動的に開きます。

デフォルトの方法の他に、設定オプションを通じて

```
1 # あらゆる再生グラフィックを閉じる

2 ffplay -showmode 0 01.mp3

3 # 音楽の波形グラフを再生する

4 ffplay -showmode 1 01.mp3

5 # 音楽のスペクトルグラフを再生する

6 ffplay -showmode 2 01.mp3
```

波形グラフは以下の通りです

株式会社日昇テクノロジー



第 27 章 スクリーン

LubanCat-RK のボードは、MIPI スクリーンと HDMI ディスプレイの表示をサポートしており、一部のボードはダブル MIPI スクリーンをサポートしています。

27.1 回転タッチスクリーン

タッチ回転方向の設定

```
1. タッチ入力デバイスを確認する
2. xinput list
```

注意: デスクトップ版のイメージを使用し、デスクトップにログインしてからコマンドを実行してください。また、ssh などのリモートログインを使用する場合、画面環境変数がリモー

株式会社日昇テクノロジー

ト端末にないため、端末で export DISPLAY=:0 コマンドを実行して画面環境変数を追加してから画面情報を取得してください。

以下の図のように、Goodix Capacitive TouchScreen デバイスが表示されます。

```
root@lubancat:~# xinput list
⎡ Virtual core pointer                          id=2    [master pointer  (3)]
⎜   ↳ Virtual core XTEST pointer                id=4    [slave  pointer  (2)]
⎜   ↳ SIGMACHIP Usb Mouse                       id=7    [slave  pointer  (2)]
⎜   ↳ Goodix Capacitive TouchScreen             id=10   [slave  pointer  (2)]
⎣ Virtual core keyboard                         id=3    [master keyboard (2)]
    ↳ Virtual core XTEST keyboard               id=5    [slave  keyboard (3)]
    ↳ adc-keys                                  id=6    [slave  keyboard (3)]
    ↳ rk805 pwrkey                              id=8    [slave  keyboard (3)]
    ↳ fdd70030.pwm                              id=9    [slave  keyboard (3)]
    ↳ rk-headset                                id=11   [slave  keyboard (3)]
    ↳ Goodix Capacitive TouchScreen             id=12   [slave  keyboard (3)]
```

1. 名前で変数を一覧表示する

xinput list-props 'pointer:Goodix Capacitive TouchScreen' | grep "Coordinate Transformation Matrix"

2. ID で変数を一覧表示する

xinput list-props 10 | grep "Coordinate Transformation Matrix"

3. タッチ方向を変更する（名前で変更可能） デフォルト方向

xinput set-prop 10 'Coordinate Transformation Matrix' 1 0 0 0 1 0 0 0 1

4. タッチ方向を変更する（名前で変更可能） 左に 90 度回転

xinput set-prop 10 'Coordinate Transformation Matrix' 0 -1 1 1 0 0 0 0 1

5. タッチ方向を変更する（名前で変更可能） 右に 90 度回転

xinput set-prop 10 'Coordinate Transformation Matrix' 0 1 0 -1 0 1 0 0 1

6. タッチ方向を変更する（名前で変更可能） 180 度回転

xinput set-prop 10 'Coordinate Transformation Matrix' -1 0 1 0 -1 1 0 0 1

27.2 タッチスクリーンのスクリーンへのバインド

1. タッチ入力デバイスを確認する

xinput list

株式会社日昇テクノロジー

以下の図のように、Goodix Capacitive TouchScreen デバイスが表示されます。

```
root@lubancat:~# xinput list
⎡ Virtual core pointer                          id=2    [master pointer  (3)]
⎜   ↳ Virtual core XTEST pointer                id=4    [slave  pointer  (2)]
⎜   ↳ SIGMACHIP Usb Mouse                       id=7    [slave  pointer  (2)]
⎜   ↳ Goodix Capacitive TouchScreen             id=10   [slave  pointer  (2)]
⎣ Virtual core keyboard                         id=3    [master keyboard (2)]
    ↳ Virtual core XTEST keyboard               id=5    [slave  keyboard (3)]
    ↳ adc-keys                                  id=6    [slave  keyboard (3)]
    ↳ rk805 pwrkey                              id=8    [slave  keyboard (3)]
    ↳ fdd70030.pwm                              id=9    [slave  keyboard (3)]
    ↳ rk-headset                                id=11   [slave  keyboard (3)]
    ↳ Goodix Capacitive TouchScreen             id=12   [slave  keyboard (3)]
```

#現在のスクリーンを確認する

xrandr -q

DSI-1：mipi スクリーン

HDMI-1：HDMI スクリーン

eDP-1：edp スクリーン

#使用しているのは DSI-1 なので、以下の設定は DSI-1 を基にしています。

#上記の Goodix Capacitive TouchScreen のタッチスクリーン ID が 10 であると仮定します。

#次に、タッチスクリーンの ID を使用してスクリーンにバインドします。

xinput map-to-output 10 DSI-1

これでタッチスクリーンのマッピングがサポートされていることが確認できます。

27.3 スクリーンの回転方向

27.3.1 コマンドライン設定

#現在のスクリーンを確認する

xrandr -q

DSI-1：mipi スクリーン

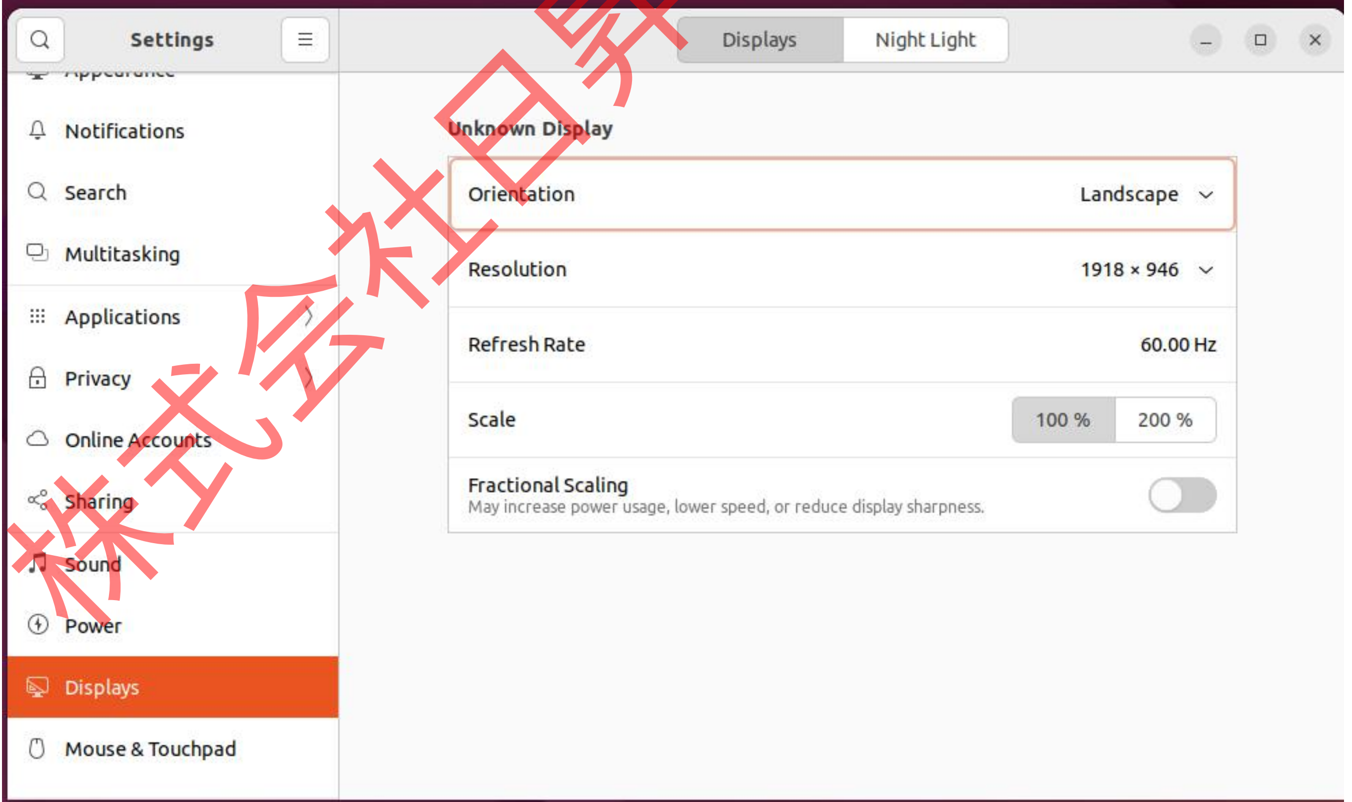
HDMI-1：HDMI スクリーン

株式会社日昇テクノロジー

```
#使用しているのは DSI-1 なので、以下の設定は DSI-1 を基にしています。

#スクリーンを通常に回転

  xrandr --output DSI-1 --rotate normal

#スクリーンを左に 90 度回転

  xrandr --output DSI-1 --rotate left

#スクリーンを右に 90 度回転

  xrandr --output DSI-1 --rotate right

#スクリーンを 180 度回転

  xrandr --output DSI-1 --rotate inverted
```

27.3.2 デスクトップ設定

デスクトップのディスプレイソフトを開き、以下の図のように設定を行います。



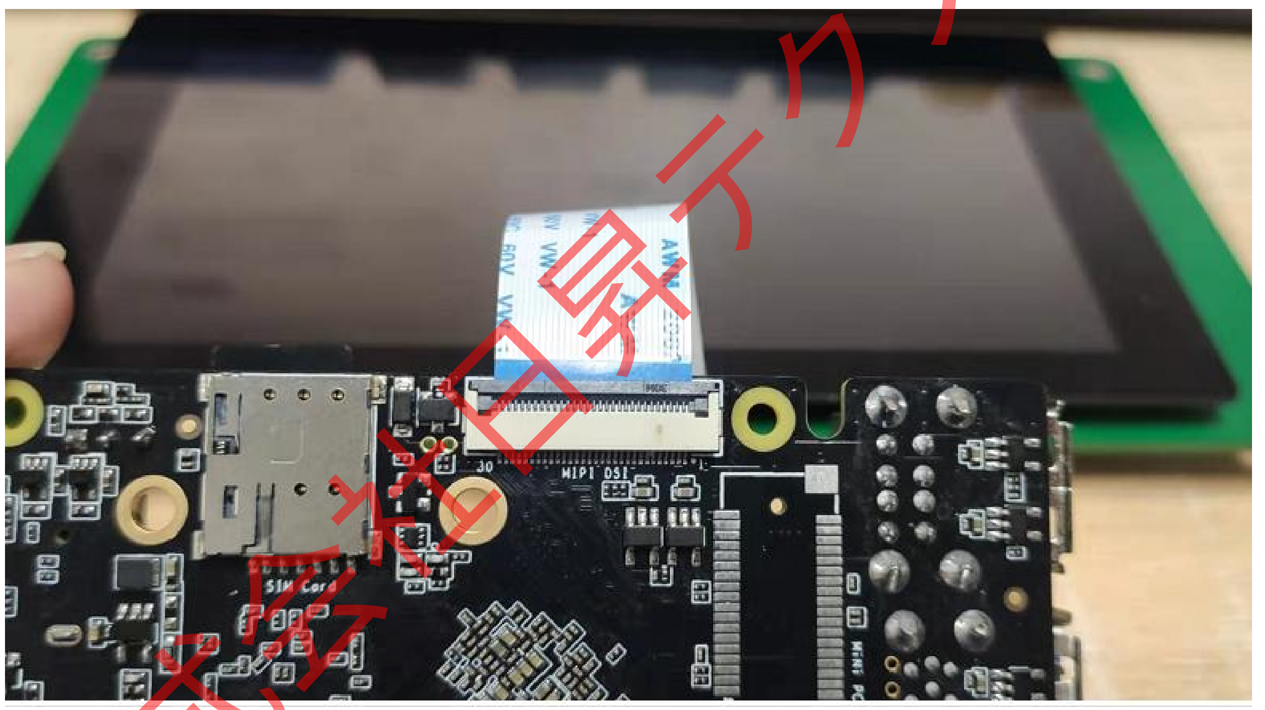
株式会社日昇テクノロジー

27.4 スクリーンインターフェース

27.4.1 mipi-dsi インターフェース

• LubanCat-RK のボードカードの mipi インターフェースのタイプはすべて同じで、30 ピンの fpc コネクタを使用しています、以下の図のように

mipi 画面との接続は以下の図のようになります



注意: mipi-dsi インターフェースはホットプラグに対応していません。電源が入っているときに画面の取り付けや取り外しをしないでください。電源が入っている状態での取り付けや取り外しは、ボードカードのショートサーキットを引き起こす可能性があり、軽い場合はボードカードが保護シャットダウンを行い、重い場合はチップのインターフェースが損傷したり、チップが焼ける可能性があります。

株式会社日昇テクノロジー

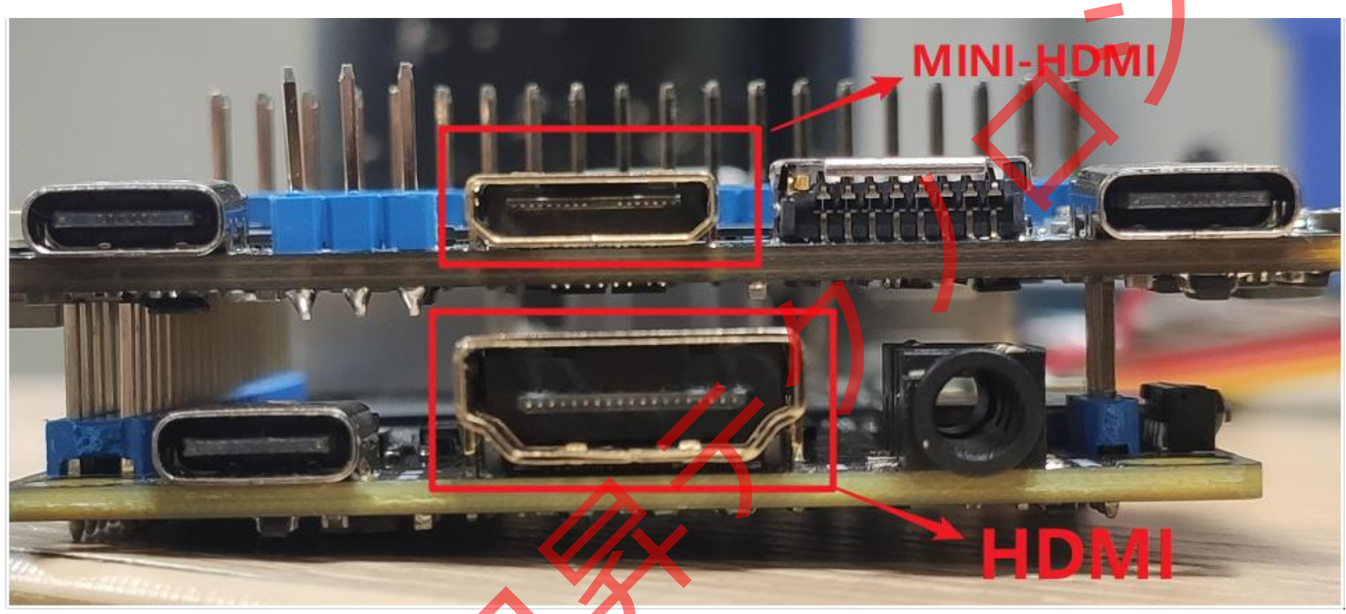
27.4.2 HDMI インターフェース

HDMI インターフェースはホットプラグに対応しています。

2 種類の HDMI インターフェースがあります:

- MINI-HDMI: LubanCat に搭載されており、外観が洗練されておりコンパクトです。

対比図は以下の通りです。



注意: HDMI と MINI-HDMI のインターフェースのサイズと形状はかなり異なるため、直接接続することはできません。HDMI と MINI-HDMI を接続して使用する場合は、HDMI から MINI-HDMI への変換アダプタを購入する必要があります。

27.5 スクリーン切り替え

デバイスツリーは、ボードがスクリーンを制御する主要な方法であり、デバイスツリーを切り替えることでスクリーンと HDMI を切り替えることができます。

27.5.1 MIPI スクリーン

設定ファイルを開きます

vi /boot/uEnv/uEnv.txt

株式会社日昇テクノロジー

MIPI DSI1 と MIPI DSI0 の起動を下の図のように設定できます。設定方法はカメラと同じです。

```
## vp0
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-hdmi0-8k-overlay.dtbo
## vp1
#
## vp2
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dp0-in-vp2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi0-800x1280-overlay.dtbo
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi0-1024x600-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi0-1080p-overlay.dtbo
## vp3
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi1-800x1280-overlay.dtbo
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi1-1024x600-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi1-1080p-overlay.dtbo

# CAM0
```

27.5.2 HDMI

設定ファイルを開きます

vi /boot/uEnv/uEnv.txt

起動を下の図のように設定できます。設定方法はカメラと同じです。

```
## vp0
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-hdmi0-8k-overlay.dtbo
## vp1
#
## vp2
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dp0-in-vp2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi0-800x1280-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi0-1024x600-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi0-1080p-overlay.dtbo
## vp3
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi1-800x1280-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi1-1024x600-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-dsi1-1080p-overlay.dtbo

# CAM0
dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam0-imx415-overlay.dtbo
# CAM1
```

27.5.3 複数スクリーン異なる表示

MIPI と MIPI 或いは MIPI と HDMI、同時動作できます。設定方法は上と同じです。

株式会社日昇テクノロジー

27.6 デスクトップログインを無効にし、スクリーンをオンに保つ

通常、スクリーンをオンにすると、新しいターミナルとしてスクリーンが使用されますが、以下はスクリーンを使用しながらスクリーンターミナルを無効にする方法です。

```
1 # ユーザーグラフィックインターフェースを無効にし、tty でログインします。

2

3 sudo systemctl set-default multi-user.target

4 sudo reboot

5

6 # ユーザーグラフィックインターフェースを有効にします。

7

8 sudo systemctl set-default graphical.target

9 sudo reboot
```

第 28 章 スクリーン表示（フレームバッファ）

28.1 LubanCat-RK ボード

このチュートリアルは、LubanCat-RK のボードに適用され、MIPI スクリーンおよび HDMI スクリーンを使用することができます。

スクリーンを使用する前に、ボードのメインデバイスツリーがスクリーンデバイスに対応していることを確認してください。

この章のサンプルコードのディレクトリは：base_linux/screen/framebuffer です。

28.2 フレームバッファの紹介

フレームバッファは、2.2.xx カーネルに登場したドライバプログラムインターフェースです。メインデバイス番号は 29 で、サブデバイス番号が増加します。

株式会社日昇テクノロジー

Linux はフレームバッファというデバイスを抽象化して、ユーザー空間のプロセスが直接スクリーンに書き込むことを可能にします。フレームバッファメカニズムは、グラフィックカードの機能を模倣し、グラフィックカードのハードウェア構造を抽象化し、フレームバッファの読み書きを通じて直接ビデオメモリに操作を行えるようにします。ユーザーは、フレームバッファをディスプレイメモリのイメージと見なし、それをプロセスアドレス空間にマッピングした後、直接読み書き操作を行うことができ、書き込み操作は即座にスクリーンに反映されます。この操作は抽象的で統一されています。

ユーザーは、物理ビデオメモリの位置、ページ交換メカニズムなどの具体的な詳細を気にする必要はありません。これらはすべてフレームバッファデバイスドライバによって完了されます。

フレームバッファは、実際には GPU に特別に予約された一連の連続した物理メモリで、LCD は専用バスを通じてフレームバッファからデータを読み取り、スクリーンに表示します。

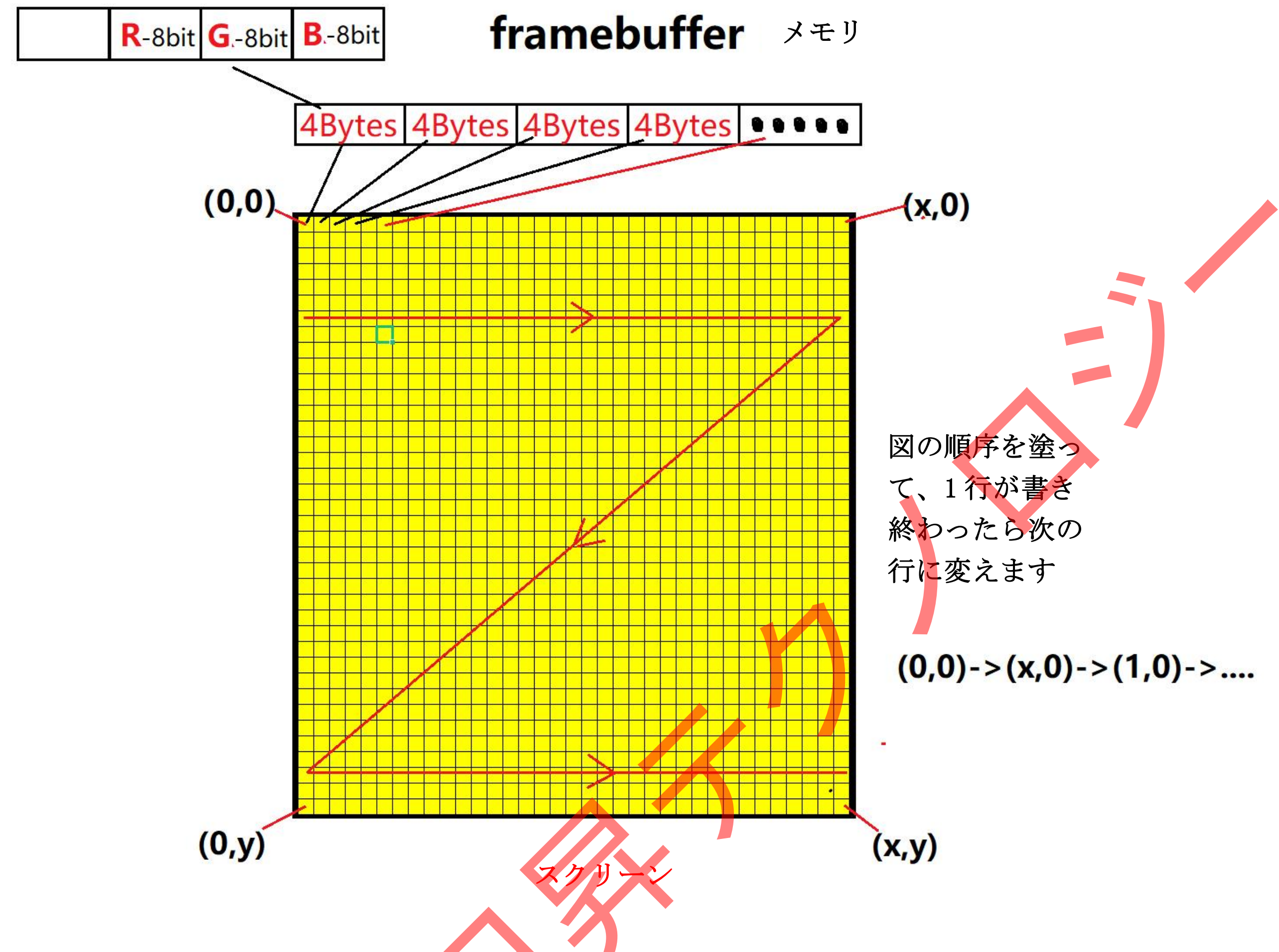
フレームバッファは本質的には表示キャッシュであり、表示キャッシュに特定の形式のデータを書き込むことは、スクリーンに内容を出力することを意味します。つまり、フレームバッファは白板のようなものです。

スクリーンの位置は上から下、左から右へとメモリアドレスと順序的な線形関係にあります。

スクリーンは RGB888 形式の他に、他の形式も持っています。

株式会社日昇テクノロジー

RG 888 を例にとる



画面は RGB888 フォーマットの他にも、以下のようなフォーマットを持っています。

1 ARGB888: |AAAAAAAAA|RRRRRRRRR|GGGGGGGG|BBBBBBBB|

2 RGB888: | |RRRRRRRRR|GGGGGGGG|BBBBBBBB|

3 RGB565: |RRRRRGGGGG|GGGBBBBB|

4 RGB555: |0RRRRRGGGG|GGGBBBBB|

株式会社日昇テクノロジー

28.3 フレームバッファアプリケーション

前書き：

注意: デスクトップ版のイメージを使用している場合は、フレームバッファを使用する前にグラフィカルインターフェースを閉じる必要があります。そうしないと、タッチ後にスクリーンが絶えず点滅する状況が発生する可能性があります。

```
1 # ユーザーグラフィカルインターフェースを閉じる

2

3 sudo systemctl set-default multi-user.target

4 sudo reboot

5

6 # ユーザーグラフィカルインターフェースを開く

7

8 sudo systemctl set-default graphical.target

9 sudo reboot
```

リスト 1: base_linux/screen/framebuffer/framebuffer.c

```
1 #include <stdio.h>

2 #include <sys/types.h> //open に必要なヘッダーファイル

3 #include <sys/stat.h>

4 #include <fcntl.h>

5 #include <unistd.h> //write

6 #include <sys/types.h>

7 #include <sys/mman.h> //mmap メモリマッピングに関する関数ライブラリ

8 #include <stdlib.h> //malloc free 動的メモリ割り当てと解放のためのヘッダーファイル

9 #include <string.h>

10 #include <linux/fb.h>

11 #include <sys/ioctl.h>

12
```

株式会社日昇テクノロジー

```
13 //32 ビットの色
14 #define Black 0x00000000
15 #define White 0xffFFFFFF
16 #define Red 0xffFF0000
17 #define Green 0xff00ff00
18 #define Blue 0xff99ffff
19
20 int fd;
21 unsigned int *fb_mem = NULL; //フレームバッファのビット数を 32 ビットに設定
22 struct fb_var_screeninfo var;
23 struct fb_fix_screeninfo fix;
24
25 int main(void)
26 {
27 unsigned int i;
28 int ret;
29
30 /*--------------ステップ 1--------------*/
31 fd = open("/dev/fb0",O_RDWR); //フレームバッファデバイスを開く
32 if(fd == -1){
33 perror("Open LCD");
34 return -1;
```

株式会社日昇テクノロジー

```
35 }
36 /*--------------ステップ 2--------------*/
37
38 //画面の可変パラメータを取得
39 ioctl(fd, FBIOGET_VSCREENINFO, &var);
40 //画面の固定パラメータを取得
41 ioctl(fd, FBIOGET_FSCREENINFO, &fix);
42
43 //解像度を表示
44 printf("xres= %d,yres= %d ¥n",var.xres,var.yres);
45 //総バイト数と 1 行の長さを表示
46 printf("line_length=%d,smem_len= %d ¥n",fix.line_length,fix.smem_len);
47 printf("xpanstep=%d,ypanstep= %d ¥n",fix.xpanstep,fix.ypanstep);
48
49 /*--------------ステップ 3--------------*/
50
51 fb_mem = (unsigned int *)mmap(NULL, var.xres*var.yres*4, //フレームバッファを取得し、メモリにマ
ッピング
52 PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
53
54 if(fb_mem == MAP_FAILED){
55 perror("Mmap LCD");
```

株式会社日昇テクノロジー

```
56 return -1;
57 }
58
59 memset(fb_mem,0xff,var.xres*var.yres*4); //画面をクリア
60 sleep(1);
61 /*--------------ステップ 4--------------*/
62 //画面を全て青色に設定
63 for(i=0;i< var.xres*var.yres ;i++)
64 fb_mem[i] = Blue;
65 sleep(2);
66 memset(fb_mem,0x00,var.xres*var.yres*4); //画面をクリア
67
68 munmap(fb_mem,var.xres*var.yres*4); //mmap でマッピングしたアドレスを解除
69 close(fd); //fb0 デバイスファイルを閉じる
70 return 0;
71 }
```

株式会社日昇テクノロジー

コンパイルして実行します。

```
1 gcc framebuffer.c -o framebuffer
2 ./framebuffer
3
4 # 実行結果
5 # スクリーン情報が表示される
6 # スクリーンが白になる
7 # 1 秒待つ
8 # スクリーンが青に変わる
```

28.3.1 コード分析

スクリーン操作には全体で四つのステップが必要です。

リスト 2: 第一ステップ

```
1 /*--------------ステップ 1--------------*/
2 fd = open("/dev/fb0",O_RDWR); //フレームバッファデバイスを開く
3 if(fd == -1){
4 perror("Open LCD");
5 return -1;
6 }
```

- このステップはフレームバッファデバイスを開き、成功するとファイル記述子 fd が返されます。これを使ってデバイスを操作します。

株式会社日昇テクノロジー

リスト 3: 第二ステップ

```
1 /*--------------ステップ 2--------------*/

2

3 //画面の可変パラメータを取得

4 ioctl(fd, FBIOGET_VSCREENINFO, &var);

5 //画面の固定パラメータを取得

6 ioctl(fd, FBIOGET_FSCREENINFO, &fix);

7

8 //解像度を表示

9 printf("xres= %d,yres= %d ¥n",var.xres,var.yres);

10 //総バイト数と 1 行の長さを表示

11 printf("line_length=%d,smem_len= %d ¥n",fix.line_length,fix.smem_len);

12 printf("xpanstep=%d,ypanstep= %d ¥n",fix.xpanstep,fix.ypanstep);
```

- このステップではスクリーンのパラメータを取得し、設定します。

- スクリーンの可変パラメータ`ioctl(fd, FBIOGET_VSCREENINFO, &var);`

- スクリーンの固定パラメータ`ioctl(fd, FBIOGET_FSCREENINFO, &fix);`

- これらの目的は`fb_var_screeninfo`、`fb_fix_screeninfo`構造体を取得することです。

- これらの構造体に含まれる内容には、固定パラメータと可変パラメータがあります。

株式会社日昇テクノロジー

```
1 //固定パラメータ

2 struct fb_fix_screeninfo {

3 char id[16]; /* 識別文字列 例 "TT Builtin" */

4 unsigned long smem_start; /* mmap 後のメモリの開始アドレス */

5 /* (物理アドレス) */

6 __u32 smem_len; /* framebuffer mmap の最大長 */

7 __u32 type; /* FB_TYPE_* を参照 */

8 __u32 type_aux; /* インターリーブ用 */

9 __u32 visual; /* FB_VISUAL_* を参照 */

10 __u16 xpanstep; /* ハードウェアパニングがない場合は 0 */

11 __u16 ypanstep; /* ハードウェアパニングがない場合は 0 */

12 __u16 ywrapstep; /* ハードウェア ywrap がない場合は 0 */

13 __u32 line_length; /* 1 行あたりのピクセルの長さ */

14 unsigned long mmio_start; /* メモリマップト I/O の開始 */

15 /* (物理アドレス) */

16 __u32 mmio_len; /* メモリマップト I/O の長さ */

17 __u32 accel; /* ドライバに特定のチップ/カードを指示 */

18 __u16 capabilities; /* FB_CAP_* を参照 */

19 __u16 reserved[2]; /* 将来の互換性のために予約されている */

20 };

21

22 //可変パラメータ
```

株式会社日昇テクノロジー

```
23 struct fb_var_screeninfo {

24 __u32 xres; /* x 軸の可視領域--x 軸の実際のピクセル */

25 __u32 yres; /* y 軸の可視領域--y 軸の実際のピクセル */

26 __u32 xres_virtual; /* x 軸の仮想領域、ダブルバッファリングに使用可能 */

27 __u32 yres_virtual; /* y 軸の仮想領域、ダブルバッファリングに使用可能 */

28 __u32 xoffset; /* x 軸のオフセット--デフォルトは 0、ダブルバッファリングに使用可能 */

29 __u32 yoffset; /* y 軸のオフセット--デフォルトは 0、ダブルバッファリングに使用可能 */

30

31 __u32 bits_per_pixel; /* 1 ピクセルあたりのビット数 */

32 __u32 grayscale; /* 0 = カラー, 1 = グレースケール, >1 = FOURCC */

33

34 struct fb_bitfield red; /* fb メモリ内のビットフィールドが本当の色の場合 */

35 struct fb_bitfield green; /* それ以外の場合は長さのみが重要 */

36 struct fb_bitfield blue;

37 struct fb_bitfield transp; /* 透明度 */

38

39 __u32 nonstd; /* != 0 非標準のピクセルフォーマット */

40

41 __u32 activate; /* FB_ACTIVATE_* を参照 */

42

43 __u32 height; /* 画像の高さ(mm) */

44 __u32 width; /* 画像の幅(mm) */
```

株式会社日昇テクノロジー

```
45
46 __u32 accel_flags; /* (廃止) fb_info.flags を参照 */
47
48 /* タイミング: 全ての値はピクセルクロックで、pixclock を除く */
49 __u32 pixclock; /* ピクセルクロック(ps 単位) */
50 __u32 left_margin; /* 同期から画像までの時間 */
51 __u32 right_margin; /* 画像から同期までの時間 */
52 __u32 upper_margin; /* 同期から画像までの時間 */
53 __u32 lower_margin;
54 __u32 hsync_len; /* 水平同期の長さ */
55 __u32 vsync_len; /* 垂直同期の長さ */
56 __u32 sync; /* FB_SYNC_* を参照 */
57 __u32 vmode; /* FB_VMODE_* を参照 */
58 __u32 rotate; /* 反時計回りに回転する角度 */
59 __u32 colorspace; /* FOURCC ベースのモード用の色空間 */
60 __u32 reserved[4]; /* 将来の互換性のために予約されている */
61 };
```

より詳細な内容は、カーネルソースコードやドキュメント`kernel/Documentation/fb/api.txt`で確認できます。

- ioctl: framebuffer では FBIOGET_VSCREENINFO, FBIOGET_FSCREENINFO 以外にも、他の設定オプションがあります。

リスト 1: framebuffer ioctl オプション

オプション名	機能	用法

株式会社日昇テクノロジー

FBIOGET_VSCREENINFO	ioctl(fd, FBIOGET_VSCREENINFO, &var);	スクリーンの可変パラメータを取得
FBIOPUT_VSCREENINFO	ioctl(fd, FBIOPUT_VSCREENINFO, &var);	スクリーンの可変パラメータを変更
FBIOGET_FSCREENINFO	ioctl(fd, FBIOGET_FSCREENINFO, &fix);	スクリーンの固定パラメータを取得
FBIOPAN_DISPLAY	ioctl(fd, FBIOPAN_DISPLAY, &var);	平行表示を設定し、スクリーンの表示を変更する。ダブルバッファリング設計に使用可能
他		他のオプションについては、カーネルソースコード `/kernel/drivers/video/fbdev/core/fbmem.c` の 1113 行目から 1267 行目を参照してください。

リスト 4: 第三ステップ

```
1 unsigned int *fb_mem = NULL; //フレームバッファのビット数を 32 ビットに設定

2 /*--------------ステップ 3--------------*/

3

4 fb_mem = (unsigned int *)mmap(NULL, var.xres*var.yres*4, //フレームバッファを取得し、メモリにマッピング

5 PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

6

7 if(fb_mem == MAP_FAILED){

8 perror("Mmap LCD");

9 return -1;

10 }
```

- このステップでは、デバイスのメモリをユーザースペースにマッピングします。この操作により、こ

株式会社日昇テクノロジー

のメモリに対する操作でスクリーンの内容を変更できます。伝統的な write 関数と比較して、データ転送の速度と利便性が大幅に向上しています。

- 最初の行で fb_mem を 32 ビットに設定すると、32 ビットのデータを使用して 24 ビットのスクリーンデータ RGB888 を簡単に操作できます。空間を節約したい場合は、unsigned char 型を使用できますが、色成分を解析して個別に書き込む必要があります。

```
1. mmap の原型は以下の通りです。
2. caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset);
```

- 戻り値はアドレスです。

- addr: 要求されるアドレス。null の場合、システムは自動的にアドレス空間を割り当てます。

- len: 要求するメモリ空間のサイズで、単位はバイトです。

- prot: アクセス権を指定する prot パラメータは、PROT_READ（読み取り可能）、PROT_WRITE（書き込み可能）、PROT_EXEC（実行可能）、PROT_NONE（アクセス不可）のいずれかの「または」で取ることができます。

- flags: MAP_PRIVATE と MAP_SHARED の 2 つのパラメータから選択できます。1. MAP_PRIVATE - プライベートな書き込み時コピーのマッピングを作成し、マッピングの更新は同じファイルにマッピングされた他のプロセスには表示されず、基礎となるファイルには反映されません。

2. MAP_SHARED - このマッピングを共有し、マッピングの更新はこのファイルをマッピングしている他のプロセスに表示され、基礎となるファイルに反映されます。

- fd: ファイル記述子で、fd=-1 の場合は匿名マッピングです。

- offset: マッピングされるファイルのオフセットで、ファイルのどこから操作を開始するかを指します。

詳細な操作については、PC 上の Ubuntu でコマンドラインに「man 2 mmap」と入力して得られます。

これに対応するのは、64 行目の munmap(fb_mem, var.xres*var.yres*4)です。

株式会社日昇テクノロジー

```
1 # 関数の機能：メモリ空間を解放する

2 int munmap(void *addr, size_t length);
```

- addr: mmap から返されたアドレスです。

- length: 回収するメモリ空間のサイズです。

リスト 5: 第四ステップ

```
1 /*--------------第四ステップ--------------*/

2 // スクリーンを全て青色に設定

3 for(i=0; i < var.xres*var.yres; i++)

4     fb_mem[i] = Blue;
```

- この操作により、framebuffer を直接操作してスクリーンを制御できます。i はスクリーン上のピクセル位置に対応します。

28.4 ダブルバッファリング設計

28.4.1 前書き

- このセクションでは、LubanCat-RK ボードでこの機能を実装するために、カーネルソースコードの自己修正が必要なダブルバッファリング framebuffer のアプリケーションデザインのアイデアのみを提供します。

- ダブルバッファリングの設計は drm でより高いサポートがあります。drm アプリケーションプログラミングを使用することを推奨します。カーネルソースコードを変更したくないが、ダブルバッファリング framebuffer を体験したい場合は、次の章を学習してください。

28.4.2 ダブルバッファリング

- コンピュータ上のアニメーションと実際のアニメーションは異なります：実際のアニメーションは事前に描かれ、再生時に直接表示されます。コンピュータアニメーションは、一枚描いては表示し、次の一枚

株式会社日昇テクノロジー

を描いては表示します。描画が簡単な場合は問題ありませんが、描画に時間がかかる複雑なグラフィックの場合、問題が顕著になります。二枚の画板を使い、一枚を描いてからスクリーンに掛ける画板と交換することで、観客が不完全な画を見ることがなくなります。この技術はコンピュータグラフィックスに応用され、「ダブルバッファリング技術」と呼ばれています。つまり、メモリ（通常はビデオメモリ）に二つの領域を確保し、一つはディスプレイへ送信するデータ用、もう一つは描画用として使用し、適切なタイミングでそれらを交換します。二つのメモリ領域を交換することは、実質的には二つのポインタを交換するだけなので、この方法は非常に高効率で、広く採用されています。

横方向にスクリーンを拡張する

株式会社日昇テクノロジー

28.4.3 実装思路

カーネルの変更：

1. fb_fix_screeninfo.smem_len このパラメータを変更するが、固定パラメータはアプリケーション層で変更できないため、カーネル内でその長さを拡張する必要がある。

2. カーネル内の mmap 関数で割り当てるメモリの長さを変更し、DMA の転送範囲を変更する。

3. カーネル内の fb_pan_display を修正して表示に適応させる。

アプリケーション層の変更：

1. アプリケーション層では、xres_virtual を xres の 2 倍に拡大する必要がある。

2. メインスクリーンに表示させたい場合は、fb_var_screeninfo.xoffset=0 を設定する。副スクリーンに表示させたい場合は、fb_var_screeninfo.xoffset=xres を設定する。

3. ioctl(fd, FBIOPUT_VSCREENINFO, &var);を使用してパラメータを変更する。

4. ioctl(fd, FBIOPAN_DISPLAY, &var);を使用してパラメータをカーネルに送り込み、画面を切り替えることができる。

28.5 参考資料

フレームバッファ

・《Android Framebuffer の紹介及び使用》

・《LCD 制御原理》

ダブルバッファリング

・《Android グラフィックシステムの分析と移植–七、ダブルバッファリング framebuffer の実装》

第 29 章 画面表示(DRM) の紹介

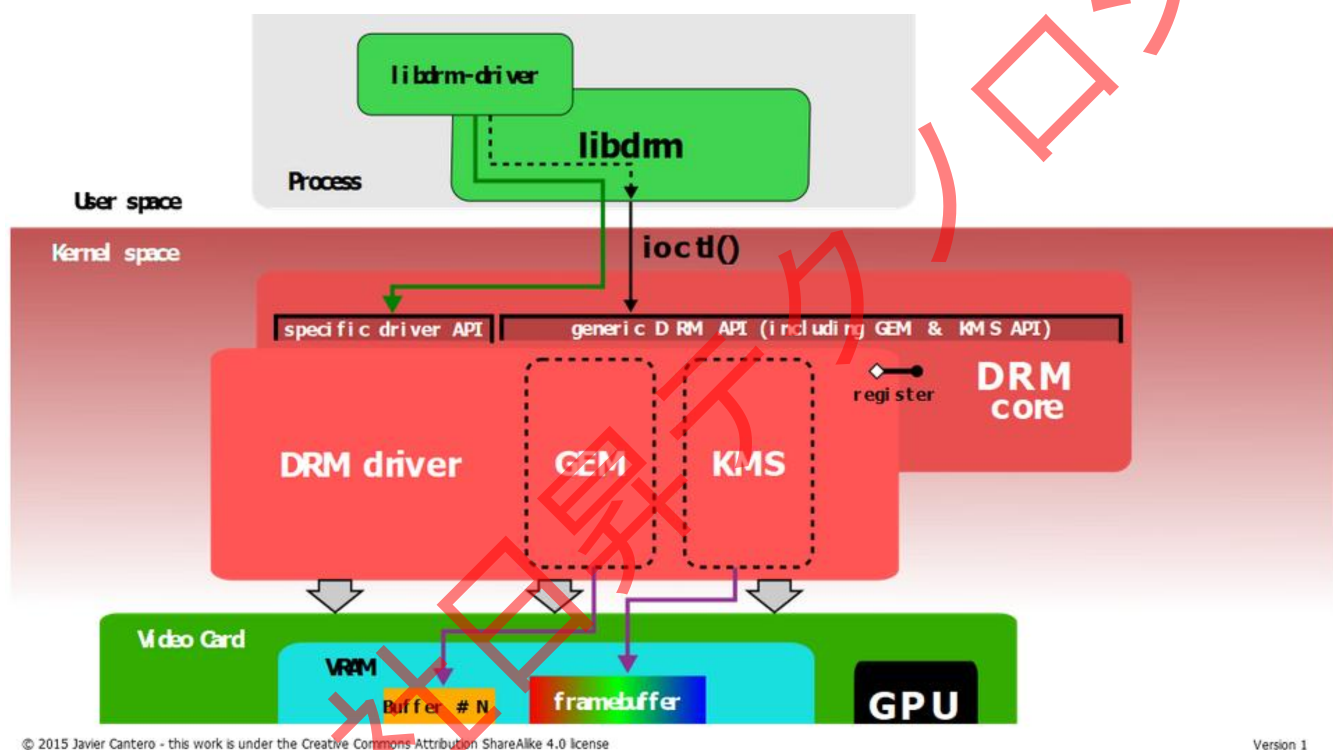
29.1 前置き

この章では《DRM(Direct Rendering Manager)》に基づいた DRM のアプリケーション開発について説明します。このには GPU、DRM ドライバの開発、グラフィックス学の関連するチュートリアルや理論分析が含まれています。

株式会社日昇テクノロジー

29.2 DRM の紹介

DRM は Linux で現在主流のグラフィック表示フレームワークで、FB アーキテクチャと比較して、現在の最新の表示ハードウェアにより適応できます。例えば、FB はネイティブに多層合成、VSYNC、DMA-BUF、非同期更新、フェンス機構などをサポートしていませんが、これらの機能は DRM でネイティブにサポートされています。同時に、DRM は GPU とディスプレイドライバを統合管理し、ソフトウェアアーキテクチャを統一し、管理とメンテナンスを容易にします。



DRM の画像システムは二つの部分に分かれています：

- アプリケーション層 - libdrm
- カーネルドライバ層 - GEM, KMS

libdrm：下層のインターフェースをラップし、上層に一般的な API インターフェースを提供します。主に様々な IOCTL インターフェースのラップです。

KMS(Kernel Mode Setting)：画面の更新と表示パラメータの設定。

1. 画面の更新：表示バッファの切り替え、複数のレイヤーの合成方法、および各レイヤーの表示位置。

2. 表示パラメータの設定：分解能、リフレッシュレート、電源状態（スリープ/ウェイクアップ）など。

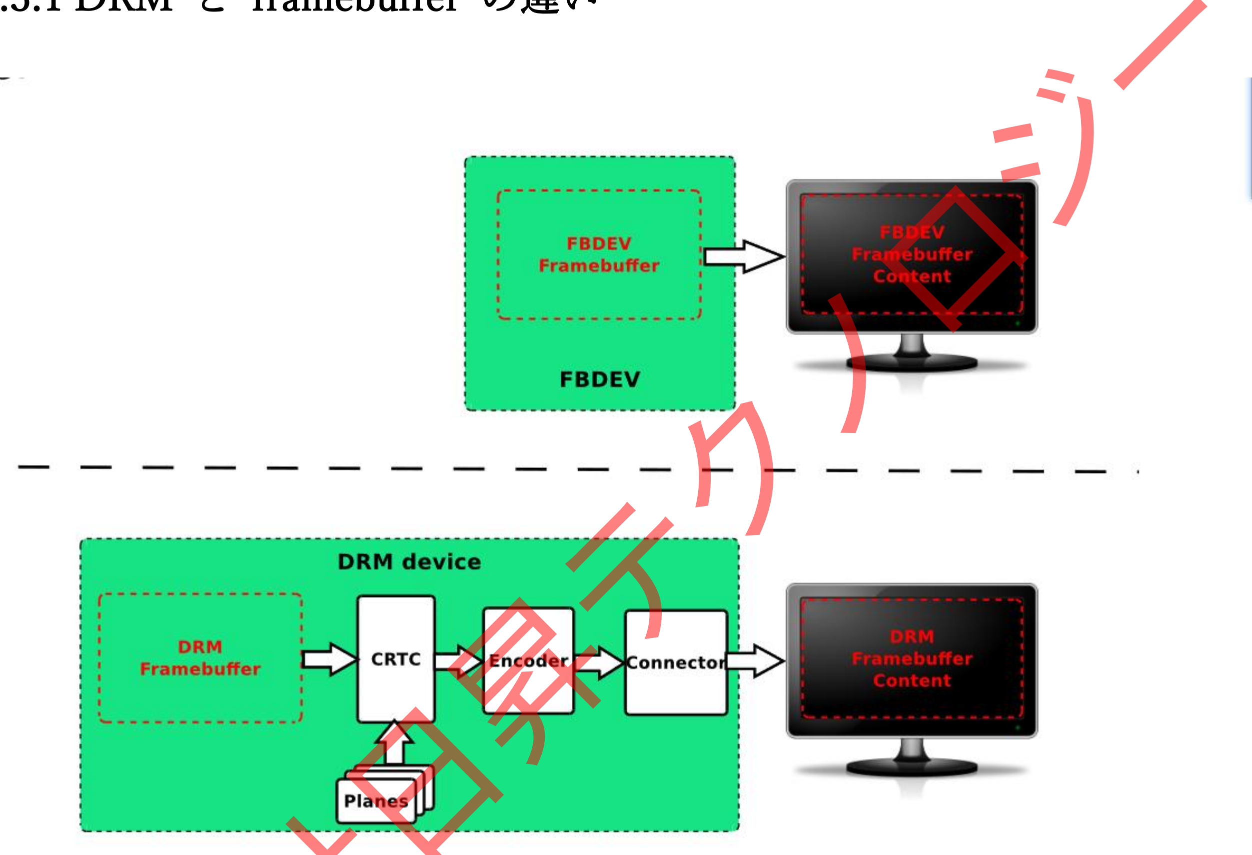
株式会社日昇テクノロジー

GEM(Graphic Execution Manager)：表示バッファの割り当てと解放、メモリ管理と同期を主に担当します。

29.3 DRM 表示

このセクションでは主に「drm-kms.pdf」の記事を引用して説明します。

29.3.1 DRM と framebuffer の違い



株式会社日昇テクノロジー

- framebuffer の使用は非常にシンプルで、ユーザースペースで framebuffer のメモリスペースを定義するだけで、直接このメモリを操作することで画面の表示を簡単に変更できます。

- DRM の場合、framebuffer とディスプレイの間には 4 つのコンポーネントがあり、framebuffer のデータはこれらのコンポーネントの共同処理を経て最終的にディスプレイに出力されます。

framebuffer と比較して、DRM には以下のような多くの不利な点があります。

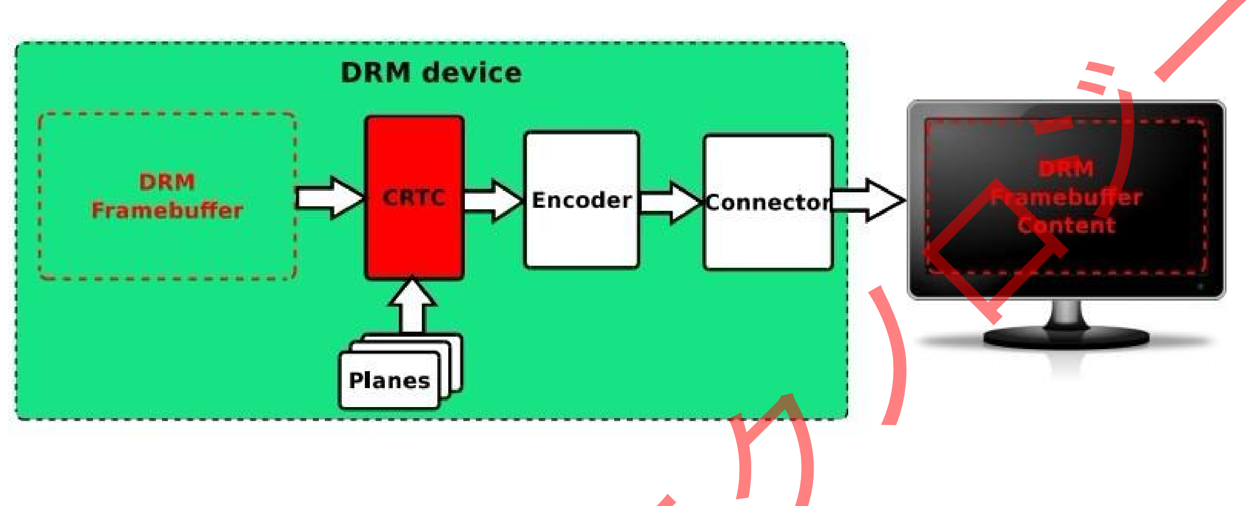
1. DRM にはより多くのコミュニティメンテナがいます。

2. DRM は表示に関してより多くの設定を提供します。

3. DRM はユーザースペースでより広範囲に活用できます。

しかし、画像がそれほど複雑でないシナリオでは、framebuffer の開発の難易度が DRM 表示システムよりも優れています。

29.3.2 DRM 表示システムの分析



緑色の枠内の 5 つのコンポーネントとその連携について分析します。

29.3.2.1 DRM Framebuffer

前章の framebuffer と同様に、画像を格納するメモリ領域であり、画像のフォーマット(RGB888,YUV,C8など)とキャンバスのサイズを設定する必要があります。

29.3.2.2 CRTC

CRTC の名前は Cathode Ray Tube Controller から来ており、陰極射線显像管控制器と言います。世界で最初のカラーテレビは CRT モニターであり、この名前を選んだ主な理由は、この部品が陰極射線显像管控制器に非常に似ているからです。電子銃から放出された電子が画面上の蛍光物質に衝突して光を放つことで、電子銃を左から右に一行スキャンする（つまり一行をスキャンする）ことにより、上から下にすべての行をスキャンすると、一枚のフレーム画像が表示されます。つまり、フレーム画像を表示する際、電子銃は「Z」形に動いており、スキャン速度が非常に速いと、完成した画像のように見えます。DRM 表示システムでは、CRTC は display timings と表示解像度（Planes が提供する）を設定して、framebuffer

株式会社日昇テクノロジー

上の内容をスキャンし、Encoder に渡します。display timings は framebuffer のスキャンタイミングで、LCD 画面の表示は 0.96 インチの画面のように、すべての表示データを直接書き込むだけで物を表示できるわけではなく、LCD 画面は正しく物を表示するために特定のタイミングが必要です。したがって、CRTC はここで非常に重要な役割を持ち、ビデオモードのタイミング信号を生成し、内容を Encoder に出力します。 Encoder と Connector はデータの変換と伝送のみを行います。LCD 画面のタイミングについては、「LCD 基礎概念（一）：LCD timing 時序参数」という記事に詳しく紹介されています。

29.3.2.3 Planes

一般に平面と訳されますが、レイヤーと訳す方が適切です。Planes は、CRTC にデータを送信するバッファブロックを含むメモリオブジェクトで、各 CRTC は少なくとも一つの Planes を関連付ける必要があります。これは、CRTC がどのビデオモードを選択するかの基準です - 表示解像度（幅と高さ）、ピクセルサイズ、ピクセルフォーマット、リフレッシュレートなど。

Planes は三つのタイプに分かれます：

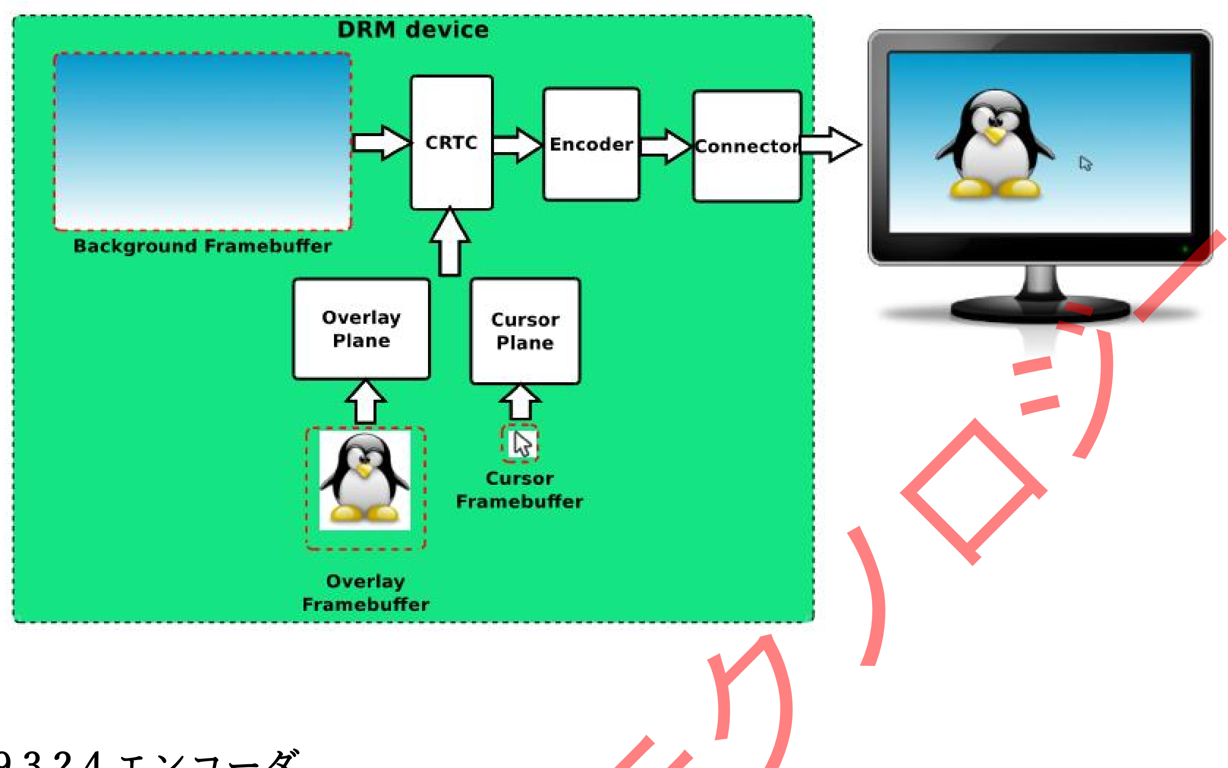
1. DRM_PLANE_TYPE_PRIMARY：主要レイヤー、背景または画像コンテンツを表示し、各 CRTC に一つ含まれます。
2. DRM_PLANE_TYPE_OVERLAY：オーバーレイ、スケーリングを表示するために使用され、各 CRTC に一つ以上含まれます。
3. DRM_PLANE_TYPE_CURSOR：マウスを表示するために使用され、各 CRTC に 0-N 個含まれます。

通常、ドライバは framebuffer を DRM_PLANE_TYPE_PRIMARY にバインドします。

LubanCat-RK のボードは三つのレイヤーを含んでおり、一つの DRM_PLANE_TYPE_PRIMARY と二つの DRM_PLANE_TYPE_OVERLAY（うち一つは正常に使用できません）が含まれます。

Planes の重ね合わせの現象は以下の通りです。

株式会社日昇テクノロジー



29.3.2.4 エンコーダ

エンコーダは、ピクセルをディスプレイに必要な信号にエンコード（変換）する役割を持ちます。異なるディスプレイに画像を出力する場合、それを DVID、VGA、YPbPr、CVBS、Mipi、eDP などの異なる電気信号に変換する必要があります。そのため、フレームを適切な形式に変換し、コネクタを通じて伝送する責任があります。例えば、HDMI コネクタは TMDS 形式のデータを使用して駆動する必要があるため、ピクセル形式を TMDS に変換できるエンコーダが必要です。

29.3.2.5 コネクタ

コネクタは、物理的な接続器（VGA, DVI, FPD-Link, HDMI, DisplayPort, S-Video など）に対応し、物理的なディスプレイ出力デバイス（モニター、ラップトップパネルなど）に接続します。現在の物理接続の出力デバイスに関連する情報（接続状態、EDID データ、DPMS 状態、サポートされるビデオモードなど）もコネクタ内に保存されます。

株式会社日昇テクノロジー

29.3.2.6 結論

これにより、DRM 表示フレームワークが非常に複雑ではないことがわかります。理解した後、表示フレームワークについて大まかな理解を持つことができ、DRM アプリケーションプログラミングを通じて、それについての理解を深めることができます。

29.3.3 LubanCat-RKdrm 分析

RK3588 を搭載したボードはシングルディスプレイのみをサポートし、複数のディスプレイを接続するとマルチディスプレイ表示を実現できません。1 つのディスプレイを接続すると、3 つのプレーン（3 番目のプレーン Cluster は AFBC 形式の画像のみサポート）があります。RK3568 を搭載したボードはマルチディスプレイ表示をサポートし、最大で 3 つのディスプレイを異なる表示でサポートできます。RK3568 の総プレーン数は 6 つです。

- 1 つのディスプレイを起動すると、3 つのレイヤー（3 番目のレイヤー Cluster は AFBC 形式の画像のみをサポート）が割り当てられます。

- 2 つのディスプレイを起動すると、各ディスプレイに 3 つのレイヤー（3 番目のレイヤー Cluster は AFBC 形式の画像のみをサポート）が割り当てられます。

- 3 つのディスプレイを起動すると、最初のディスプレイに 3 つのレイヤー、2 番目のディスプレイに 2 つのレイヤー、最後のディスプレイに 1 つのレイヤー（最初のディスプレイの 3 番目のレイヤー Cluster および 2 番目のディスプレイの 2 番目のレイヤー Cluster は AFBC 形式の画像のみをサポート）が割り当てられます。

注意：そのため、ガイドでは Cluster レイヤーを操作することはできません。使用する画像形式は RGB であり、Cluster レイヤーは AFBC 形式を必要とします。

注意：Arm Frame Buffer Compression (AFBC)は、Arm GPU/DPU/VPU とグラフィックビデオメモリ間のデータ転送用に Arm が定義した無損失画像圧縮形式で、帯域幅と消費電力を大幅に削減し、グラフィッ

株式会社日昇テクノロジー

クピクセルへのより細かいアクセスを提供します。

vp は rk3588 の表示ユニットで、合計 3 つの表示ユニットがあり、VP0, VP1, VP2 として、レイヤーの割り当てには優先順位があります、VP1>VP0>VP2 です。

3 つのディスプレイを起動した時の割り当ては以下の通りです：

- VP0→HDMI プラグアンドプレイ→2 つ（最大 4096x2304@60Hz をサポート）
- VP1→MIPI メインディスプレイ→3 つ（最大 1920x1080@60Hz をサポート）
- VP2→DP から VGA への変換、プラグアンドプレイ→1 つ（最大 1920x1080@60Hz をサポート）

カーネル内でレイヤーを割り当てることができます。

29.3.3.1 drm の関連情報の確認

```
1 # テストツールのインストール
2 sudo apt install libdrm-tests
3
4
5 # 二画面異表示の例
6 modetest
7
8 cat@lubancat:~/lubancat-test/base_linux$ modetest
9 trying to open device 'rockchip'...done
10 Encoders:
11 id crtc type possible crtcs possible clones
12 149 0 Virtual 0x00000003 0x00000000
13 151 71 TMDS 0x00000001 0x00000000
```

株式会社日昇テクノロジー

```
14 162 87 DSI 0x00000002 0x00000000

15

16 Connectors:

17 id encoder status name size (mm) modes encoders 18 152 151 connected HDMI-A-1 600x340 32 151

19 modes:

20 index name refresh (Hz) hdisp hss hse htot vdisp vss vse vtot

21 #0 1920x1080 60.00 1920 2008 2052 2200 1080 1084 1089 1125 148500 flags: phsync, pvsync; type:
preferred, driver

22 (省略.........)

23 props:

24 (省略.........)

25

26 163 162 connected DSI-1 0x0 1 162

27 modes:

28 index name refresh (Hz) hdisp hss hse htot vdisp vss vse vtot

29 #0 720x1280 60.00 720 730 736 756 1280 1290 1294 1314 59603 flags:nhsync, nvsync; type: preferred,
driver


30 props:

31 (省略.........)

32

33 CRTCs:

34 id fb pos size
```

株式会社日昇テクノロジー

35 71 167 (0,0) (1920x1080)

36 #0 1920x1080 60.00 1920 2008 2052 2200 1080 1084 1089 1125 148500flags: phsync, pvsync; type:

preferred, driver

37 props:

38 (省略.........)

39

40 87 164 (0,0) (720x1280)

41 #0 720x1280 60.00 720 730 736 756 1280 1290 1294 1314 59603 flags: nhsync, nvsync; type: preferred,

driver

42 props:

43 (省略.........)

44

45 Planes:

46 id crtc fb CRTC x,y x,y gamma size possible crtcs

47 57 71 167 0,0 0,0 0 0x00000001

48 formats: XR24 AR24 XB24 AB24 RG24 BG24 RG16 BG16

49 props:

50 8 type:

51 flags: immutable enum

52 enums: Overlay=0 Primary=1 Cursor=2

53 value: 1

54 63 NAME:

55 flags: immutable bitmask

株式会社日昇テクノロジー

```
56 values: Smart1-win0=0x2
57 value: 2
58 (省略.........)
59
60 73 87 164 0,0 0,0 0  0x00000002
61 formats: XR24 AR24 XB24 AB24 RG24 BG24 RG16 BG16
62 props:
63 8 type:
64 flags: immutable enum
65 enums: Overlay=0 Primary=1 Cursor=2
66 value: 1
67 79 NAME:
68 flags: immutable bitmask
69 values: Smart0-win0=0x1
70 value: 1
71 (省略.........)
72
73 89 0 0 0,0 0,0 0  0x00000001
74 formats: XR24 AR24 XB24 AB24 RG24 BG24 RG16 BG16 NV12 NV16 NV24 NA12 NA16 NA24 YVYU
VYUY
75 props:
76 8 type:
77 flags: immutable enum
```

株式会社日昇テクノロジー

```
78 enums: Overlay=0 Primary=1 Cursor=2
79 value: 0
80 95 NAME:
81 flags: immutable bitmask
82 values: Esmart1-win0=0x4
83 value: 4
84 (省略.........)
85
86 103 0 0 0,0 0,0 0  0x00000002
87 formats: XR24 AR24 XB24 AB24 RG24 BG24 RG16 BG16 NV12 NV16 NV24 NA12 NA16 NA24 YVYU
VYUY
88 props:
89 8 type:
90 flags: immutable enum
91 enums: Overlay=0 Primary=1 Cursor=2
92 value: 0
93 109 NAME:
94 flags: immutable bitmask
95 values: Esmart0-win0=0x8
96 value: 8
97 (省略.........)
98
99 117 0 0 0,0 0,0 0  0x00000002
```

株式会社日昇テクノロジー

```
100 formats: XR24 AR24 XB24 AB24 RG24 BG24 RG16 BG16 NV12 NV16 NV24 NA12 NA16 NA24
YUYV
101 props:
102 8 type:
103 flags: immutable enum
104 enums: Overlay=0 Primary=1 Cursor=2
105 value: 0
106 123 NAME:
107 flags: immutable bitmask
108 values: Cluster0-win0=0x10
109 value: 16
110 (省略.........)
111
112 131 0 0 0,0 0,0 0  0x00000001
113 formats: XR24 AR24 XB24 AB24 RG24 BG24 RG16 BG16 NV12 NV16 NV24 NA12 NA16 NA24
YUYV
114 props:
115 8 type:
116 flags: immutable enum
117 enums: Overlay=0 Primary=1 Cursor=2
118 value: 0
119 137 NAME:
120 flags: immutable bitmask
```

株式会社日昇テクノロジー

```
121 values: Cluster1-win0=0x40

122 value: 64

123 (省略.........)
```

- id が 151 の TMDS は HDMI に、

- id が 162 の DSI は MIPI スクリーンに該当します。

CRTC、Planes、Connectors がリストアップされており、省略された部分には一部のバインド情報が含まれています。ソースコードを分析することで詳細を確認できます。

第 30 章 DRM アプリケーションプログラミング - レガシーインターフェース

このインターフェースは旧式の DRM プログラミングのアプリケーション開発インターフェースですが、このインターフェースを理解することで DRM についての理解が深まります。そのため、まず旧式インターフェースをガイドとして、DRM プログラミングの具体的なフローや実験について説明し、次の章で主流のインターフェースであるアトミックインターフェースについて説明します。

このセクションでは、4 つの実験を通じて DRM アプリケーションプログラミングに入門します。

この章のサンプルコードのディレクトリは: base_linux/screen/drm/drm-legacy

30.1 最も簡単な DRM (drm-single)

以下に、最も簡単な DRM アプリケーションを作成する方法を擬似コードで簡単に紹介します。

株式会社日昇テクノロジー

擬似コード：

```
int main(int argc, char **argv)
{
  // DRM デバイスを開く
  open("/dev/dri/card0");

  // DRM の情報を取得する
  drmModeGetResources(...);

  // 表示モードのコネクタ情報を取得する
  drmModeGetConnector(...);

  // ダムバッファを作成する
  drmIoctl(DRM_IOCTL_MODE_CREATE_DUMB);

  // バッファを FB オブジェクトにバインドする
  drmModeAddFB(...);

  // バッファをユーザースペースにマップし、バッファのポインタを取得する
  drmIoctl(DRM_IOCTL_MODE_MAP_DUMB);
  mmap(...);
```

株式会社日昇テクノロジー

```
    // バッファを特定のディスプレイコネクタ、表示モードなどと関連付ける
    drmModeSetCrtc(crtc_id, fb_id, connector_id, mode);
}
```

詳細な参考コードは以下の通りです：

表 1: base_linux/screen/drm/drm-legacy/drm-single.c

```
#define _GNU_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <time.h>
#include <unistd.h>
#include <xf86drm.h>
#include <xf86drmMode.h>

struct drm_device {
    uint32_t width; // ディスプレイの幅のピクセル数
    uint32_t height; // ディスプレイの高さのピクセル数
    uint32_t pitch; // 各行が占めるバイト数
```

株式会社日昇テクノロジー

```
    uint32_t handle; // drm_mode_create_dumb の返すハンドル
    uint32_t size; // ディスプレイが占める総バイト数
    uint32_t *vaddr; // mmap の先頭アドレス
    uint32_t fb_id; // 作成されたフレームバッファの ID
    struct drm_mode_create_dumb create; // 作成されたダム
    struct drm_mode_map_dumb map; // メモリマップの構造体
};

drmModeConnector *conn; // コネクタ関連の構造体
drmModeRes *res; // リソース
uint32_t conn_id; // コネクタの ID
uint32_t crtc_id; // CRTC の ID
int fd; // ファイルディスクリプタ

#define RED 0XFF0000
#define GREEN 0X00FF00
#define BLUE 0X0000FF

struct drm_device buf;

static int drm_create_fb(struct drm_device *bo)
{
    /* ダムバッファを作成、ピクセルフォーマットは XRGB888 */
```

株式会社日昇テクノロジー

```
bo->create.width = bo->width;

bo->create.height = bo->height;

bo->create.bpp = 32;


/* ハンドル、ピッチ、サイズが返される */

drmIoctl(fd, DRM_IOCTL_MODE_CREATE_DUMB, &bo->create);


/* ダムバッファを FB オブジェクトにバインドする */

bo->pitch = bo->create.pitch;

bo->size = bo->create.size;

bo->handle = bo->create.handle;

drmModeAddFB(fd, bo->width, bo->height, 24, 32, bo->pitch, bo->handle, &bo->fb_id);


// 各行が占めるバイト数、総バイト数、MAP_DUMB のハンドルを表示

printf("pitch = %d, size = %d, handle = %d¥n", bo->pitch, bo->size, bo->handle);


/* ダムバッファをユーザースペースにマップする */

bo->map.handle = bo->create.handle;

drmIoctl(fd, DRM_IOCTL_MODE_MAP_DUMB, &bo->map);


bo->vaddr = mmap(0, bo->create.size, PROT_READ | PROT_WRITE, MAP_SHARED,
fd, bo->map.offset);
```

株式会社日昇テクノロジー

```
    /* ダムバッファを白色で初期化する */
    memset(bo->vaddr, 0xff, bo->size);

    return 0;
}

static void drm_destroy_fb(struct drm_device *bo)
{
    struct drm_mode_destroy_dumb destroy = {};
    drmModeRmFB(fd, bo->fb_id);
    munmap(bo->vaddr, bo->size);
    destroy.handle = bo->handle;
    drmIoctl(fd, DRM_IOCTL_MODE_DESTROY_DUMB, &destroy);
}

int drm_init()
{
    // DRM デバイスを開く。デバイスはデバイスツリーの変更に応じて変更される。複数のデバイスがある場合は各スクリーンデバイスに対応する DRM デバイスを確認すること。
    fd = open("/dev/dri/card0", O_RDWR | O_CLOEXEC);
    if (fd < 0) {
        printf("wrong¥n");
        return 0;
```

株式会社日昇テクノロジー

```
}


// DRM の情報を取得する
res = drmModeGetResources(fd);
crtc_id = res->crtcs[0];
conn_id = res->connectors[0];


// CRTC およびコネクタの ID を表示する
printf("crtc = %d, connector = %d¥n", crtc_id, conn_id);


conn = drmModeGetConnector(fd, conn_id);
buf.width = conn->modes[0].hdisplay;
buf.height = conn->modes[0].vdisplay;


// ディスプレイの解像度を表示する
printf("width = %d, height = %d¥n", buf.width, buf.height);


// フレームバッファを作成する
drm_create_fb(&buf);


// CRTC を設定する
drmModeSetCrtc(fd, crtc_id, buf.fb_id, 0, 0, &conn_id, 1, &conn->modes[0]);
```

株式会社日昇テクノロジー

```
    return 0;
}


void drm_exit()
{
    drm_destroy_fb(&buf);
    drmModeFreeConnector(conn);
    drmModeFreeResources(res);
    close(fd);
}


int main(int argc, char **argv)
{
    int i;
    drm_init();
    sleep(2);


    // 画面をクリアして色を設定する
    for (i = 0; i < buf.width * buf.height; i++)
        buf.vaddr[i] = 0x123456;


    sleep(2);
    drm_exit();
```

株式会社日昇テクノロジー

```
    exit(0);
}
```

main 関数では、以下の操作を実行します：

- 125 行目で、drm_init 関数を呼び出して初期化します。
- 128～129 行目で、ループを使用してフレームバッファのすべてのピクセルを 0x123456 に設定し、画面の色を変更します。
- 132 行目で、drm_exit 関数を呼び出してクリーンアップおよび終了します。

drm_init 関数では、以下の操作を実行します：

- 84 行目で、DRM デバイスファイル（/dev/dri/card0）を開きます。
- 91 行目で、drmModeGetResources 関数を呼び出して DRM デバイスの情報を取得し、CRTC（Cathode Ray Tube Controller）およびコネクタ（Connector）の ID を取得します。
- 97 行目で、drmModeGetConnector 関数を呼び出してコネクタの詳細情報を取得し、画面の解像度を取得します。
- 105 行目で、drm_create_fb 関数を呼び出してフレームバッファを作成します。
- 108 行目で、drmModeSetCrtc 関数を呼び出してフレームバッファを CRTC に関連付けます。

drm_create_fb 関数では、以下の操作を実行します：

- 41～44 行目で、フレームバッファの幅、高さ、およびピクセルフォーマットを設定します。

株式会社日昇テクノロジー

- 47 行目で、drmIoctl 関数を呼び出し、DRM_IOCTL_MODE_CREATE_DUMB コマンドを渡してダムバッファを作成します。この関数は、バッファのハンドル、オフセット、行のバイト数などの情報を返します。

- 53 行目で、`drmModeAddFB`関数を使用してダムバッファをフレームバッファオブジェクト（FBO）にバインドし、フレームバッファの ID を取得します。

- 61 行目で、`drmIoctl`関数を呼び出し、`DRM_IOCTL_MODE_MAP_DUMB`コマンドを渡してダムバッファをユーザースペースにマップします。`mmap`関数を使用してバッファのポインタを取得します。

- 67 行目で、`memset`関数を使用してフレームバッファを白色で初期化します。

30.1.1 詳細コード分析

全体的に、4 つのステップで最小の DRM ディスプレイプログラムを初期化できます：

1. デバイスを開く。
2. crtc_id、connector_id、およびそれらの構造体情報を取得する。
3. フレームバッファを作成する。
4. CRTC を設定する。

リスト 2: 初期化

```
int drm_init()
{
    // DRM デバイスを開く。デバイスはデバイスツリーの変更に応じて変わるため、複数のデバイスがある場合は各スクリーンデバイスに対応する DRM デバイスを確認すること。
    fd = open("/dev/dri/card0", O_RDWR | O_CLOEXEC);
```

株式会社日昇テクノロジー

```
if (fd < 0) {
    printf("wrong¥n");
    return 0;
}

// DRM の情報を取得する
res = drmModeGetResources(fd);
crtc_id = res->crtcs[0];
conn_id = res->connectors[0];
// CRTCS とコネクタの ID を表示する
printf("crtc = %d, connector = %d¥n", crtc_id, conn_id);

conn = drmModeGetConnector(fd, conn_id);
buf.width = conn->modes[0].hdisplay;
buf.height = conn->modes[0].vdisplay;

// 画面の解像度を表示する
printf("width = %d, height = %d¥n", buf.width, buf.height);

// フレームバッファを作成する
drm_create_fb(&buf);

// CRTCS を設定する
```

株式会社日昇テクノロジー

```
    drmModeSetCrtc(fd, crtc_id, buf.fb_id, 0, 0, &conn_id, 1, &conn->modes[0]);


    return 0;
}
```

- 6～11 行目: デバイスを開く。DRM デバイスは/dev/dri/cardx にあり、他のプロセスに占有されている場合はエラーが発生します。専用の DRM デバイスを開く関数も使用できます。ここでは Rockchip のチップを使用しており、公式のドライバを使用しているため、"rockchip"を使用します。

```
fd = drmOpen("rockchip", NULL);
if (fd < 0) {
    printf("failed to open rockchip drm¥n");
    return fd;
}

// これは modetest プログラム内のデバイスマッチングテーブルです。
static const char * const modules[] = {
    "i915", "amdgpu", "radeon", "nouveau",
    "vmwgfx", "omapdrm", "exynos", "tilcdc",
    "msm", "sti", "tegra", "imx-drm",
    "rockchip", "atmel-hlcdc", "fsl-dcu-drm",
    "vc4", "virtio_gpu", "mediatek", "meson",
    "pl111", "stm", "sun4i-drm",
```

株式会社日昇テクノロジー

};

- 10～15 行目: DRM のリソースを取得し、CRTC とコネクタの ID を取得して表示します。

res は drmModeRes 構造体へのポインタで、drmModeRes 構造体のプロトタイプは以下の通りです。

```
typedef struct _drmModeRes {
    int count_fbs; // フレームバッファの数
    uint32_t *fbs;

    int count_crtcs; // CRTCs の数
    uint32_t *crtcs;

    int count_connectors; // コネクタの数
    uint32_t *connectors;

    int count_encoders; // エンコーダの数
    uint32_t *encoders;

    uint32_t min_width, max_width; // 最小幅と最大幅
    uint32_t min_height, max_height; // 最小高さと最大高さ
} drmModeRes, *drmModeResPtr;
```

- 17～22 行目: コネクタ ID を使用してコネクタのリソースを取得し、画面の幅と高さを

株式会社日昇テクノロジー

構造体に記録して表示します。

conn は drmModeConnector 構造体へのポインタで、drmModeConnector 構造体のプロトタイプは以下の通りです。

```
typedef struct _drmModeConnector {
    uint32_t connector_id; // 自身の ID
    uint32_t encoder_id; // 接続されるエンコーダ ID
    uint32_t connector_type;
    uint32_t connector_type_id;
    drmModeConnection connection; // コネクタの接続情報の列挙
    uint32_t mmWidth, mmHeight; // ミリメートル単位の幅と高さ
    drmModeSubPixel subpixel; // サブピクセルの列挙

    int count_modes; // モードの数
    drmModeModeInfoPtr modes; // 解像度、タイミング、クロックなどの情報を保持するポインタ

    int count_props; // アトミックモードで使用される
    uint32_t *props; // プロパティ ID のリスト
    uint64_t *prop_values; // プロパティ値のリスト

    int count_encoders; // エンコーダの数
    uint32_t *encoders; // エンコーダ ID のリスト
```

株式会社日昇テクノロジー

} drmModeConnector, *drmModeConnectorPtr;

- 24～26 行目: フレームバッファを作成し、ユーザーメモリに mmap します。詳細は後述します。

- 27～30 行目: CRTC を設定します。このステップで、画面に表示されるものが見えるようになります。

フレームバッファと CRTC の関係図:

株式会社日昇テクノロジー

drmModeSetCrtc 関数のプロトタイプ:

```
int drmModeSetCrtc(int fd, uint32_t crtcId, uint32_t bufferId,
                uint32_t x, uint32_t y, uint32_t *connectors, int count,
                drmModeModeInfoPtr mode);
```

1. fd: ファイルディスクリプタ。

2. crtcId: 設定する crtc-id。

3. bufferId: 設定するフレームバッファ ID。

4. x: x 軸のオフセット。オフセットを設定するとフレームバッファの他の領域が表示されます。

5. y: y 軸のオフセット。オフセットを設定するとフレームバッファの他の領域が表示されます。

6. connectors: 接続するコネクタ ID。

7. count: コネクタの数。

8. mode: 使用するモード。

初期化が完了したら、フレームバッファを操作して画面を制御できるようになります。

フレームバッファ作成部分の説明:

リスト 3: フレームバッファの作成

```
struct drm_device {
    struct drm_mode_create_dumb create; // ダム作成
    struct drm_mode_map_dumb map; // メモリマップ構造体
};

static int drm_create_fb(struct drm_device *bo)
{
    /* ダムバッファを作成。ピクセルフォーマットは XRGB888 */
    bo->create.width = bo->width;
```

株式会社日昇テクノロジー

```
bo->create.height = bo->height;
bo->create.bpp = 32;

/* ハンドル、ピッチ、サイズが返される */
drmIoctl(fd, DRM_IOCTL_MODE_CREATE_DUMB, &bo->create);

/* ダムバッファを FB オブジェクトにバインド */
bo->pitch = bo->create.pitch;
bo->size = bo->create.size;
bo->handle = bo->create.handle;
drmModeAddFB(fd, bo->width, bo->height, 24, 32, bo->pitch, bo->handle, &bo->fb_id);

// 各行が占めるバイト数、総バイト数、MAP_DUMB のハンドルを表示
printf("pitch = %d, size = %d, handle = %d¥n", bo->pitch, bo->size, bo->handle);

/* ダムバッファをユーザースペースにマップ */
bo->map.handle = bo->create.handle;
drmIoctl(fd, DRM_IOCTL_MODE_MAP_DUMB, &bo->map);

bo->vaddr = mmap(0, bo->create.size, PROT_READ | PROT_WRITE, MAP_SHARED,
fd, bo->map.offset);

/* ダムバッファを白色で初期化 */
```

株式会社日昇テクノロジー

```
    memset(bo->vaddr, 0xff, bo->size);
```

```
    return 0;
}
```

- 3 行目: drm_mode_create_dumb 構造体はフレームバッファの属性を構築するために使用されます。

```
struct drm_mode_create_dumb {
    __u32 height; // 高さのピクセル数
    __u32 width; // 幅のピクセル数
    __u32 bpp; // 各ピクセルのビット数
    __u32 flags;
    /* これらのパラメータは作成後に返される */
    __u

32 handle; // mmap に使用するハンドル
    __u32 pitch; // 各行のバイト数
    __u64 size; // 総バイト数
};
```

- 4 行目: drm_mode_map_dumb は mmap メモリ領域を構築するために使用されます。

```
struct drm_mode_map_dumb {
    __u32 handle; // マップするオブジェクトのハンドル
    __u32 pad;
    __u64 offset; // mmap 呼び出しに使用するフェイクオフセット
```

株式会社日昇テクノロジー

};

- 10～14 行目: フレームバッファのサイズ属性を構築します。この実験では、フレームバッファのサイズを表示領域と同じに設定していますが、数値を変更してより大きなフレームバッファ属性を構築することもできます。

- 15～16 行目: 作成した属性を送信します。成功すると、drm_mode_create_dumb の他の属性が返されます。

- 18～21 行目: 返された属性値をグローバル変数 drm_device に格納します。

- 22～23 行目: フレームバッファを作成します。

```
/* スキャンアウトバッファとしてバッファオブジェクトを使用して新しいフレームバッファを作成 */
extern int drmModeAddFB(int fd, uint32_t width, uint32_t height, uint8_t depth, uint8_t bpp, uint32_t pitch, uint32_t bo_handle, uint32_t *buf_id);
/* 特定のピクセルフォーマットで作成 */
extern int drmModeAddFB2(int fd, uint32_t width, uint32_t height, uint32_t pixel_format, const uint32_t bo_handles[4], const uint32_t pitches[4], const uint32_t offsets[4], uint32_t *buf_id, uint32_t flags);
extern drmModeFBPtr drmModeGetFB(int fd, uint32_t bufferId);
```

ここでは主に drmModeAddFB 関数について説明します。他の関数についてはソースコードを読んで理解してください。

drmModeAddFB 関数の分析：

1. fd: ファイルディスクリプタ。

2. width: フレームバッファの幅のピクセル数。

3. height: フレームバッファの高さのピクセル数。

株式会社日昇テクノロジー

4. depth: フレームバッファの各ピクセルの実際のビット数（RGB888 の場合は 3 バイト）。

5. bpp: 各ピクセルが占めるビット数（XRGB8888 の場合は 4 バイト）。

6. pitch: 各行が占めるバイト数（720x4）。

7. bo_handle: 上記で作成したダムのハンドル。

8. buf_id: フレームバッファの ID。ポインタ形式で渡され、成功すると値が返されます。

drmModeAddFB2 関数は、特殊なフォーマット（例えば YUV や C8 フォーマット）のフレームバッファを作成します。

drmModeGetFB 関数は、フレームバッファのリソースを取得します。

```
typedef struct _drmModeFB {
    uint32_t fb_id;
    uint32_t width, height;
    uint32_t pitch;
    uint32_t bpp;
    uint32_t depth;
    /* ドライバ固有のハンドル */
    uint32_t handle;
} drmModeFB, *drmModeFBPtr;
```

- 28～34 行目: mmap を設定し、フレームバッファのメモリ領域をユーザースペースにマップします。

- 36 行目: フレームバッファ領域全体を白色に設定します。

これらの手順により、フレームバッファが作成され、ユーザースペースにマップされ、ユーザーが使用できるようになります。

株式会社日昇テクノロジー

1. フレームバッファ領域の属性を構築し、ダムを作成します。

2. 属性に基づいてフレームバッファを作成します。

3. ハンドルを使用して、フレームバッファをユーザースペースにマップします。

リスト 4: DRM の解除

```
void drm_exit()
{
    drm_destroy_fb(&buf);
    drmModeFreeConnector(conn);
    drmModeFreeResources(res);
    close(fd);
}


static void drm_destroy_fb(struct drm_device *bo)
{
    struct drm_mode_destroy_dumb destroy = {};
    drmModeRmFB(fd, bo->fb_id);
    munmap(bo->vaddr, bo->size);
    destroy.handle = bo->handle;
    drmIoctl(fd, DRM_IOCTL_MODE_DESTROY_DUMB, &destroy);
}
```

DRM の解除関数は簡単で、以下の手順を実行するだけです。

1. フレームバッファを削除し、メモリマップを解除し、ダムを破棄します。

株式会社日昇テクノロジー

2. コネクタリソースを解放します。

3. リソースを解放します。

4. ファイルディスクリプタを閉じます。

30.1.2 コンパイル

lubancat_rk_code_storage/base_linux/screen/drm/drm-legacy/drm-single ディレクトリに移動します。

方法 1：

```
# コンパイル
make
```

方法 2：

```
# コンパイル
gcc -o drm-single drm-single.c `pkg-config --cflags libdrm` `pkg-config --libs libdrm`
```

30.1.3 実行

```
# 実行
./drm-single
```

現象：

スクリーンが最初に白くなり、その後青くなります。プログラムが終了すると、スクリーンは黒くなります。また、端末にはプログラムが出力したスクリーンの CRTC の ID、コネクタの ID、解像度、各行のバイト数、および総バイト数が表示されます。

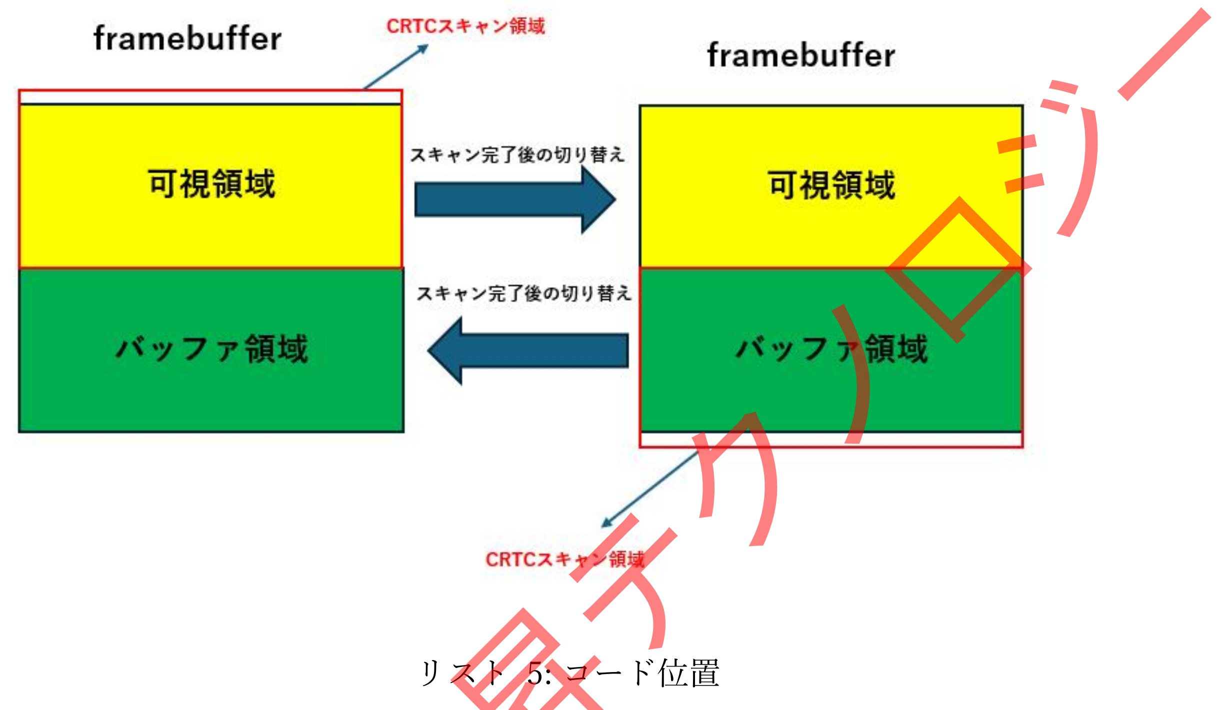
30.2 ダブルバッファ DRM（drm-double.c）

ダブルバッファの原理は、CRTC のスキャンメモリの位置を変更することです。

株式会社日昇テクノロジー

30.2.1 単一フレームバッファ ダブルバッファ

- 表示領域の 2 倍のフレームバッファを作成し、オフセットを変更してフレームを切り替えます。一つのフレームをスキャンした後、別のフレームに切り替えます。



リスト 5: コード位置

```
base_linux/screen/drm/drm-legacy/drm-double-one-fb/drm-double-one-fb.c
```

30.2.2 コンパイル

方法 1：

```
# コンパイル
make
```

方法 2：

```
# コンパイル
gcc -o drm-double-one-fb drm-double-one-fb.c `pkg-config --cflags libdrm` `pkg-config --libs libdrm`
```

株式会社日昇テクノロジー

30.2.3 実行

```
./drm-double-one-fb
```

現象：

1. スクリーンが赤くなる

2. キーを押す

3. スクリーンが青くなる

4. キーを押す

5. スクリーンが赤くなる

6. キーを押す

7. プログラムが終了

最小プログラムとほぼ同じ操作ですが、ここでは違いについてのみ説明します。

リスト 6: フレームバッファの作成

```
static int drm_create_fb(struct drm_device *bo)
{
    /* ダムバッファを作成。ピクセルフォーマットは XRGB888 */
    bo->create.width = bo->width;
    bo->create.height = bo->height * 2;
    bo->create.bpp = 32;

    /* ハンドル、ピッチ、サイズが返される */
    drmIoctl(fd, DRM_IOCTL_MODE_CREATE_DUMB, &bo->create);
```

株式会社日昇テクノロジー

```
/* ダムバッファを FB オブジェクトにバインド */
bo->pitch = bo->create.pitch;
bo->size = bo->create.size;
bo->handle = bo->create.handle;
drmModeAddFB(fd, bo->width, bo->height * 2, 24, 32, bo->pitch, bo->handle, &bo->fb_id);

// 各行のバイト数、総バイト数、MAP_DUMB のハンドルを表示
printf("pitch = %d, size = %d, handle = %d¥n", bo->pitch, bo->size, bo->handle);

/* ダムバッファをユーザースペースにマップ */
bo->map.handle = bo->create.handle;
drmIoctl(fd, DRM_IOCTL_MODE_MAP_DUMB, &bo->map);

bo->vaddr = mmap(0, bo->create.size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, bo->map.offset);

/* ダムバッファを白色で初期化 */
memset(bo->vaddr, 0x00, bo->size);

return 0;
}
```

異なる点は：

株式会社日昇テクノロジー

- 5 行目でダムの高さを 2 倍にしています。

- 15 行目でフレームバッファを作成する際に高さを 2 倍にしています。

- これにより、フレームバッファのサイズが前回の実験の 2 倍になります。

リスト 7: main 関数

```
int main(int argc, char **argv)
{
    int i;
    int size;
    drm_init();
    size = buf.width * buf.height;
    // 上層バッファを赤で満たす
    for (i = 0; i < size; i++)
        buf.vaddr[i] = RED;
    // 下層バッファを青で満たす
    for (i = size; i < size * 2; i++)
        buf.vaddr[i] = BLUE;

    // 文字入力を待つ
    getchar();
    // 下層バッファに切り替え
    drmModeSetCrtc(fd, crtc_id, buf.fb_id, 0, 1080, &conn_id, 1, &conn->modes[0]);
    // 注意：1080 のパラメータは実際のスクリーンの高さに合わせて変更してください。例
```

株式会社日昇テクノロジー

えば、1024x600 のスクリーンの場合、1080 を 600 に変更します。

```
    // 文字入力を待つ
    getchar();
    // 上層バッファに切り替え
    drmModeSetCrtc(fd, crtc_id, buf.fb_id, 0, 0, &conn_id, 1, &conn->modes[0]);

    // 文字入力を待つ
    getchar();

    drm_exit();

    return 0;
}
```

- 切り替え操作は drmModeSetCrtc()関数を使用します。この関数はフレームバッファを下にオフセットすることで表示内容を変更します。

30.2.4 複数フレームバッファ ダブルバッファ

- 2 つのフレームバッファを作成し、フレームバッファを切り替えることでダブルバッファの表示を行います。

リスト 8: コード位置

```
base_linux/screen/drm/drm-legacy/drm-double-muti-fb/drm-double-muti-fb.c
```

株式会社日昇テクノロジー

30.2.5 コンパイル

方法 1：

```
# コンパイル

make
```

方法 2：

```
# コンパイル

gcc -o drm-double-muti-fb drm-double-muti-fb.c `pkg-config --cflags libdrm` `pkg-config --libs libdrm`
```

30.2.6 実行

```
./drm-double-muti-fb
```

現象：

1. スクリーンが赤くなる

2. キーを押す

3. スクリーンが青くなる

4. キーを押す

5. スクリーンが赤くなる

6. キーを押す

7. プログラムが終了

最小プログラムとほぼ同じ操作ですが、ここでは違いについてのみ説明します。

リスト 9: drm 初期化

```
struct drm_device buf[2];
```

株式会社日昇テクノロジー

int drm_init()

{

// DRM デバイスを開く。デバイスはデバイスツリーの変更に応じて変わるため、複数のデバイスがある場合は各スクリーンデバイスに対応する DRM デバイスを確認すること。

```
fd = open("/dev/dri/card0", O_RDWR | O_CLOEXEC);
if (fd < 0) {
    printf("wrong¥n");
    return 0;
}

// DRM の情報を取得する
res = drmModeGetResources(fd);
crtc_id = res->crtcs[0];
conn_id = res->connectors[0];
// CRTCS とコネクタの ID を表示する
printf("crtc = %d, connector = %d¥n", crtc_id, conn_id);

conn = drmModeGetConnector(fd, conn_id);
buf[0].width = conn->modes[0].hdisplay;
buf[0].height = conn->modes[0].vdisplay;

buf[1].width = conn->modes[0].hdisplay;
```

株式会社日昇テクノロジー

```
    buf[1].height = conn->modes[0].vdisplay;


    // 画面の解像度を表示する
    printf("width = %d, height = %d¥n", buf[0].width, buf[0].height);
```

```
    // フレームバッファを作成する
    drm_create_fb(&buf[0]);
    drm_create_fb(&buf[1]);


    // CRTCS を設定する
    drmModeSetCrtc(fd, crtc_id, buf[0].fb_id, 0, 0, &conn_id, 1, &conn->modes[0

]);


    return 0;
}
```

- 1 行目: 2 つの構造体を定義し、それぞれにフレームバッファ情報を格納します。
- 20～24 行目: 解像度をそれぞれ取得します。
- 29～31 行目: フレームバッファを設定し、ユーザースペースにメモリをマップします。
- 33～36 行目: スクリーンを buf[0]のデータに設定します。

リスト 10: main 関数

```
int main(int argc, char **argv)
{
```

株式会社日昇テクノロジー

```
int i;

int size0, size1;

drm_init();

size0 = buf[0].width * buf[0].height;

size1 = buf[1].width * buf[1].height;

// 上層バッファを赤で満たす

for (i = 0; i < size0; i++)

   buf[0].vaddr[i] = RED;

// 下層バッファを青で満たす

for (i = 0; i < size1; i++)

   buf[1].vaddr[i] = BLUE;


drmModeSetCrtc(fd, crtc_id, buf[0].fb_id, 0, 0, &conn_id, 1, &conn->modes[0]);


// 文字入力を待つ

getchar();

// 下層バッファに切り替え

drmModeSetCrtc(fd, crtc_id, buf[1].fb_id, 0, 0, &conn_id, 1, &conn->modes[0]);


// 文字入力を待つ

getchar();

// 上層バッファに切り替え

drmModeSetCrtc(fd, crtc_id, buf[0].fb_id, 0, 0, &conn_id, 1, &conn->modes[0]);
```

株式会社日昇テクノロジー

```
    // 文字入力を待つ
    getchar();

    drm_exit();

    return 0;
}
```

単一フレームバッファと比較すると、ほぼ同じ形式ですが、最大の違いは渡す fb_id とオフセットがないことです。操作する fb_id を指定するだけでスクリーンを操作できます。

30.2.7 まとめ

この 2 つの方法にはそれぞれの利点があります。単一フレームバッファは簡単で、記述が早く、単純な画像表示に適しています。一方、複数フレームバッファは複雑ですが、拡張性が高く、複雑な画像表示に適しています。

30.3 ページフリップ

drmModePageFlip()は drmModeSetCrtc()と同じように表示内容を更新するために使用されますが、最大の違いは、drmModePageFlip()は VSYNC が到達するまでフレームバッファの切り替えを実行しない点です。一方、drmModeSetCrtc()は即座にフレームバッファの切り替えを実行します。これは、一部のハードウェアではティアリング（tear effect）問題を引き起こしやすいですが、drmModePageFlip()ではこの問題は発生しません。

drmModePageFlip()は VSYNC イベント機構に基づいているため、低レベルの DRM ドライバは VBLANK イベントをサポートする必要があります。

株式会社日昇テクノロジー

Update buffer
Pixel pointer
Tearing effect
Update without page flipping
Pixel pointer
Flip buffers
Update with page flipping
Frame N
Frame N
Frame N + 1

30.3.1 ページフリップの実験

リスト 11: コード位置

```
base_linux/screen/drm/drm-legacy/drm-page-flip/drm-page-flip.c
```

30.3.2 コンパイル

方法 1：

```
# コンパイル

make
```

方法 2：

```
# コンパイル

gcc -o drm-page-flip drm-page-flip.c `pkg-config --cflags libdrm` `pkg-config --libs libdrm`
```

株式会社日昇テクノロジー

30.3.3 実行

```
./drm-page-flip
```

現象：

1. スクリーンが赤くなる

2. キーを押す

3. スクリーンが青くなる

4. キーを押す

5. スクリーンが赤くなる

6. キーを押す

7. プログラムが終了

30.3.4 プログラム分析

リスト 12: main 関数

```
static void drm_page_flip_handler(int fd, uint32_t frame, uint32_t sec, uint32_t usec, void
*data)
{
    static int i = 0;
    uint32_t crtc_id = *(uint32_t *)data;

    if (i == 0)
        i = 1;
    else
        i = 0;
```

株式会社日昇テクノロジー

```
    drmModePageFlip(fd, crtc_id, buf[i].fb_id, DRM_MODE_PAGE_FLIP_EVENT, data);
}


int main(int argc, char **argv)
{
    int i;
    int size0, size1;
    ev.version = DRM_EVENT_CONTEXT_VERSION;
    ev.page_flip_handler = drm_page_flip_handler;


    drm_init();
    size0 = buf[0].width * buf[0].height;
    size1 = buf[1].width * buf[1].height;
    // buffer1 を赤で満たす
    for (i = 0; i < size0; i++)
        buf[0].vaddr[i] = RED;
    // buffer2 を青で満たす
    for (i = 0; i < size1; i++)
        buf[1].vaddr[i] = BLUE;


    drmModePageFlip(fd, crtc_id, buf[0].fb_id, DRM_MODE_PAGE_FLIP_EVENT,
&crtc_id);
```

株式会社日昇テクノロジー

```
    // 文字入力を待つ
    getchar();
    // buffer2 に切り替え
    drmHandleEvent(fd, &ev);
    // 文字入力を待つ
    getchar();
    // buffer1 に切り替え
    drmHandleEvent(fd, &ev);
    // 文字入力を待つ
    getchar();

    drm_exit();

    exit 0;
}
```

この実験は複数フレームバッファのダブルバッファの構造に似ていますが、違いを説明します：

- 最初に drmEventContext を定義し、ev.version と ev.page_flip_handler を設定する必要があります。

- drmHandleEvent(fd, &ev);が実行されると、drm_page_flip_handler()関数がトリガーされ、ページフリップが行われます。

drmModePageFlip 関数のプロトタイプ：

```
int drmModePageFlip (int fd, uint32_t crtc_id, uint32_t fb_id, uint32_t flags, void *user_data)
```

株式会社日昇テクノロジー

1. fd: 開かれた DRM デバイスのファイルディスクリプタ。

2. crtc_id: CRTC が変更するフレームバッファの CRTC ID。

3. fb_id: 表示するフレームバッファの ID。

4. flags: 操作に影響するフラグ。サポートされる値は、

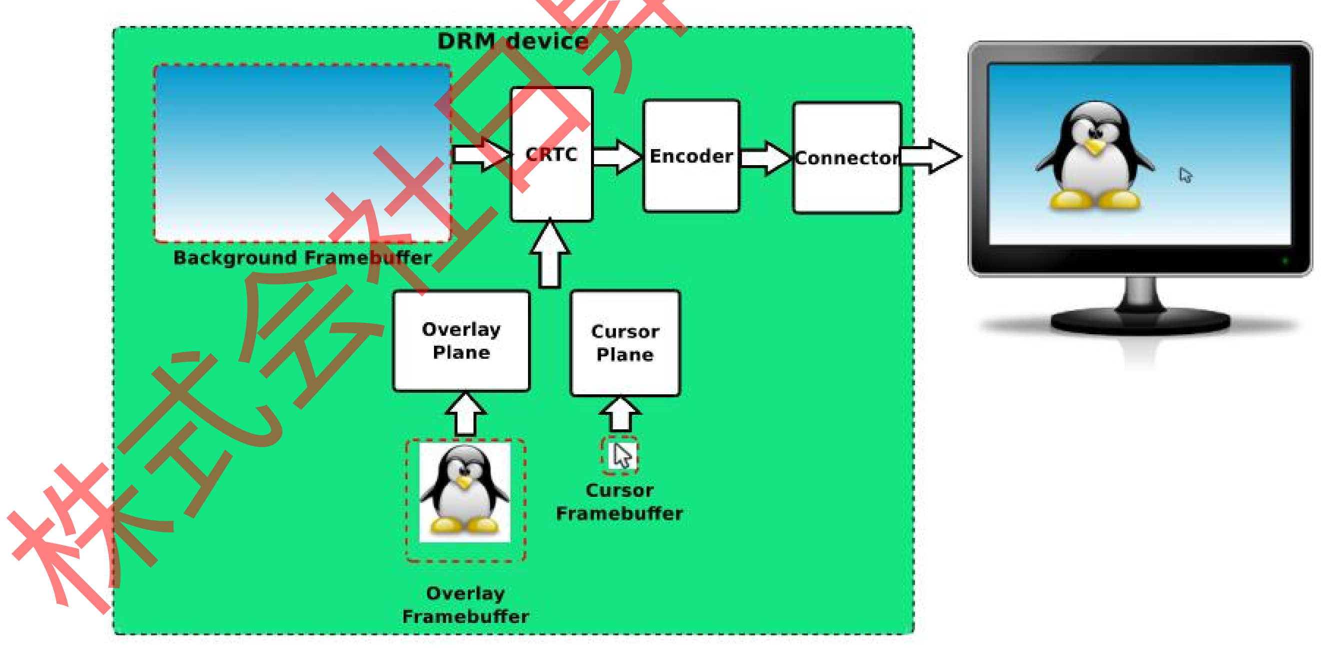
DRM_MODE_PAGE_FLIP_ASYNC（即座にフリップ）および

DRM_MODE_PAGE_FLIP_EVENT（ページフリップイベントを送信）です。

5. user_data: vblank イベントを要求する場合、ページフリップハンドラーが使用するデータ。

30.4 drm-planes

Planes には、複数の plane を重ねて、レイヤーを自由にクリップ、スケーリング、および合成する強力な特性があります。



株式会社日昇テクノロジー

30.4.1 drm-planes 実験

この実験は最も簡単な DRM（drm-single）を基に一部変更して作成しました。

リスト 13: コード位置

```
base_linux/screen/drm/drm-legacy/drm-planes/drm-planes.c
```

30.4.2 コンパイル

方法 1：

```
# コンパイル

make
```

方法 2：

```
# コンパイル

gcc -o drm-planes drm-planes.c `pkg-config --cflags libdrm` `pkg-config --libs libdrm`
```

30.4.3 実行

```
./drm-planes
```

実験現象：

- ./drm-planes を実行
- キーを押す
- スクリーンに赤、緑、青の 3 本の横線が表示される

株式会社日昇テクノロジー

株式会社日昇テクノロジー

30.4.4 プログラム分析

リスト 14: 新規初期化コード

```
drmModePlaneRes *plane_res; // Plane リソース
uint32_t plane_id[3]; // Plane ID 配列


drmSetClientCap(fd, DRM_CLIENT_CAP_UNIVERSAL_PLANES, 1);
plane_res = drmModeGetPlaneResources(fd);
printf("count_planes = %d¥n", plane_res->count_planes);
for (i = 0; i < 3; i++) {
   plane_id[i] = plane_res->planes[i];
   printf("planes[%d]= %d¥n", i, plane_id[i]);
}
```

drmModePlaneRes 構造体のプロトタイプ:

```
typedef struct _drmModePlaneRes {
   uint32_t count_planes; // planes の数
   uint32_t *planes; // planes-id 配列へのポインタ
} drmModePlaneRes, *drmModePlaneResPtr;
```

- drmSetClientCap(fd, DRM_CLIENT_CAP_UNIVERSAL_PLANES, 1);は、DRM のコアドライバがすべての planes をユーザースペースに公開するように設定します。これを設定すると planes を操作できるようになります。

- 5～10 行目: planes のリソースを取得し、すべての plane の ID を配列に格納して表示します。

株式会社日昇テクノロジー

リスト 15: main 関数

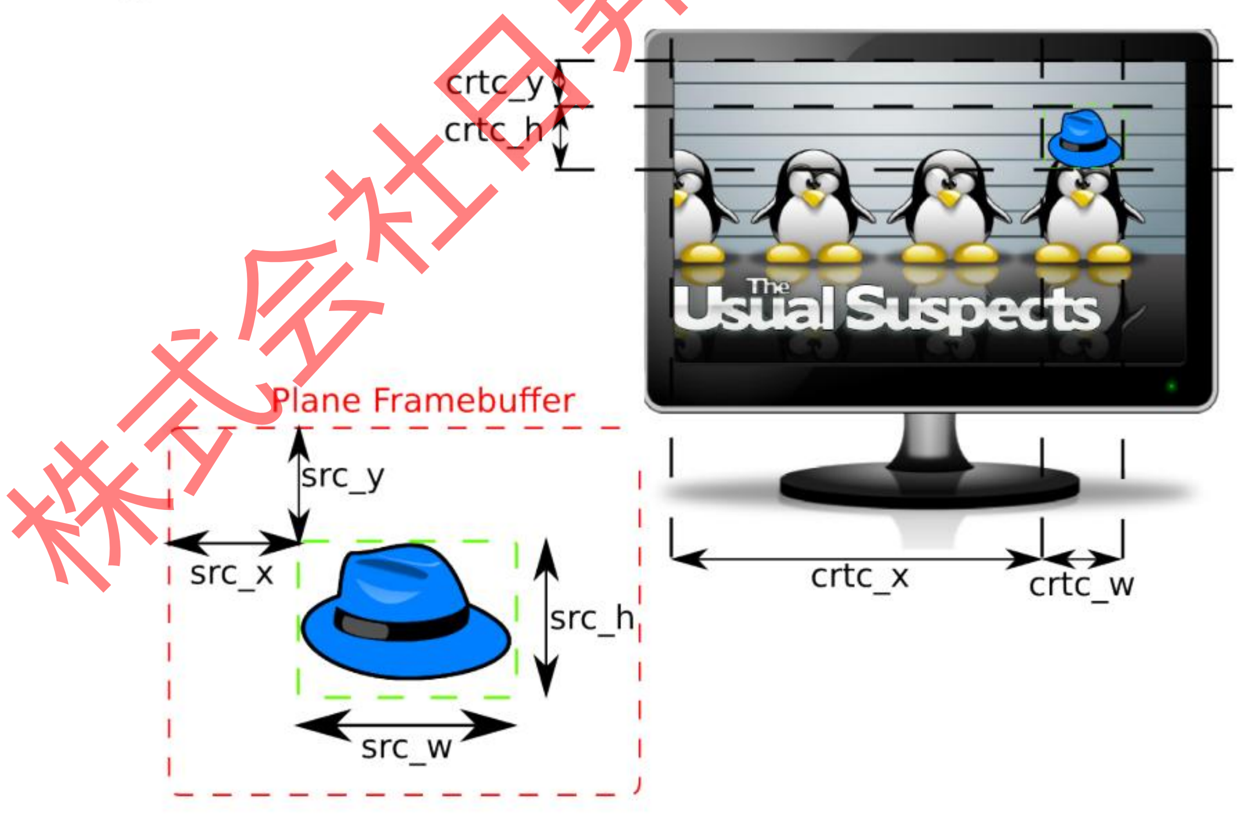
```
int main(int argc, char **argv)
{
    int i;
    int j = 0;
    drm_init();

    // 3 色を表示
    for (j = 0; j < 3; j++) {
        for (i = j * buf.width * buf.height / 3; i < (j + 1) * buf.width * buf.height / 3; i++)
            buf.vaddr[i] = color_table[j];
    }

    getchar();
    // フレームバッファの上 2/3 の領域を plane 1 に設定。スクリーンは変更され、フレームバッファの領域がスクリーン全体に拡大表示される。
    drmModeSetPlane(fd, plane_id[0], crtc_id, buf.fb_id, 0, 0, 0, buf.width, buf.height, 0 << 16, 0 << 16, buf.width << 16, buf.height / 3 * 2 << 16);

    getchar();
    // フレームバッファの領域を縮小して plane 2 に設定し、スクリーンの下半分に配置。plane 1 の一部を覆う。
    drmModeSetPlane(fd, plane_id[1], crtc_id, buf.fb_id, 0, buf.width / 4, buf.height / 2,
```

株式会社日昇テクノロジー

```
buf.width / 2, buf.height / 2, 0 << 16, 0 << 16, buf.width << 16, buf.height << 16);

    getchar();

    drm_exit();

    return 0;
}
```

- drmModeSetPlane() 関数のプロトタイプ:

```
extern int drmModeSetPlane(int fd, uint32_t plane_id, uint32_t crtc_id, uint32_t fb_id,
uint32_t flags, int32_t crtc_x, int32_t crtc_y, uint32_t crtc_w, uint32_t crtc_h, uint32_t src_x,
uint32_t src_y, uint32_t src_w, uint32_t src_h);
```



上図は、関数を使用して帽子を画像に重ねる方法を示しています。

株式会社日昇テクノロジー

ret = drmModeSetPlane(fd, plane_id, crtc_id, fb_id, flags, crtc_x, crtc_y, crtc_w, crtc_h, src_x << 16, src_y << 16, src_w << 16, src_h << 16);

上記の関数は、上図の移動を対応しています。

- 17～20 行目: フレームバッファの(0,0)から開始して上部 2/3 の領域を切り取り、スクリーン全体に拡大表示します。
- 22～26 行目: フレームバッファ全体を縮小して plane 2 の下半分の中央に配置し、plane 2 を表示します（plane 1 の一部を覆います）。
- SRC と CRTC の X/Y が異なる場合、平行移動の効果が得られます。
- SRC と CRTC の W/H が異なる場合、拡大縮小の効果が得られます。
- SRC と FrameBuffer の W/H が異なる場合、クリッピングの効果が得られます。

30.5 レガシーインターフェース関数

前述のレガシーインターフェースで使用した関数は以下の通りです：

1. drmModeSetCrtc() 2. drmModeSetPlane()

- これらの関数は、一部の機能が重複しており、両方の関数が同時に効果を発揮する場合があります。
- DRM ドライバがアクティブになると、フレームバッファは PRIMARY Planes にバインドされ、drmModeSetCrtc()を使用して直接フレームバッファを操作できます。
- drmModeSetPlane()は、Planes を操作するための正式な関数です。
- ダブルバッファ設計では、drmModeSetCrtc()を使用する代わりに drmModeSetPlane()を使用してダブルバッファを構築できます。

30.6 まとめ

株式会社日昇テクノロジー

DRM アプリケーションプログラミングのレガシーインターフェースのフレームワークは、最も簡単な DRM（drm-single）の実験に基づいており、他の操作も最も簡単な DRM の実験フレームワーク上で修正を加えています。手順は以下の通りです：

1. デバイスを開く（2 つの方法があります）。
2. page-flip-handle を作成する（page-flip で使用）。
3. plane リソースを取得する（planes で使用）。
4. DRM のリソースを取得し、CRTCs とコネクタの位置番号を含む。
5. コネクタの基本情報を取得し、高さと幅のピクセル数を取得する。
6. フレームバッファを作成し、mmap 領域を作成。ダブルバッファ DRM の場合、フレームバッファの追加設定が必要です。
7. CRTCs または page-flip を設定する。
8. plane リソースを設定する（planes で使用）。
9. スクリーンを操作する。

30.7 デュアルスクリーン異表示

rk3588 のデュアルスクリーン異表示およびマルチスクリーン異表示の際、Planes の設定は少し特殊です。

3 つのスクリーンをオンにした場合の割り当て：

1. VP0 -> HDMI 可插拔 -> 二
2. VP1 -> MIPI 主スクリーン -> 三
3. VP2 -> DP 转 VGA，可插拔 -> 一

デュアルスクリーン異表示をオンにした場合、ここでは HDMI と MIPI スクリーンを同時にオンにする例を示します：

- drminit()関数内の crtc_id = res->crtcs[0];を設定し、この関数で HDMI の CRTC を

株式会社日昇テクノロジー

取得します。

- drminit()関数内の crtc_id = res->crtcs[1];を設定し、この関数で MIPI スクリーンの CRTC を取得します。

planes の設定にはいくつかの手順が必要です：

```
// すべての planes リソースを取得
plane_res = drmModeGetPlaneResources(fd);
for (i = 0; i < plane_res->count_planes; i++) {
    plane_id[i] = plane_res->planes[i];
}
```

planes のバインド情報は関数を使用して取得する必要があります：

```
// plane_id を使用して drmModePlanePtr のリソースを取得
extern drmModePlanePtr drmModeGetPlane(int fd, uint32_t plane_id);


typedef struct _drmModePlane {
    uint32_t count_formats; // フォーマット数
    uint32_t *formats; // フォーマットポインタ
    uint32_t plane_id;


    uint32_t crtc_id; // バインドされる crtc_id
    uint32_t fb_id; // バインドされる crtc_id


    uint32_t crtc_x, crtc_y;
    uint32_t x, y;
```

株式会社日昇テクノロジー

```
    uint32_t possible_crtcs; // バインドされる可能性の


ある crtc_id
    uint32_t gamma_size;
} drmModePlane, *drmModePlanePtr;
```

- 本ボードでは、Planes は可変であり、PRIMARY_PLANE 以外は crtc_id から取得できます。

- その他の Planes は possible_crtcs から取得する必要があり、possible_crtcs = 1 -> HDMI, possible_crtcs = 1 -> MIPI です。

このため、Planes を操作する際には、どの CRTC がどの Planes に対応しているかを知っておく必要があります。ここで直接答えを提供します：

1. HDMI -> plane_id[0], plane_id[2], plane_id[5]

2. MIPI -> plane_id[1], plane_id[3], plane_id[4]

Planes の設定は、上記に基づいて行うことができます。

第 31 章 DRM アプリケーションプログラミング - アトミックインターフェース

現在、DRM（ダイレクト・レンダリング・マネージャー）では主に Atomic（アトミック）インターフェースの使用が推奨されています。以前のプログラムで使用されていた Legacy（レガシー）インターフェースは既に時代遅れとなっています。

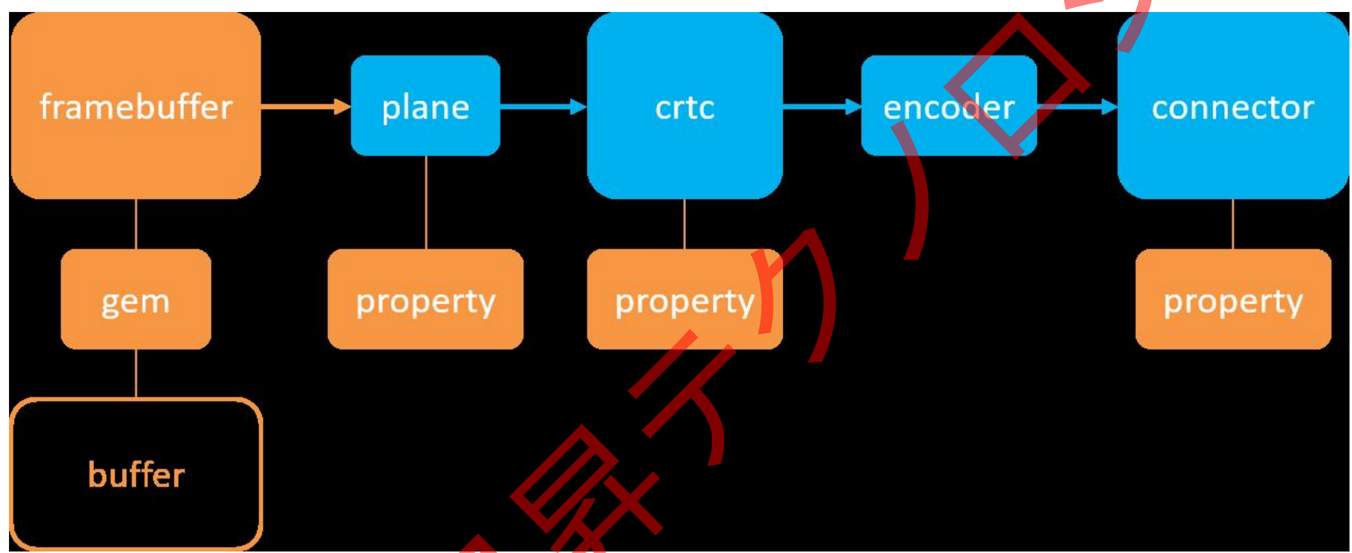
本章のサンプルコードディレクトリは以下の通りです：base_linux/screen/drm/drm-atomic/

株式会社日昇テクノロジー

31.1 Property（プロパティ）

Property（プロパティ）は、Atomic 操作に必須の基本要素です。

プロパティは、以前のレガシーインターフェースで渡されたパラメータを独立したグローバルプロパティとして抽出したものです。これらのプロパティパラメータを設定することで、表示パラメータの設定が完了します。



プロパティの構造は主に以下の 3 つの部分で構成されます：name、id、および value。このうち id は、DRM フレームワーク内でグローバルに一意の識別子です。

プロパティメカニズムの利点は以下の通りです：

1. 上位アプリケーションインターフェースの保守作業量の削減。新しい機能を追加する場合、新しい関数名や IOCTL を追加する必要はなく、ドライバの中で新しいプロパティを追加し、アプリケーションでそのプロパティの値を取得・操作するだけで済みます。

2. パラメータ設定の柔軟性の向上。一度の IOCTL で複数のプロパティを同時に設定でき、ユーザースペースとカーネルスペース間の切り替え回数を減らし、異なるハードウェアのパラメータ設定要求を最大限に満たし、ソフトウェアの効率を向上させます。

DRM のプロパティは主に機能によって分類され、多くの標準プロパティも定義されてい

株式会社日昇テクノロジー

ます。これらの標準プロパティは、どのプラットフォームでも作成されます。

表 1: CRTC

name	機能
ACTIVE	CRTC の現在の有効状態を表し、CRTC のオン/オフを制御します。
MODE_ID	CRTC の現在使用している表示モード ID。これにより具体的な表示モード設定を取得できます
OUT_FENCE_PTR	現在表示中のバッファに対応するフェンス fd へのポインタ。このフェンスは DRM ドライバによって作成され、上位アプリケーションが使用します。これは現在のバッファが CRTC に占有されているかを示します。
GAMMA_LUT	ガンマルックアップテーブルパラメータ。
GAMMA_LUT_SIZE	ガンマルックアップテーブルの長さ。

表 2: CONNECTOR

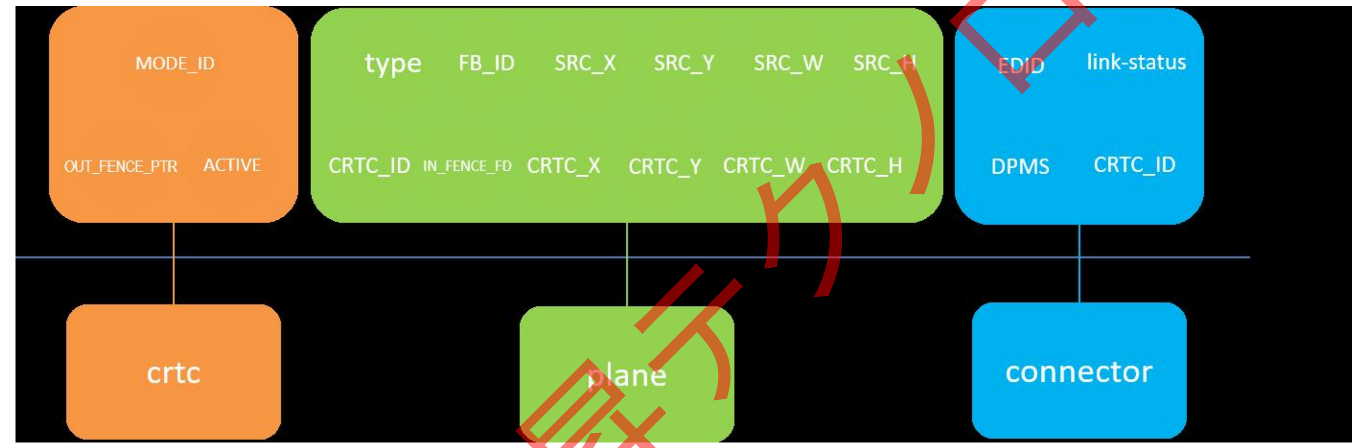
name	機能
EDID	拡張ディスプレイ識別データ。ディスプレイのパラメータ情報を示し、VESA 標準のデータフォーマットです。
DPMS	ディスプレイパワーマネジメントシグナリング。ディスプレイの電源状態を制御します（例：スリープ、ウェイクアップ）。これも VESA 標準です。
linkstatus	現在のコネクタの接続状態を示します（例：Good/Bad）。
CRTC_ID	現在のコネクタに接続されている CRTC オブジェクト ID。PLANE の CRTC_ID プロパティと同じプロパティです。

表 3: Planes

name	機能
type	plane の種類（CURSOR、PRIMARY、または OVERLAY）。
FB_ID	現在の plane にバインドされているフレームバッファオブジェクト ID。
IN_FENCE_FD	現在の plane に関連するインプットフェンス fd。これはバッファのプロデューサーが作成し、DRM ドライバが使用して現在のバッファがアクセス可能であるかを示します。

株式会社日昇テクノロジー

CRTC_ID	現在の plane に関連する CRTC オブジェクト ID。CONNECTOR の CRTC_ID プロパティと同じプロパティです。
SRC_X	現在のフレームバッファ領域の開始オフセット x 座標。
SRC_Y	現在のフレームバッファ領域の開始オフセット y 座標。
SRC_W	現在のフレームバッファ領域の幅。
SRC_H	現在のフレームバッファ領域の高さ。
CRTC_X	画面表示領域の開始オフセット x 座標。
CRTC_Y	画面表示領域の開始オフセット y 座標。
CRTC_W	画面表示領域の幅。
CRTC_H	画面表示領域の高さ。



上図のプロパティはすべてのプロパティを網羅しているわけではありません。ここでは一般的なプロパティの一部を示しています。さらに多くのプロパティについては、後のプログラム分析で説明します。

上記のプロパティを操作することで、CRTC および画面表示を設定することができます。

31.2 DRM アプリケーションプログラミング（drm-atomic-crtc）

以前のレガシーインターフェースで使用していた以下の関数：

1. drmModeSetCrtc()

2. drmModeSetPlane()

したがって、レガシーインターフェースから Atomic インターフェースに移行するには、上記のインターフェースを関連するインターフェースに変更するだけで操作可能です

株式会社日昇テクノロジー

この例では、DRM アプリケーションプログラミング–レガシーインターフェースの drm-planes 実験を基にして、レガシーインターフェースの`drmModeSetCrtc()`を atomic インターフェースに変更します。

コードリスト 1: base_linux/screen/drm/drm-atomic/drm-atomic-crtcs/drm-atomic-crtcs.c

```
#define _GNU_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <time.h>
#include <unistd.h>
#include <xf86drm.h>
#include <xf86drmMode.h>

struct drm_device {
    uint32_t width; // ディスプレイの幅のピクセル数
    uint32_t height; // ディスプレイの高さのピクセル数
    uint32_t pitch; // 各行のバイト数
    uint32_t handle; // drm_mode_create_dumb の戻り値ハンドル
```

株式会社日昇テクノロジー

```
    uint32_t size; // ディスプレイの総バイト数
    uint32_t *vaddr; // mmap の開始アドレス
    uint32_t fb_id; // 作成したフレームバッファの ID
    struct drm_mode_create_dumb create; // 作成した dumb
    struct drm_mode_map_dumb map; // メモリマップ構造体
};

struct property_crtc {
    uint32_t blob_id;
    uint32_t property_crtc_id;
    uint32_t property_mode_id;
    uint32_t property_active;
};

drmModeConnector *conn; // コネクタ関連の構造体
drmModeRes *res; // リソース
drmModePlaneRes *plane_res;

int fd; // ファイルディスクリプタ
uint32_t conn_id;
uint32_t crtc_id;
uint32_t plane_id[3];
```

株式会社日昇テクノロジー

```
#define RED 0xFF0000

#define GREEN 0x00FF00

#define BLUE 0x0000FF


uint32_t color_table[6] = {RED, GREEN, BLUE, RED, GREEN, BLUE};


struct drm_device buf;

struct property_crtc pc;


static int drm_create_fb(struct drm_device *bo) {

    /* create a dumb-buffer, the pixel format is XRGB888 */

    bo->create.width = bo->width;

    bo->create.height = bo->height;

    bo->create.bpp = 32;


    /* handle, pitch, size will be returned */

    drmIoctl(fd, DRM_IOCTL_MODE_CREATE_DUMB


, &bo->create);


    /* bind the dumb-buffer to an FB object */

    bo->pitch = bo->create.pitch;

    bo->size = bo->create.size;
```

株式会社日昇テクノロジー

```
    bo->handle = bo->create.handle;
    drmModeAddFB(fd, bo->width, bo->height, 24, 32, bo->pitch,
                 bo->handle, &bo->fb_id);

    // 各行のバイト数、総バイト数、MAP_DUMB のハンドルを表示
    printf("pitch = %d ,size = %d, handle = %d ¥n", bo->pitch, bo->size, bo->handle);

    /* map the dumb-buffer to userspace */
    bo->map.handle = bo->create.handle;
    drmIoctl(fd, DRM_IOCTL_MODE_MAP_DUMB, &bo->map);

    bo->vaddr = mmap(0, bo->create.size, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, bo->map.offset);

    /* initialize the dumb-buffer with white-color */
    memset(bo->vaddr, 0xff, bo->size);

    return 0;
}

static void drm_destroy_fb(struct drm_device *bo) {
    struct drm_mode_destroy_dumb destroy = {};
    drmModeRmFB(fd, bo->fb_id);
```

株式会社日昇テクノロジー

```
    munmap(bo->vaddr, bo->size);

    destroy.handle = bo->handle;

    drmIoctl(fd, DRM_IOCTL_MODE_DESTROY_DUMB, &destroy);
}


static uint32_t get_property(int fd, drmModeObjectProperties *props) {
    drmModePropertyPtr property;
    uint32_t i, id = 0;


    for (i = 0; i < props->count_props; i++) {
        property = drmModeGetProperty(fd, props->props[i]);
        printf("¥"%s¥"¥t¥t---", property->name);
        printf("id = %d , value=%ld¥n", props->props[i], props->prop_values[i]);
    }
    return 0;
}


static uint32_t get_property_id(int fd, drmModeObjectProperties *props,
                                const char *name) {
    drmModePropertyPtr property;
    uint32_t i, id = 0;


    /* find property according to the name */
```

株式会社日昇テクノロジー

```
    for (i = 0; i < props->count_props; i++) {
        property = drmModeGetProperty(fd, props->props[i]);
        if (!strcmp(property->name, name))
            id = property->prop_id;
        drmModeFreeProperty(property);

        if (id)
            break;
    }

    return id;
}

int drm_init() {
    int i;

    drmModeObjectProperties *props;
    drmModeAtomicReq *req;

    fd = open("/dev/dri/card0", O_RDWR | O_CLOEXEC);

    res = drmModeGetResources(fd);
    crtc_id = res->crtcs[0];
```

株式会社日昇テクノロジー

```
conn_id = res->connectors[0];

drmSetClientCap(fd, DRM_CLIENT_CAP_UNIVERSAL_PLANES, 1);
plane_res = drmModeGetPlaneResources(fd);
for (i = 0; i < 3; i++) {
    plane_id[i] = plane_res->planes[i];
    printf("planes[%d]= %d¥n", i, plane_id[i]);
}

conn = drmModeGetConnector(fd, conn_id);
buf.width = conn->modes[0].hdisplay;
buf.height = conn->modes[0].vdisplay;
drm_create_fb(&buf);

drmSetClientCap(fd, DRM_CLIENT_CAP_ATOMIC, 1);

/* get connector properties */
props = drmModeObjectGetProperties(fd, conn_id,
DRM_MODE_OBJECT_CONNECTOR);
printf("/-----conn_Property-----/¥n");
get_property(fd, props);
printf("¥n");
pc.property_crtc_id = get_property_id(fd, props, "CRTC_ID");
```


株式会社日昇テクノロジー

```
drmModeFreeObjectProperties(props);
```

```
/* get crtc properties */
props = drmModeObjectGetProperties(fd, crtc_id, DRM_MODE_OBJECT_CRTC);
printf("/-----CRTC_Property-----/¥n");
get_property(fd, props);
printf("¥n");
pc.property_active = get_property_id(fd, props, "ACTIVE");
pc.property_mode_id = get_property_id(fd, props, "MODE_ID");
drmModeFreeObjectProperties(props);

/* create blob to store current mode, and return the blob id */
drmModeCreatePropertyBlob(fd, &conn->modes[0],
                    sizeof(conn->modes[0]), &pc.blob_id);

/* start modesetting */
req = drmModeAtomicAlloc();
drmModeAtomicAddProperty(req, crtc_id, pc.property_active, 1);
drmModeAtomicAddProperty(req, crtc_id, pc.property_mode_id, pc.blob_id);
drmModeAtomicAddProperty(req, conn_id, pc.property_crtc_id, crtc_id);
drmModeAtomicCommit(fd, req, DRM_MODE_ATOMIC_ALLOW_MODESET, NULL);
drmModeAtomicFree(req);
}
```

株式会社日昇テクノロジー

```
int drm_exit() {
    drm_destroy_fb(&buf);
    drmModeFreeConnector(conn);
    drmModeFreePlaneResources(plane_res);
    drmModeFreeResources(res);
    close(fd);
}

int main(int argc, char **argv) {
    int i, j;
    drm_init();
    // 表示 3 色
    for (j = 0; j < 3; j++) {
        for (i = j * buf.width * buf.height / 3; i < (j + 1) * buf.width * buf.height / 3; i++)
            buf.vaddr[i] = color_table[j];
    }
    // 1：1 で画面設定、これがないと画面が表示されません
    drmModeSetPlane(fd, plane_id[0], crtc_id, buf.fb_id, 0,
                0, 0, buf.width, buf.height,
                0 << 16, 0 << 16, buf.width << 16, buf.height << 16);

    getchar();
```

株式会社日昇テクノロジー

// フレームバッファの上 2/3 の領域をレイヤー 1 に配置、

// このとき画面が変わり、フレームバッファの領域が画面全体に拡大されます

```
drmModeSetPlane(fd, plane_id[0], crtc_id, buf.fb_id, 0,
        0, 0, buf.width, buf.height,
        0 << 16, 0 << 16, buf.width << 16, buf.height / 3 * 2 << 16);

getchar();
// フレームバッファ領域を 2 倍に縮小してレイヤー 2 に配置し、レイヤー 2 を画面の下部に配置
// レイヤー 1 に重ねて表示され、レイヤー 2 がレイヤー 1 の一部を覆うことが確認できます
drmModeSetPlane(fd, plane_id[1], crtc_id, buf.fb_id, 0,
        buf.width / 4, buf.height / 2, buf.width / 2, buf.height / 2,
        0 << 16, 0 << 16, buf.width << 16, buf.height << 16);

getchar();
drm_exit();

return 0;
}
```

31.2.1 コンパイル

方法 1：

株式会社日昇テクノロジー

```
1. # コンパイル

2. make
```

方法 2：

```
1. # コンパイル

2. gcc -o drm-atomic-crtcs drm-atomic-crtcs.c $(pkg-config --cflags libdrm) $(pkg-config --
libs libdrm)
```

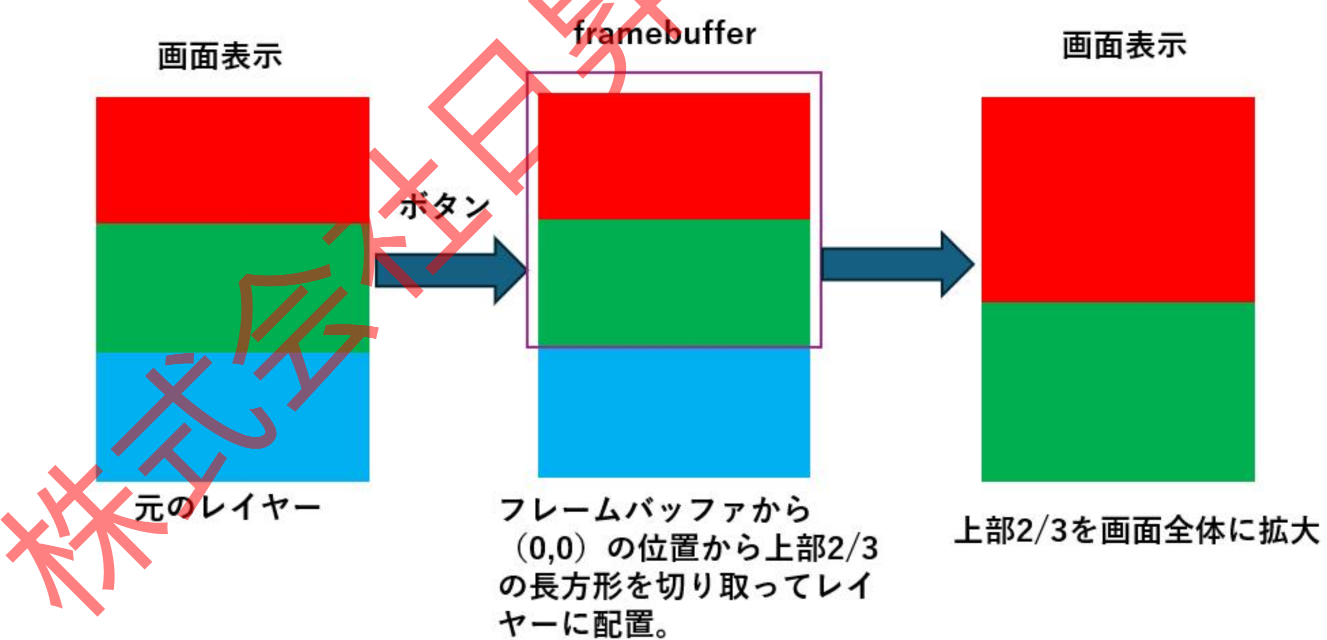
31.2.2 実行

実行結果：上記の drm-planes.c と同じ

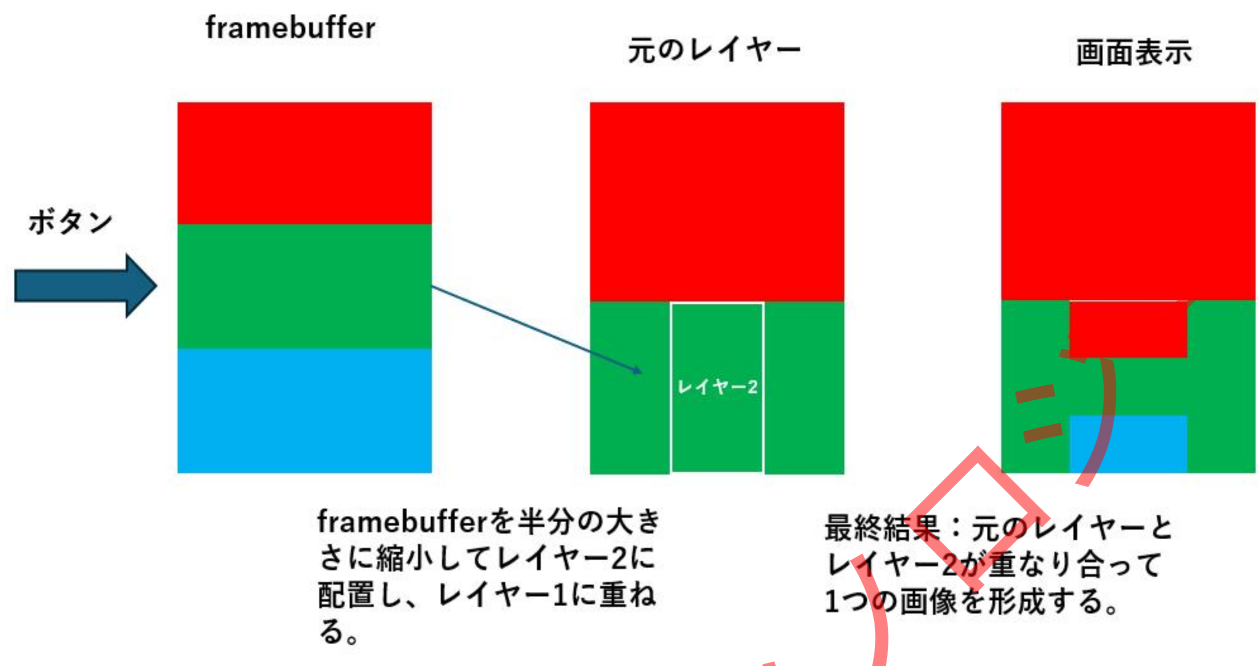
```
./drm-atomic-ctrcs
```

実験現象：

ボタンを押すと、画面に赤、緑、青の 3 本の横線が表示されます。



株式会社日昇テクノロジー



31.2.3 実験分析

この例は、DRM アプリケーションプログラミング–drm-planes 実験のフレームを基に、レガシーインターフェースを変更して得られたものです。ここでは、drm-planes と異なる部分のみを分析します。

コードリスト 2: drm_init

```
struct property_crtc {
    uint32_t blob_id;
    uint32_t property_crtc_id;
    uint32_t property_mode_id;
    uint32_t property_active;
};


int drm_init() {
```

株式会社日昇テクノロジー

```
int i;


drmModeObjectProperties *props;

drmModeAtomicReq *req;


fd = open("/dev/dri/card0", O_RDWR | O_CLOEXEC);


res = drmModeGetResources(fd);

crtc_id = res->crtcs[0];

conn_id = res->connectors[0];


drmSetClientCap(fd, DRM_CLIENT_CAP_UNIVERSAL_PLANES, 1);

plane_res = drmModeGetPlaneResources(fd);

for (i = 0; i < 3; i++) {

    plane_id[i] = plane_res->planes[i];

    printf("planes[%d]= %d¥n", i, plane_id[i]);

}


conn = drmModeGetConnector(fd, conn_id);

buf.width = conn->modes[0].hdisplay;

buf.height = conn->modes[0].vdisplay;

drm_create_fb(&buf);
```

株式会社日昇テクノロジー

```
    drmSetClientCap(fd, DRM_CLIENT_CAP_ATOMIC,

1);

    /* get connector properties */
    props = drmModeObjectGetProperties(fd, conn_id,
DRM_MODE_OBJECT_CONNECTOR);
    printf("/-----conn_Property-----/¥n");
    get_property(fd, props);
    printf("¥n");
    pc.property_crtc_id = get_property_id(fd, props, "CRTC_ID");
    drmModeFreeObjectProperties(props);

    /* get crtc properties */
    props = drmModeObjectGetProperties(fd, crtc_id, DRM_MODE_OBJECT_CRTC);
    printf("/-----CRTC_Property-----/¥n");
    get_property(fd, props);
    printf("¥n");
    pc.property_active = get_property_id(fd, props, "ACTIVE");
    pc.property_mode_id = get_property_id(fd, props, "MODE_ID");
    drmModeFreeObjectProperties(props);

    /* create blob to store current mode, and return the blob id */
```

株式会社日昇テクノロジー

```
drmModeCreatePropertyBlob(fd, &conn->modes[0],
                    sizeof(conn->modes[0]), &pc.blob_id);


/* start modesetting */
req = drmModeAtomicAlloc();
drmModeAtomicAddProperty(req, crtc_id, pc.property_active, 1);
drmModeAtomicAddProperty(req, crtc_id, pc.property_mode_id, pc.blob_id);
drmModeAtomicAddProperty(req, conn_id, pc.property_crtc_id, crtc_id);
drmModeAtomicCommit(fd, req, DRM_MODE_ATOMIC_ALLOW_MODESET, NULL);
drmModeAtomicFree(req);
}
```

- 第 15-31 行は drm-planes.c の初期化コードフレームと同じため、ここでは詳細な説明は省略します。

- 第 33 行、drmSetClientCap(fd, DRM_CLIENT_CAP_ATOMIC, 1); は、DRM_CLIENT_CAP_ATOMIC を 1 に設定すると、DRM コアはユーザー空間にアトミックプロパティを公開します。このオプションを 1 に設定すると、DRM ドライバーフレームワークコアはアトミックプロパティをユーザー空間に公開し、このオプションを設定することでアトミックインターフェースを使用できるようになります。

- 第 36 行と 44 行で、特定のリソースの props ポインタを取得し、後で props ポインタを介して property_id を取得しやすくします。

株式会社日昇テクノロジー

```
1. // 関数プロトタイプ
2. extern drmModeObjectPropertiesPtr drmModeObjectGetProperties(int fd,
3. uint32_t object_id,
4. uint32_t object_type);


5. //props ポインタプロトタイプ
6. typedef struct _drmModeObjectProperties {
7. uint32_t count_props; //props の数
8. //props ポインタ、id 配列を指します。props_id によってプロパティを取得できます。
9. uint32_t *props;
10. uint64_t *prop_values; //プロパティ値の配列を指します。
11. } drmModeObjectProperties, *drmModeObjectPropertiesPtr;
```

- fd: ファイルディスクリプタ。
- object_id: 目標 ID。第 18、19 行で取得可能。
- object_type: 目標タイプ。以下のマクロ定義。

```
1. #define DRM_MODE_OBJECT_CRTC 0xcccccccc //CRTC
2. #define DRM_MODE_OBJECT_CONNECTOR 0xc0c0c0c0 //CONNECTOR
3. #define DRM_MODE_OBJECT_ENCODER 0xe0e0e0e0 //ENCODER
4. #define DRM_MODE_OBJECT_MODE 0xdededede //mode
5. #define DRM_MODE_OBJECT_PROPERTY 0xb0b0b0b0 //PROPERTY
6. #define DRM_MODE_OBJECT_FB 0xfbfbfbfb //FB
7. #define DRM_MODE_OBJECT_BLOB 0xbbbbbbbb //BLOB
8. #define DRM_MODE_OBJECT_PLANE 0xeeeeeeee //PLANE
```

株式会社日昇テクノロジー

```
9. #define DRM_MODE_OBJECT_ANY 0 //all
```

• 第 37-39、45-47 行で、各 props のプロパティを出力します。後で説明します。

• 第 40、48、49 行で get_property_id(fd, props, "XXXX");を使用して名前と一致する property_id を取得します。

• 第 41、50 行で drmModeFreeObjectProperties();を使用して取得した props リソースを解放します。

• 第 53-54 行で、drmModeCreatePropertyBlob を使用して Blob_property を作成し、カスタムの長さのメモリブロックを作成してカスタム構造体データを格納します。ここでは、CRTC の mode を blob に書き込み、MODE_ID の設定に使用します。

• 第 56-62 行で、先ほど取得した property_id を使用して CRTC を設定します。

• 第 57 行で、アトミック操作のトークンを取得し、req を取得してから操作を続行します。

```
1. // 関数プロトタイプ
2. extern drmModeAtomicReqPtr drmModeAtomicAlloc(void);
3. // トークンを返す
```

• 第 58-60 行で、プロパティの属性を設定します。

1. "ACTIVE"を 1 に設定して CRTC を有効にする。

2. "MODE_ID"に先ほど作成した blob_id を渡し、ドライバは blob の構造体を参照して CRTC のスキャン領域を作成する。

3. "CRTC_ID"に CRTC_ID を渡して conn_id に設定し、コネクタと CRTC を接続する。

```
1. // 関数プロトタイプ
2. extern int drmModeAtomicAddProperty(drmModeAtomicReqPtr req,
```

株式会社日昇テクノロジー

```
3. uint32_t object_id, uint32_t property_id, uint64_t value);
```

• 第 61 行で、drmModeAtomicCommit();を使用してすべてのプロパティを一括でコミットします。このコミット操作は成功するか、または元の状態を維持するかのいずれかであり、途中で操作が失敗した場合は、成功した設定が元の状態に戻され、まるでコミット操作が行われなかったかのようになります。これがアトミックの意味です。成功したかどうかは戻り値で確認できます。通常、戻り値が負の値の場合、操作は失敗しています。

```
1. // 関数プロトタイプ
2. extern int drmModeAtomicCommit(int fd,
3. drmModeAtomicReqPtr req,
4. uint32_t flags,
5. void *user_data);
```

• 第 62 行で、drmModeAtomicFree(req);を使用してトークンを解放します。

```
1. // 関数プロトタイプ
2. extern void drmModeAtomicFree(drmModeAtomicReqPtr req);
```

コードリスト 3: get_property

```
static uint32_t get_property(int fd, drmModeObjectProperties *props) {
  drmModePropertyPtr property;
  uint32_t i, id = 0;

  for (i = 0; i < props->count_props; i++) {
    property = drmModeGetProperty(fd, props->props[i]);
    printf("¥"%s¥"¥t¥t---", property->name);
```

株式会社日昇テクノロジー

```
        printf("id = %d , value=%ld¥n", props->props[i], props->prop_values[i]);
    }
```

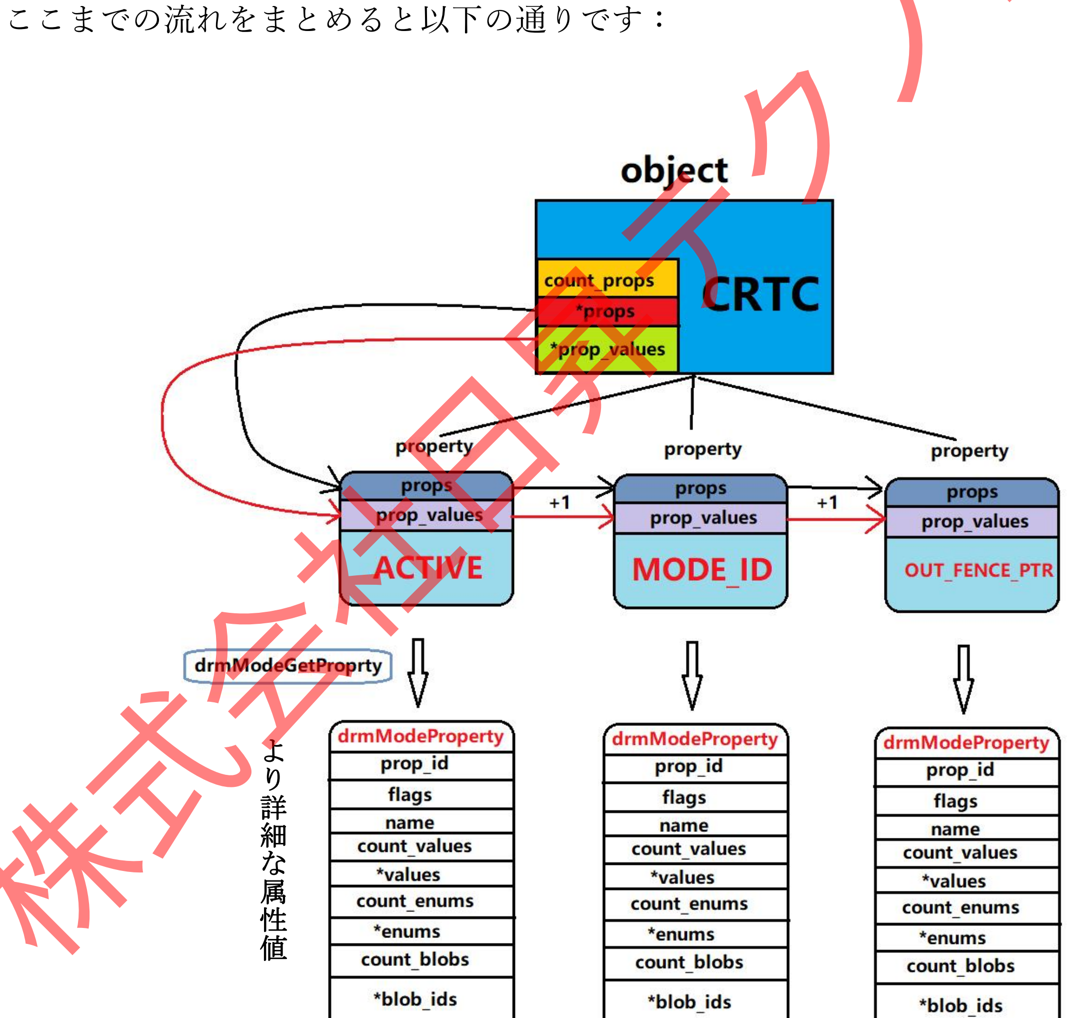
```
    return 0;
}


static uint32_t get_property_id(int fd, drmModeObjectProperties *props,
                                const char *name) {
    drmModePropertyPtr property;
    uint32_t i, id = 0;

    /* find property according to the name */
    for (i = 0; i < props->count_props; i++) {
        property = drmModeGetProperty(fd, props->props[i]);
        if (!strcmp(property->name, name))
            id = property->prop_id;
        drmModeFreeProperty(property);

        if (id)
            break;
    }

    return id;
}
```

株式会社日昇テクノロジー

- 第 1-15 行は、すべてのプロパティの名前、ID、および値を表示するためのものです。
- 第 7 行は、ID 番号に基づいてプロパティの具体的なリソースを取得します。
- 第 14-33 行は、文字列の一致に基づいてプロパティ ID を取得するためのものです。
- 第 23 行は、プロパティの具体的なリソースを取得します。
- 第 24,25 行は、文字列を比較し、一致した場合はループを終了します。
- 第 26 行は、取得したリソースを解放します。

31.2.4 まとめ

ここまでの流れをまとめると以下の通りです：



- CRTC の設定を行う際、CRTC にフレームバッファがバインドされていないため、

株式会社日昇テクノロジー

CRTC の設定だけでは画面を表示できません。Planes の設定を行う必要があります。

Atomic インターフェースを使用した CRTC の初期化手順：

1. Atomic を設定し、Atomic インターフェースをユーザースペースで使用可能にする。
2. conn_id を使用して"CRTC_ID"のプロパティ ID を取得し、リソースを解放する。
3. crtc_id を使用して"ACTIVE"および"MODE_ID"のプロパティ ID を取得し、リソースを解放する。
4. blob を作成し、CRTC のモードを格納する。
5. アトミック操作のキーを取得する。
6. プロパティ ID を使用して属性を設定する。connector は"CRTC_ID"を設定し、CRTC は"ACTIVE"および"MODE_ID"を設定する。
7. 設定を一括してコミットする。コミットが成功しない場合は、以前の設定が維持される。
8. アトミック操作のキーを解放する。

31.3 DRM アプリケーションプログラミング（drm-atomic-planes）

この例では、DRM アプリケーションプログラミング–drm-atomic-crtc 実験を基に、レガシーインターフェースの`drmModeSetPlane()`を atomic インターフェースに変更します。

コードリスト 4: コード位置

```
base_linux/screen/drm/drm-atomic/drm-atomic-planes/drm-atomic-planes.c
```

31.3.1 コンパイル

方法 1：

```
1. # コンパイル
```

株式会社日昇テクノロジー

2. make

方法 2：

```
1. # コンパイル
2. gcc -o drm-atomic-planes drm-atomic-planes.c $(pkg-config --cflags libdrm) $(pkg-config --libs libdrm)
```

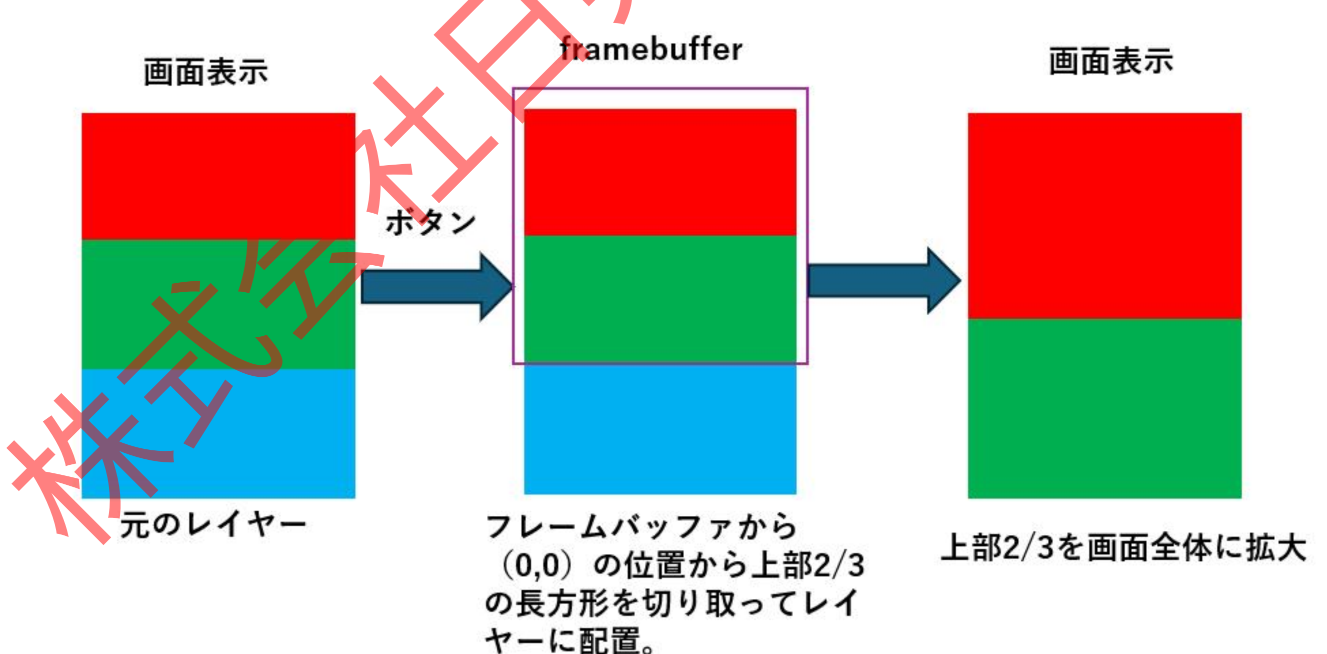
31.3.2 実行

実行結果：上記の drm-planes.c と同じ

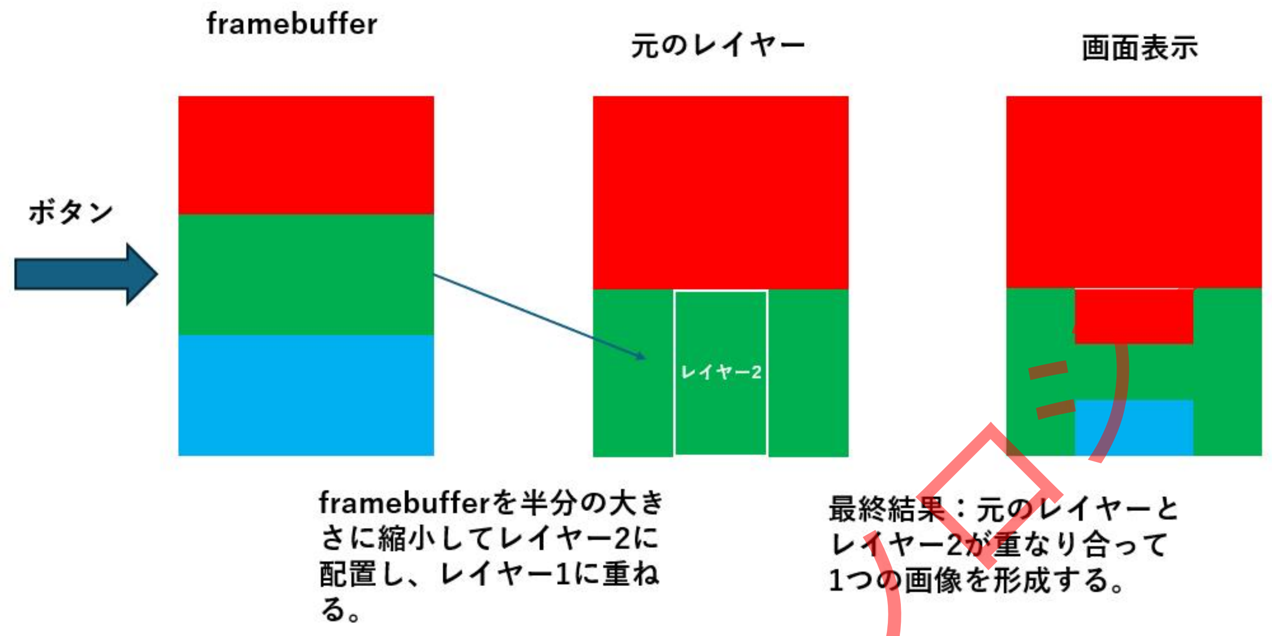
```
./drm-atomic-planes
```

実験現象：

ボタンを押すと、画面に赤、緑、青の三本の横線が表示されます。



株式会社日昇テクノロジー



31.3.3 実験分析

この例は、DRM アプリケーションプログラミング–drm-atomic-crtc 実験を基に、部分的に変更したものです。ここでは、差異のある部分のみを分析します。

コードリスト 5: プロパティの取得

```
struct property_planes {
    uint32_t plane_id;
    uint32_t property_fb_id;
    uint32_t property_crtc_x;
    uint32_t property_crtc_y;
    uint32_t property_crtc_w;
    uint32_t property_crtc_h;
    uint32_t property_src_x;
    uint32_t property_src_y;
```

株式会社日昇テクノロジー

```
    uint32_t property_src_w;
    uint32_t property_src_h;
};

static uint32_t drm_get_plane_property_id(int fd, uint32_t plane_id) {
    drmModeObjectProperties *props;
    int i, num;
    for (i = 0; i < buf.count_plane + 1; i++) {
        if (pp[i].plane_id == plane_id) {
            num = i;
            break;
        } else {
            num = buf.count_plane;
            buf.count_plane++;
            break;
        }
    }
    pp[num].plane_id = plane_id;
    /* get plane properties */
    props = drmModeObjectGetProperties(fd, plane_id, DRM_MODE_OBJECT_PLANE);
    if (printf_status == 1) {
        get_property(fd, props);
        printf_status = 0;
```

株式会社日昇テクノロジー

```
 }
 pp[num].property_fb_id = get_property_id(fd, props, "FB_ID");
 pp[num].property_crtc_x = get_property_id(fd, props, "CRTC_X");
 pp[num].property_crtc_y = get_property_id(fd, props, "CRTC_Y");
 pp[num].property_crtc_w = get_property_id(fd, props, "CRTC_W");
 pp[num].property_crtc_h = get_property_id(fd, props, "CRTC_H");
 pp[num].property_src_x = get_property_id(fd, props, "SRC_X");
 pp[num].property_src_y = get_property_id(fd, props, "SRC_Y");
 pp[num].property_src_w = get_property_id(fd, props, "SRC_W");
 pp[num].property_src_h = get_property_id(fd, props

, "SRC_H");
 drmModeFreeObjectProperties(props);
 buf.count_plane++;
}
```

- 第 1-12 行は、plane に必要なプロパティを含む構造体を定義したものです。複数の plane を操作する際には構造体配列を定義し、その内容を格納します。

- 第 14-47 行は、指定された plane_id に基づいて必要なプロパティを含む構造体を自動生成する関数です。

- 第 32-35 行は、plane 上のすべてのプロパティを取得し、表示します。さらに開発を進める場合は、ソースコードを読み解くか、リソースを参照してください。

株式会社日昇テクノロジー

コードリスト 6: planes の設定

```
struct planes_setting {
    uint32_t crtc_id;
    uint32_t plane_id;
    uint32_t fb_id;
    uint32_t crtc_x;
    uint32_t crtc_y;
    uint32_t crtc_w;
    uint32_t crtc_h;
    uint32_t src_x;
    uint32_t src_y;
    uint32_t src_w;
    uint32_t src_h;
};

static uint32_t drm_set_plane(int fd, struct planes_setting *ps) {
    drmModeAtomicReq *req;
    int i;
    int num;
    uint32_t plane = ps->plane_id;
    for (i = 0; i < buf.count_plane + 1; i++) {
        if (pp[i].plane_id == ps->plane_id) {
            num = i;
```

株式会社日昇テクノロジー

```
            break;
        }
    }

    req = drmModeAtomicAlloc();
    drmModeAtomicAddProperty(req, ps->plane_id, pc.property_crtc_id, crtc_id);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_fb_id, ps->fb_id);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_crtc_x, ps->crtc_x);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_crtc_y, ps->crtc_y);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_crtc_w, ps->crtc_w);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_crtc_h, ps->crtc_h);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_src_x, ps->src_x <<
16);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_src_y, ps->src_y <<
16);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_src_w, ps->src_w <<
16);
    drmModeAtomicAddProperty(req, ps->plane_id, pp[num].property_src_h, ps->src_h <<
16);
    drmModeAtomicCommit(fd, req, 0, NULL);
    drmModeAtomicFree(req);
}
```

株式会社日昇テクノロジー

- 第 1-13 行は、planes のプロパティを設定するための構造体を定義しています。

- 第 15-41 行は、指定された構造体に基づいて planes のプロパティを自動的に設定する関数を定義しています。

コードリスト 7: main

```
int main(int argc, char **argv) {
    int i, j;
    drmModeAtomicReq *req;
    struct planes_setting ps5;

    drm_init();
    drm_get_plane_property_id(fd, plane_id[0]);
    drm_get_plane_property_id(fd, plane_id[1]);
    // 表示 3 色
    for (j = 0; j < 3; j++) {
        for (i = j * buf.width * buf.height / 3; i < (j + 1) * buf.width * buf.height / 3; i++)
            buf.vaddr[i] = color_table[j];
    }
    // 1：1 で画面設定、これがないと画面が表示されません
    ps5.plane_id = plane_id[0];
    ps5.fb_id = buf.fb_id;
    ps5.crtc_x = 0;
    ps5.crtc_y = 0;
    ps5.crtc_w = buf.width;
```

株式会社日昇テクノロジー

```
ps5.crtc_h = buf.height;
ps5.src_x = 0;
ps5.src_y = 0;
ps5.src_w = buf.width;
ps5.src_h = buf.height;
drm_set_plane(fd, &ps5);


getchar();
// フレームバッファの上 2/3 の領域をレイヤー 1 に配置、
// このとき画面が変わり、フレームバッファの領域が画面全体に拡大されます
ps5.src_w = buf.width;
ps5.src_h = buf.height / 3 * 2;
drm_set_plane(fd, &ps5);


getchar();
// フレームバッファ領域を 2 倍に縮小してレイヤー 2 に配置し、レイヤー 2 を画面の下部に配置
// レイヤー 1 に重ねて表示され、レイヤー 2 がレイヤー 1 の一部を覆うことが確認できます
ps5.plane_id = plane_id[1];
ps5.fb_id = buf.fb_id;
ps5.crtc_x = buf.width / 4;
ps5.crtc_y = buf.height / 2;
```

株式会社日昇テクノロジー

```
    ps5.crtc_w = buf.width / 2;
    ps5.crtc_h = buf.height / 2;
    ps5.src_x = 0;
    ps5.src_y = 0;
    ps5.src_w = buf.width;
    ps5.src_h = buf.height;
    drm_set_plane(fd, &ps5);
    // PS5 の価格が上がっている、お金が足りない、くそ！
    // 待て！俺の車が安くなりそうだ、PC ゲームに突入だ！
    getchar();
    drm_exit();

    return 0;
}
```

- 第 8-9 行は、plane[0]と plane[1]のプロパティを取得し、構造体に保存します。

- 第 18-28 行は、構造体のデータを編集し、`drm_set_plane()`に渡して plane の設定を完了します。

- 第 40-49 行は、plane[1]を表示する設定を行います。

31.3.4 まとめ

atomic インターフェースの plane 設定と crtc 設定のフレームは基本的に同じです。これらをそのまま使用することができます。他のパラメータも同様に設定可能です。

株式会社日昇テクノロジー

31.4 DRM アプリケーションプログラミング（drm-atomic-page-flip）

コードリスト 8: コード位置

```
base_linux/screen/drm/drm-atomic/drm-atomic-page-flip/drm-atomic-page-flip.c
```

31.4.1 コード分析

この実験は、前の実験および drm-page-flip 実験を基に修正されたものです。修正内容は以下の通りです。

コードリスト 9: フレームバッファの作成

```
static int drm_create_fb(struct drm_device *bo) {
    /* create a dumb-buffer, the pixel format is XRGB888 */
    bo->create.width = bo->width;
    bo->create.height = bo->height * 2;
    bo->create.bpp = 32;

    /* handle, pitch, size will be returned */
    drmIoctl(fd, DRM_IOCTL_MODE_CREATE_DUMB, &bo->create);

    /* bind the dumb-buffer to an FB object */
    bo->pitch = bo->create.pitch;
    bo->size = bo->create.size;
    bo->handle = bo->create.handle;
    drmModeAddFB(fd, bo->width, bo->height * 2, 24, 32, bo->pitch,
```

株式会社日昇テクノロジー

```
            bo->handle, &bo->fb_id);


    // 各行のバイト数、総バイト数、MAP_DUMB のハンドルを表示
    printf("pitch = %d ,size = %d, handle = %d ¥n", bo->pitch, bo->size, bo->handle);


    /* map the dumb-buffer to userspace */
    bo->map.handle = bo->create.handle;
    drmIoctl(fd, DRM_IOCTL_MODE_MAP_DUMB, &bo->map);


    bo->vaddr = mmap(0, bo->create.size, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, bo->map.offset);


    /* initialize the dumb-buffer with white-color */
    memset(bo->vaddr, 0xff, bo->size);


    return 0;
}
```

- フレームバッファを 2 倍に拡大し、オフセットを変更することでダブルバッファリングを実現します。

株式会社日昇テクノロジー

コードリスト 10: set_crtc

```
static uint32_t drm_set_crtc(void) {
    drmModeAtomicReq *req;
    req = drmModeAtomicAlloc();
    drmModeAtomicAddProperty(req, crtc_id, pc.property_active, 1);
    drmMode

AtomicAddProperty(req, crtc_id, pc.property_mode_id, pc.blob_id);
    drmModeAtomicAddProperty(req, conn_id, pc.property_crtc_id, crtc_id);
   drmModeAtomicCommit(fd, req, DRM_MODE_ATOMIC_ALLOW_MODESET |
DRM_MODE_PAGE_FLIP_EVENT, NULL);
    drmModeAtomicFree(req);
    return 0;
}
```

- crtc のアトミックコミットを別の関数に移動し、呼び出しやすくします。

- drmModeAtomicCommit(fd, req, DRM_MODE_ATOMIC_ALLOW_MODESET | DRM_MODE_PAGE_FLIP_NULL)にフラグ DRM_MODE_PAGE_FLIP_EVENT を追加し、この設定により crtc がページフリップ機能を持つようになります。

株式会社日昇テクノロジー

コードリスト 11: drm_page_flip_handler

```
static void drm_page_flip_handler(int fd, uint32_t frame,
                                  uint32_t sec, uint32_t usec,
                                  void *data) {
    drmModeAtomicReq *req;
    // 表示を切り替えます
    if (count == 0)
        count = 1;
    else if (count == 1)
        count = 0;
    // オフセットを設定します
    if (count == 1)
        ps5.src_y = 1280;
    else
        ps5.src_y = 0;
    drm_set_plane(fd, &ps5);
    drm_set_crtc();
}
```

- planes を設定した後、crtc の設定も再度行う必要があります。さもないと、シグナルを送信し続けるとブロック状態に入り、プログラムがそこで停止します。

株式会社日昇テクノロジー

コードリスト 12: main 関数

```
int main(int argc, char **argv) {
    int i, j;
    drmModeAtomicReq *req;
    ev.version = DRM_EVENT_CONTEXT_VERSION;
    ev.page_flip_handler = drm_page_flip_handler;

    drm_init();
    drm_get_plane_property_id(fd, plane_id[0]);
    drm_get_plane_property_id(fd, plane_id[1]);

    for (i = 0; i < buf.width * buf.height; i++)
        buf.vaddr[i] = RED;

    for (i = buf.width * buf.height; i < buf.width * buf.height * 2; i++)
        buf.vaddr[i] = BLUE;

    ps5.plane_id = plane_id[0];
    ps5.fb_id = buf.fb_id;
    ps5.crtc_x = 0;
    ps5.crtc_y = 0;
    ps5.crtc_w = buf.width;
    ps5.crtc_h = buf.height;
```

株式会社日昇テクノロジー

```
    ps5.src_x = 0;

    ps5.src_y = 0;

    ps5.src_w = buf.width;

    ps5.src_h = buf.height;

    drm_set_plane(fd, &ps5);


    getchar();

    drmHandleEvent(fd, &ev);

    getchar();

    drmHandleEvent(fd, &ev);

    getchar();

    drm_exit();


    return 0;

}
```

- drm_page_flip_handler を設定します。

31.4.2 コンパイル

方法 1：

```
1. # コンパイル

2. make
```

株式会社日昇テクノロジー

方法 2：

```
1. # コンパイル
2. gcc -o drm-atomic-page-flip drm-atomic-page-flip.c $(pkg-config --cflags libdrm) $(pkg-config --libs libdrm)
```

31.4.3 実行

```
./drm-atomic-page-flip
```

実験現象：開始-> 赤-> 青-> 赤-> 終了（ボタンを押して次のステップに進みます）。

31.5 Makefile 管理

コードリスト 13: コード位置

```
base_linux/screen/drm/project
```

最後の実験では、コード量が 360 行を超えていることがわかります。main 関数が比較的短い場合でも、コードが増えると読みにくくなります。そのため、ここでまとめとして、drm-atomc-planes.c をプロジェクトに変更し、make コマンドを使用してコンパイルを行い、Makefile を使用して管理します。

```
1 # ソースフォルダー
2
3 project/
4 |-- Makefile #make コンパイル
5 |-- includes
6 | `-- drm-core.h # 関数定義および構造体を格納
7 `-- sources
8 |-- main.c # メインプログラム
```

株式会社日昇テクノロジー

```
9 `-- drm-core.c # DRM 表示コア関数を含む c ファイル

11 # コンパイル後

12 project/

13 |-- Makefile #make コンパイル

14 |-- build

15 | |-- drm-core.o

16 | |-- main.o

17 | `-- test # 実行可能バイナリープログラム（名前は Makefile で変更可能）

18 |-- includes

19 | `-- drm-core.h

20 |-- sources

21 | |-- drm-core.c

22 | `-- main.c

23 `-- test -> build/test # 実行ファイルへのシンボリックリンク。フォルダーを移動せずに
実行可能
```

main.c は以下の通り：

リスト 14: project/sources/main.c

```
1 #include "drm-core.h"

2

3 uint32_t color_table[6] = {RED,GREEN,BLUE,BLACK,WHITE,BLACK_BLUE};

4

5 int main(int argc, char **argv)

6 {
```

株式会社日昇テクノロジー

```
7 int i,j;

8 drm_init();

9

10 getchar();

11 for(j = 0; j< 6; j++){

12 for(i = 0;i< buf.width*buf.height;i++)

13 buf.vaddr[i] = color_table[j];

14 getchar();

15 }

16

17 drm_exit();

18

19 return 0;

20 }
```

リスト 15: project/Makefile

```
1 # 変数定義

2 TARGET = test

3 # Makefile のパスに実行プログラムをシンボリックリンクとして作成

4 OTHER_TARGET = test

5 # 中間ファイルのパス

6 BUILD_DIR = build

7 # ソースファイルのフォルダー

8 SRC_DIR = sources
```

株式会社日昇テクノロジー

```
9 # ヘッダーファイルのフォルダー
10 INC_DIR = includes
11 # ソースファイル
12 SRCS = $(wildcard $(SRC_DIR)/*.c)
13 # 目標ファイル（*.o）
14 OBJS = $(patsubst %.c, $(BUILD_DIR)/%.o, $(notdir $(SRCS)))
15 # ヘッダーファイル
16 DEPS = $(wildcard $(INC_DIR)/*.h)
17 # ヘッダーファイルのパス指定
18 CFLAGS = $(patsubst %, -I%, $(INC_DIR))
19 # コンパイラ
20 CC = gcc
21 # ライブラリ参照
22 LIBDRM = `pkg-config --cflags libdrm` `pkg-config --libs libdrm`
23 # 目標ファイル
24 $(BUILD_DIR)/$(TARGET): $(OBJS)
25 $(CC) -o $@ $^ $(CFLAGS) $(LIBDRM)
26 @ln -sf $(BUILD_DIR)/$(TARGET) $(OTHER_TARGET)
27
28 # *.o ファイル生成ルール
29 $(BUILD_DIR)/%.o: $(SRC_DIR)/%.c $(DEPS)
30
31 # 中間ファイルを格納するビルドディレクトリを作成
```

株式会社日昇テクノロジー

```
32 # コマンドの前に「@」を付けると、ターミナルに出力しない
33 @mkdir -p $(BUILD_DIR)
34 $(CC) -c -o $@ $< $(CFLAGS) $(LIBDRM)
35
36 # 擬似ターゲット
37 .PHONY: clean cleanall
38
39 # 出力フォルダー削除
40 clean:
41 rm -rf $(BUILD_DIR)
42 @rm $(OTHER_TARGET)
43
44 # 全削除
45 cleanall:
46 rm -rf $(BUILD_DIR)
47 @rm $(OTHER_TARGET)
```

31.6 スクリーン二つそれぞれ表示

rk3588 のスクリーン二つそれぞれ表示の設定時には、Planes の設定にいくつかの手順が必要です。

3 つのスクリーンがアクティブな場合の分配：

1. VP0–>HDMI（プラグイン可能）–>2
2. VP1–>MIPI（メインスクリーン）–>3
3. VP2–>DP（VGA 変換可能、プラグイン可能）–>1

株式会社日昇テクノロジー

例として、HDMI と MIPI スクリーンを同時にアクティブにする場合：

- drm_init()`で`crtc_id = res->crtcs[0]を設定して HDMI の CRTC を取得します。

- drm_init()`で`crtc_id = res->crtcs[1]を設定して MIPI スクリーンの CRTC を取得します。

Planes の設定にはいくつかの追加手順が必要です。

```
// すべての planes リソースを取得します
plane_res = drmModeGetPlaneResources(fd);
for (i = 0; i < plane_res->count_planes; i++) {
   plane_id[i] = plane_res->planes[i];
```

Planes のバインディング情報は以下の関数を使用して取得します。

```
// plane_id を使用して drmModePlanePtr リソースを取得します
extern drmModePlanePtr drmModeGetPlane(int fd, uint32_t plane_id);


typedef struct _drmModePlane {
   uint32_t count_formats; // フォーマットの数
   uint32


_t *formats; // フォーマットのポインタ
   uint32_t plane_id;
   uint32_t crtc_id; // バインドされた crtc_id
   uint32_t fb_id; // バインドされた fb_id
   uint32_t crtc_x, crtc_y;
   uint32_t x, y;
```

株式会社日昇テクノロジー

```
    uint32_t possible_crtcs; // バインド可能な crtc_id

    uint32_t gamma_size;

} drmModePlane, *drmModePlanePtr;
```

本ボードでは、PRIMARY_PLANE は crtc_id から取得できますが、その他の Planes は possible_crtcs から取得する必要があります。

- possible_crtcs = 1 -> HDMI

- possible_crtcs = 2 -> MIPI

以下に対応する Planes を示します。

- HDMI -> plane_id[0], plane_id[2], plane_id[5]

- MIPI -> plane_id[1], plane_id[3], plane_id[4]

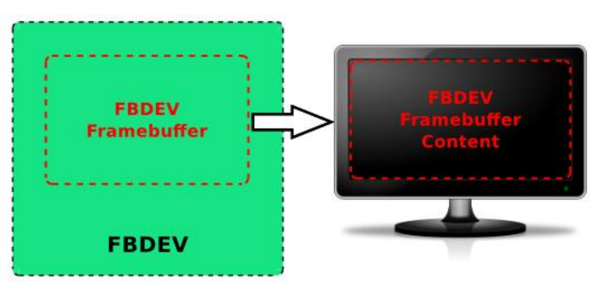
これを基に、Planes を設定することができます。

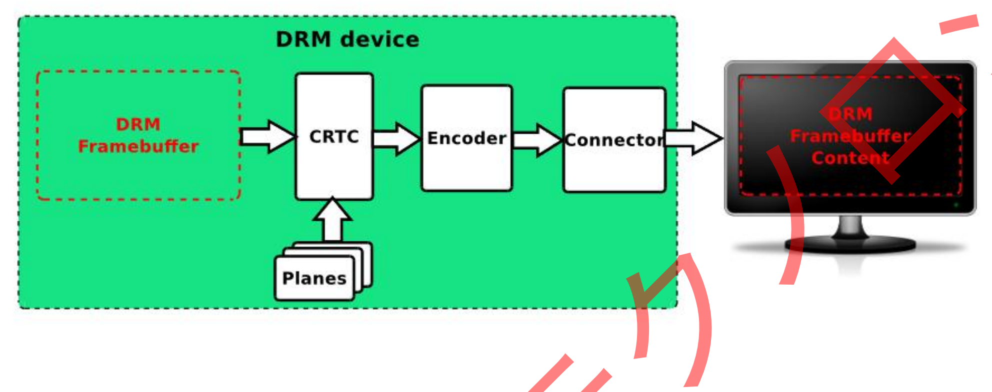
第 32 章 文字表示

```
1 リポジトリを取得します。

2 git clone https://github.com/LubanCat/lubancat_rk_code_storage.git

3

4 この章のコードは以下の場所にあります。

5 lubancat_rk_code_storage/base_linux/screen/char/
```

前の章で、framebuffer と drm のアプリケーション開発について紹介しました。framebuffer と drm のフレームワーク図から、framebuffer と drm の出発点と終点が一致していることがわかります。

株式会社日昇テクノロジー





出発点：framebuffer ---> 終点：画面出力

画面上で操作するには、framebuffer を操作するだけで画面の出力を制御できます。文字や画像の出力は、framebuffer（メモリの一部）を変更することで実現できます。この章の表示文字の実験では、前章の drm のプロジェクトを使用して実験します。（drm の設定やレイヤー設計をスキップして、画面の変更や framebuffer の使用方法についても同様です）

32.1 ASCII 文字の表示（8x16）

この章の実験では、画面上に 256 個の ASCII 文字を表示する機能を実装します。

32.1.1 実装方法

画面上に ASCII 文字を表示するには、文字に対応するドットマトリックスを見つける必要があります。Linux カーネルソースコードの`kernel/lib/fonts/`には、さまざまなサイズの ASCII 文字の配列テーブルがあり、それぞれの文字のドットマトリックスが配列形式で保存されています。

株式会社日昇テクノロジー

```
1 root@ubuntu_135:~/rk/kernel/lib/fonts# tree
2 .
3 ├── font_10x18.c
4 ├── font_6x10.c
5 ├── font_6x11.c
6 ├── font_7x14.c
7 ├── font_8x16.c
8 ├── font_8x8.c
9 ├── font_acorn_8x8.c
10 ├── font_mini_4x6.c
11 ├── font_pearl_8x8.c
12 ├── fonts.c
13 ├── font_sun12x22.c
14 ├── font_sun8x16.c
15 ├── Kconfig
16 └── Makefile
```

font_8x16.c を開く

```
1 #include "font.h"
2
3 const unsigned char fontdata_8x16[FONTDATAMAX] = {
4 /* 0 0x00 '^@' */
5 0x00, /* 00000000 */
```

株式会社日昇テクノロジー

```
6 0x00, /* 00000000 */

7 0x00, /* 00000000 */

8 0x00, /* 00000000 */

9 0x00, /* 00000000 */

10 0x00, /* 00000000 */

11 0x00, /* 00000000 */

12 0x00, /* 00000000 */

13 0x00, /* 00000000 */

14 0x00, /* 00000000 */

15 0x00, /* 00000000 */

16 0x00, /* 00000000 */

17 0x00, /* 00000000 */

18 0x00, /* 00000000 */

19 0x00, /* 00000000 */

20 0x00, /* 00000000 */

21

22 /* 1 0x01 '^A' */

23 0x00, /* 00000000 */

24 0x00, /* 00000000 */

25 }
```

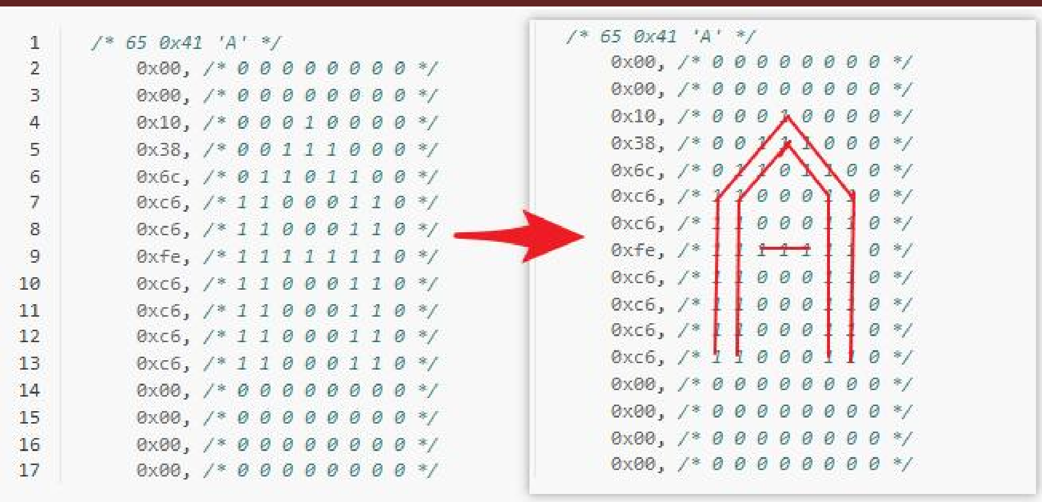
文字は配列によって構成されており、1 つの文字は 16 バイトで構成されます。例えば、「A」は 65 番目に位置します。後ろの文字はスペースキーで区切られます。各バイトが文字の一行をリストするため、画面上に文字を表示する方法は明らかです。

株式会社日昇テクノロジー

```
1 /* 65 0x41 'A' */
2 0x00, /* 00000000 */
3 0x00, /* 00000000 */
4 0x10, /* 00010000 */
5 0x38, /* 00111000 */
6 0x6c, /* 01101100 */
7 0xc6, /* 11000110 */
8 0xc6, /* 11000110 */
9 0xfe, /* 11111110 */
10 0xc6, /* 11000110 */
11 0xc6, /* 11000110 */
12 0xc6, /* 11000110 */
13 0xc6, /* 11000110 */
14 0x00, /* 00000000 */
15 0x00, /* 00000000 */
16 0x00, /* 00000000 */
17 0x00, /* 00000000 */
```

キャラクターの後ろの文字をスペースキーで区切ります。

株式会社日昇テクノロジー



各バイトがキャラクターの一行をリストしているので、画面上にキャラクターを表示する方法が非常に明確になります。

1. 文字の一行をループで取得します。

2. 各ビットを判断し、1 であれば点を描画します。

3. 16 行をループすることで、文字を完全に表示できます。

32.1.2 実験

この実験は、drm 章の最後のコンテンツ「project」を基に変更を加えて行います。framebuffer も同様に使用できます。

リスト 1: コードの場所

```
1 lubancat/base_linux/screen/char/font
```

```
1 ファイル構造
2 .
3 |-- Makefile # コンパイル
4 |-- includes
5 |   |-- drm-core.h #drm ヘッダーファイル
6 |   `-- font.h # フォントヘッダーファイル
```

株式会社日昇テクノロジー

```
7 `-- sources
8    |-- main.c #main 関数
9    |-- drm-core.c #drm の初期化及び呼び出し
10   `-- font.c # 文字配列ファイル
11
12 2 directories, 6 files
13
14 # コンパイル
15 make
16
17 # 実行
18 ./test
19
20 # 実験の現象
21 1. 画面が深い青色になる
22 2. 画面に 4 行の ASCII 文字が表示される
```

株式会社日昇テクノロジー

32.1.3 プログラム分析

リスト 2: font/sources/main.c

```
1 #include "drm-core.h"

2 #include "font.h"

3 uint32_t color_table[6] = {RED, GREEN, BLUE, BLACK, WHITE, BLACK_BLUE};

4

5 //描画関数

6 void show_pixel(uint32_t x, uint32_t y, uint32_t color)

7 {

8     if(x > buf.width || y > buf.height){

9         printf("wrong set¥n");

10     }

11

12     buf.vaddr[y*buf.width + x] = color;

13 }

14

15

16 //単一の 8x16 文字の描画

17 void show_8x16(uint32_t x, uint32_t y, uint32_t color, unsigned char num)

18 {

19     int i, j;

20     unsigned char dot;
```

株式会社日昇テクノロジー

```
21     for(i = 0; i < 16; i++){
22         dot = fontdata_8x16[num*16+i];
23         for(j = 0; j < 8; j++){
24             if(dot & 0x80)
25                 show_pixel(x+j, y+i, color);
26             dot = dot << 1;
27         }
28     }
29 }
30
31
32 //256 個の ASCII 文字を表示
33 void show_string(uint32_t color)
34 {
35     int i, j;
36     int row = 64;
37     int x_offset = (buf.width - 64*8) / 2;
38     int y_offset = (buf.height - 16*4) / 2;
39     for(j = 0; j < 4; j++){
40         for(i = 0; i < 64; i++){
41             show_8x16(i*8+x_offset, 16*j+y_offset, color, i+j*64);
42         }
```

株式会社日昇テクノロジー

```
43    }
44 }
45
46 int main(int argc, char **argv)
47 {
48    int i;
49    //画面の初期化
50    drm_init();
51    //画面の色を変更---ライトブルー
52    for(i = 0; i < buf.width*buf.height; i++)
53        buf.vaddr[i] = BLACK_BLUE;
54    //画面の中央に文字を表示
55    show_string(WHITE);
56    //文字を取得--Enter キーで次のステップへ
57    getchar();
58    drm_exit();
59
60    return 0;
61 }
62
```

- 第 5-13 行、描画関数

- 第 16-29 行、単一文字の表示。文字の点を framebuffer に上書きすることで、1 行を読み取り、最

株式会社日昇テクノロジー

上位ビットと比較し、8 回ループして 1 バイトの比較を完了し、16 バイトで 1 文字のスキャンを 16 回ループして完了します。

- 第 33-44 行、256 個の ASCII 文字を画面中央に 4 行に分けて表示。

- 第 46-59 行、drm を初期化し、背景色を設定し、256 個の文字を表示してから drm を終了します。frambuffer の操作は、drm_init と drm_exit を対応する関数に置き換えるだけで、同様の効果を達成できます。

32.2 漢字の表示（8x16）

32.2.1 実装方法

実装方法は上述の文字の実装と似ていますが、単一のバイトで全ての漢字をリスト現することはできません。そのため、漢字のエンコーディングには、GBK2312 や Unicode のような他の文字セットが必要です。

- **Unicode** — 文字セットとエンコーディングスキームを含むコンピュータ科学分野の業界標準です。Unicode は、各言語の各文字に一意で統一された二進コードを割り当てることで、言語やプラットフォームを越えたテキストの変換や処理を可能にします。

- **UTF-8** — Unicode のための可変長文字エンコーディングです。Unicode 標準の任意の文字をリスト現でき、エンコーディングの最初のバイトは ASCII と互換性があります。インターネットやコンピュータでは、通信に UTF-8 エンコーディングが使用されています。

32.2.2 実験

この実験は、前のセクションに基づいて変更され、framebuffer も同様に使用できます。

リスト 3: コードの場所

```
1 lubancat/base_linux/screen/char/chinese_16x16
```

株式会社日昇テクノロジー

```
1 # ファイル構造
2 .
3 |-- Makefile # コンパイル
4 |-- includes
5 | |-- drm-core.h # DRM ヘッダーファイル
6 | `-- font.h # フォントヘッダーファイル
7 `-- sources
8 |-- main.c # main 関数
9 |-- drm-core.c # DRM の初期化及び呼び出し
10 `-- font.c # 文字配列ファイル
11
12 2 ディレクトリ, 6 ファイル
13
14 # コンパイル
15 make
16
17 # 実行
18 ./test
19
20 # 実験現象
21 1. 画面が濃い青色になる
22 2. 画面に ASCII 文字の 4 行が表示される
```

株式会社日昇テクノロジー

32.2.3 プログラム分析

上記の実験との違いを示す部分です。

リスト 4: show_japanese

```
1
2 void show_japanese(int x, int y, unsigned char *str)
3 {
4 unsigned int area = str[0] - 0xA1;
5 unsigned int where = str[1] - 0xA1;
6 unsigned char *dots = hzkmem + (area * 94 + where)*32;
7 unsigned char byte;
8
9 int i, j, b;
10 for (i = 0; i < 16; i++){
11 for (j = 0; j < 2; j++){
12 byte = dots[i*2 + j];
13 for (b = 7; b >=0; b--){
14 if (byte & (1<<b))
15 show_pixel(x+j*8+7-b, y+i, WHITE);
16 }
17 }
18 }
19 }
```

株式会社日昇テクノロジー

- 第 2-21 行: `show_japanese()`関数は、2 バイトのエンコードされた配列を受け取り、デコードして文字のドットマトリックスを取得し、表示します。

- 第 5-6 行: エリアコードとエリア内コードを取得します。コードから見ると、漢字のエリアコードは 0xA1-0xA9 および 0xB0-0xF7、エリア内コードは 0xA1-0xFE です。しかし、HZK16 ファイルでは、関連のない ASCII コードや漢字のコードを削除して、エリアコードとエリア内コードで漢字を索引するため、0xA1 を引いて HZK16 の漢字配列番号を取得します。

- 第 7 行: 必要な漢字コードのファイル内位置を計算します。`hzkmem`は、HZK16 ファイルの先頭アドレスを指すアドレスで、オフセットは(エリアコード * 94 + エリア内コード) * 32 です(32 は 1 つの漢字が 32 バイトでリストされるため)。

リスト 5: main 関数

```
1 #include "drm-core.h"

2 #include "font.h"

3

4 int fd_hzk16;

5 struct stat hzk_stat;

6 unsigned char *hzkmem;

7 int main(int argc, char **argv)

8 {

9 int i;

10

11 unsigned char YHKJ[8] = {0XD2,0XB0,0XBB,0XF0,0XBF,0XC6,0XBC,0XBC};

12 //公式サイト
```

株式会社日昇テクノロジー

```
13 unsigned char *web = "www.embedfire.com";
14 //初期化
15 drm_init();
16 //画面の色を変更---浅い青
17 for(i = 0; i < buf.width * buf.height; i++)
18 buf.vaddr[i] = BLACK_BLUE;
19 //漢字フォントファイルを開く
20 fd_hzk16 = open("file/HZK16", O_RDONLY);
21 if (fd_hzk16 < 0){
22 printf("can't open HZK16¥n");
23 return -1;
24 }
25 //ファイル長を取得
26 if(fstat(fd_hzk16, &hzk_stat)){
27 printf("can't get fstat¥n");
28 return -1;
29 }
30 //ファイルをメモリにマッピング
31 hzkmem = (unsigned char *)mmap(NULL, hzk_stat.st_size, PROT_READ, MAP_SHARED, fd_hzk16, 0);
32 if (hzkmem == (unsigned char *)-1){
```

株式会社日昇テクノロジー

```
33 printf("can't mmap for hzk16¥n");
34 return -1;
35 }
36 //255 個の文字を表示
37 show_string(WHITE);
38
39 for(i = 0; i < 4; i++)
40 show_chinese(330 + i * 16, 1000, YHKJ + i * 2);
41 //ウェブサイト"www.embedfire.com"を表示
42 for(i = 0; i < strlen(web); i++){
43 show_8x16(290 + i * 8, 1025, RED, web[i]);
44 }
45 //キー入力を待つ
46 getchar();
47 //drm を終了
48 drm_exit();
49
50 return 0;
51 }
```

株式会社日昇テクノロジー

- 第 20-24 行: HZK16 ファイルを開き、ファイル ID を取得します。

- 第 25-29 行: HZK16 ファイルのサイズを取得し、ファイル全体をメモリにマップします。

- 第 30-35 行: ファイル全体をメモリにマップして、メモリから各漢字の配列を読み取ります。

- 第 38-40 行: 中文を表示します。表示する中文は GBK2312 エンコードされた漢字配列を定義する必

要があります。

32.3 ベクターフォント(FreeType)

これまでの文字表示方法はいずれも少し原始的で、文字の配列を番号で検索し、その配列を使って点を打つ関数で描画していました。さらに、表示される文字のサイズは固定で、異なるサイズの文字を使用したい場合は、異なるライブラリを切り替える必要がありました。このような方法は開発に不便です。

ベクターフォントは、サイズを自由に設定でき、回転なども設定できるため、非常に便利です。ほとんどの字形ファイル(.ttc)は Unicode 形式でエンコードされており、ボードで使用するのに便利です。

これは FreeType の公式ウェブサイトで、詳細な紹介や例があります。以下の説明は FreeType の氷山の一角かもしれませんが、日常使用には十分です。さらに学びたい場合は、公式ウェブサイトでさらに学ぶことができます。

アドレス：https://freetype.org/

32.3.1 実装方法

ベクターフォントは、字形ファイルから字形構造を取得し、設定された字形のサイズ、回転角などに基づいて、レンダリング方法で文字の点配列を取得します。そのため、字形ファイルを変更することで、異なるスタイルの文字を表示できます。

ベクターフォントの生成は 3 ステップで行います：

1. 字形の確定（字形ファイルによって提供される）
2. 数学曲線（ベジェ曲線）を使用して頭のキーポイントを接続する（レンダリング）
3. 閉じた領域の内部空間を塗りつぶす（レンダリング）

FreeType はオープンソースのフォントエンジンライブラリで、多種多様なフォントフォーマットファイルに統一されたインターフェースを提供し、ベクターフォント表示を実現します。対応する API インターフェースを呼び出し、フォントファイルを提供するだけで、FreeType ライブラリがキーポイントの取り出

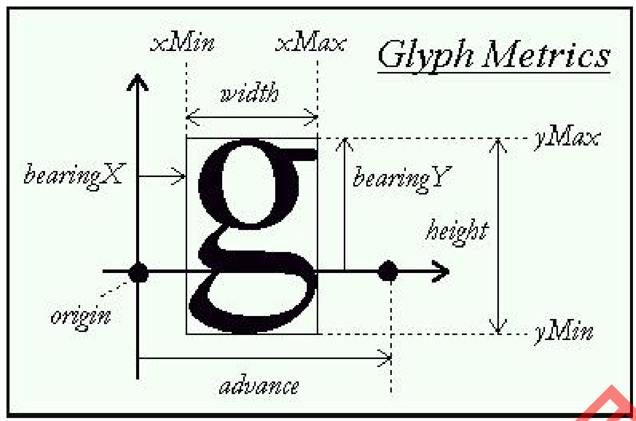
株式会社日昇テクノロジー

し、閉曲線の実現、色の塗りつぶしを行い、ベクターフォントの表示を実現できます。

後の実験では、simsun.ttc 新宋フォントファイルを使用します。このファイルは Unicode で字形を検索することをサポートしており、utf-8 から Unicode への変換を直接使用できます。

32.3.1.1 ベクターテキスト

Freetype のベクターテキストには以下のパラメータがあります（ここでは横向きを例に、縦向きは公式ウェブサイトで確認できます）：



- **origin** : 文字の形の基点で、すべての文字形状ファイルのレンダリングは基点に基づいて行われます。文字形状の構造体内のすべてのデータは、原点のデータに基づいてオフセットされます。定義しない場合、原点は(0,0)です。

- **advance** : 「font_size」に基づいて調整可能です。

- **bearingX** : 原点から表示エリア（レンダリングエリア）の最初の列への x 軸のオフセット（slot->bitmap_left に対応）。

- **bearingY** : 原点から表示エリア（レンダリングエリア）の最初の行への y 軸のオフセット（slot->bitmap_top に対応）。

- **width** : 表示エリア（レンダリングエリア）の一行の長さ（bitmap->width に対応）。

- **height** : 表示エリア（レンダリングエリア）の高さ（bitmap->rows に対応）。

- **xMin** : 文字列ボックスを計算するために使用できます。文字形状構造体内の値は原点からのオフ

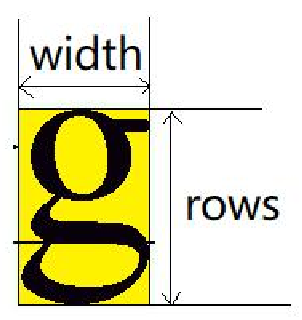
株式会社日昇テクノロジー

セットです。

- **xMax** : 文字列ボックスを計算するために使用できます。文字形状構造体内の値は原点からのオフセットです。

- **yMin** : 文字列ボックスを計算するために使用できます。文字形状構造体内の値は原点からのオフセットです。

- **yMax** : 文字列ボックスを計算するために使用できます。文字形状構造体内の値は原点からのオフセットです。



レンダリングされたテキストメモリ

・上の図はレンダリングされたバッファ領域であり、これらのバッファ領域を利用して画面表示を行います

32.3.2 単一のベクターフォントの表示

この実験は Freetype のチュートリアルと drm 章のプロジェクト実験を基にしています。Freetype についての詳細は以下のウェブサイトで学ぶことができます。

https://freetype.org/freetype2/docs/design/index.html

株式会社日昇テクノロジー

リスト 6: コードの場所

```
1 lubancat/base_linux/screen/char/freetype/
```

```
1 ファイル構成:
2 .
3 |-- Makefile
4 |-- file
5 |   `-- simsun.ttc
6 |-- includes
7 |   |-- drm-core.h
8 |   `-- font.h
9 `-- sources
10    |-- drm-core.c
11    |-- font.c
12    `-- main.c # main 関数
```

この実験では Freetype ライブラリを使用するため、コンパイル時にライブラリを含める必要があります。LubanCat-ubuntu システムにはこれらのライブラリがすでに含まれているため、コンパイル時に直接使用できます。Makefile ファイルにはライブラリコンパイルオプションがすでに含まれています。

今回の実験では freetype ライブラリを使用するため、コンパイル時にライブラリを追加する必要があります。LubanCatubuntu システムにはこれらのライブラリが既に含まれているため、コンパイル時にそのまま適用できます。Makefile ファイルにはライブラリのコンパイルオプションが既に追加されており、そのまま使用できます。

```
1. GCC の後ろに`pkg-config --cflags freetype2` `pkg-config --libs freetype2`を追加します。
```

株式会社日昇テクノロジー

今回の実験では math ライブラリも使用するため、コンパイル時にライブラリを追加する必要があります。

1. GCC の後ろに`-lm`を追加します。

32.3.2.1 コンパイル&実行

```
1 # コンパイル
2 make
3
4 # 実行
5 ./test
6
7 # 実験現象
8
```

32.3.2.2 プログラム分析

リスト 7: utf_8 to unicode

```
1 int utf_8_to_unicode_string(uint8_t *utf_8, uint16_t *word)
2 {
3     int len = 0;
4     int utf_8_size = strlen(utf_8);
5     int utf_8_len = 0;
6     uint16_t unicode[2];
7
8     while(utf_8_size > 0){
9         // 1 バイトの utf_8 を 2 バイトの unicode に変換
```

株式会社日昇テクノロジー

```
10      if(utf_8[utf_8_len] < 0x80){
11          unicode[0] = 0;
12          unicode[1] = utf_8[utf_8_len];
13          word[len] = (unicode[0] << 8) | unicode[1];
14          len++;
15          utf_8_len++;
16          utf_8_size--;
17          continue;
18      }
19      // 2 バイトの utf_8 を 2 バイトの unicode に変換
20      else if(utf_8[utf_8_len] > 0xc0 & utf_8[utf_8_len] < 0xe0){
21          unicode[1] = (utf_8[utf_8_len+1] & 0x3f) | ((utf_8[utf_8_len] << 6) & 0xc0);
22          unicode[0] = ((utf_8[utf_8_len] >> 2) & 0x07);
23          word[len] = (unicode[0] << 8) | unicode[1];
24          len++;
25          utf_8_len += 2;
26          utf_8_size -= 2;
27          continue;
28      }
29      // 3 バイトの utf_8 を 2 バイトの unicode に変換
30      else if(utf_8[utf_8_len] > 0xe0 & utf_8[utf_8_len] < 0xf0){
31          unicode[1] = (utf_8[utf_8_len+2] & 0x3f) | ((utf_8[utf_8_len+1] << 6) & 0xc0);
```

株式会社日昇テクノロジー

```
32          unicode[0] = ((utf_8[utf_8_len+1] >> 2) & 0x0f) | ((utf_8[utf_8_len] << 4) & 0xf0);
33          word[len] = (unicode[0] << 8) | unicode[1];
34          len++;
35          utf_8_len += 3;
36          utf_8_size -= 3;
37          continue;
38      }
39      // 4 バイトの utf_8 を unicode に変換する場合は、操作が複雑になるため、
40      // 基本的には 3 バイトの utf_8 でリスト現できる中国語に対応し、4 バイト以上のデコードは行
いません
41      else
42          return -1;
43  }
44  return len;
45 }
```

- この関数は、3 バイト以内の utf-8 文字列を 2 バイトの Unicode 文字列に変換するためのものです。変換の原理は、参考資料を参照してください。

UCS-4 (UNICODE) 编码	UTF-8字节流
U-00000000 – U-0000007F	0xxxxxxx
U-00000080 – U-000007FF	110xxxxx 10xxxxxx
U-00000800 – U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx

株式会社日昇テクノロジー

リスト 8: main 関数

```
1 #include "drm-core.h"

2 #include <ft2build.h>

3 #include <math.h>

4 #include FT_FREETYPE_H

5 #include FT_GLYPH_H

6

7 uint32_t color_table[6] = {RED,GREEN,BLUE,BLACK,WHITE,BLACK_BLUE};

8

9 //ピクセル表示

10 void show_pixel(uint32_t x , uint32_t y , uint32_t color)

11 {

12 if(x > buf.width || y > buf.height)

13 printf("設定エラー¥n");

14 buf.vaddr[ y*buf.width + x] = color;

15 }

16

17 int utf_8_to_unicode_string(uint8_t *utf_8,uint16_t *word)

18 {

19 int len = 0;

20 int utf_8_size = strlen(utf_8);

21 int utf_8_len = 0;
```

株式会社日昇テクノロジー

```
22 uint16_t unicode[2];

23

24 while(utf_8_size > 0){

25 //1 バイトの utf_8 を 2 バイトの unicode に変換

26 if(utf_8[utf_8_len] < 0x80){

27 unicode[0] = 0;

28 unicode[1] = utf_8[utf_8_len];

29 word[len] = (unicode[0]<<8) | unicode[1];

30 len ++;

31 utf_8_len ++;

32 utf_8_size--;

33 continue;

34 }

35 //2 バイトの utf_8 を 2 バイトの unicode に変換

36 else if(utf_8[utf_8_len] > 0xc0 & utf_8[utf_8_len] <0xe0){

37 unicode[1] = (utf_8[utf_8_len+1]&0x3f) | ((utf_8[utf_8_len]<< 6)& 0xc0 );

38 unicode[0] = ((utf_8[utf_8_len]>>2) & 0x07) ;

39 word[len] = (unicode[0]<<8) | unicode[1];

40 len ++;

41 utf_8_len +=2;

42 utf_8_size -=2;

43 continue;
```

株式会社日昇テクノロジー

```
44 }
45 //3 バイトの utf_8 を 2 バイトの unicode に変換
46 else if(utf_8[utf_8_len] > 0xe0 & utf_8[utf_8_len]<0xf0){
47 unicode[1] = (utf_8[utf_8_len+2]&0x3f) | ((utf_8[utf_8_len+1] << 6)& 0xc0);
48 unicode[0] = ((utf_8[utf_8_len+1]>>2)&0x0f) | ((utf_8[utf_8_len] <<4)& 0xf0) ;
49 word[len] = (unicode[0]<<8) | unicode[1];
50 len ++;
51 utf_8_len +=3;
52 utf_8_size -=3;
53 continue;
54 }
55 //4 バイトの utf_8 を 3~4 バイトの unicode に変換する必要があり、操作が不便なため、
56
57 else
58 return -1;
59 }
60 return len;
61 }
62
63 void draw_bitmap( FT_Bitmap* bitmap,FT_Int x_ori,FT_Int y_ori)
64 {
65 FT_Int x, y;
```

株式会社日昇テクノロジー

```
66 FT_Int x_count, y_count;

67 unsigned char show;

68 uint32_t buffer_size = bitmap->width * bitmap->rows;

69 uint8_t buffer[buffer_size];

70 uint32_t color;

71 FT_Int x_max = x_ori + bitmap->width;

72 FT_Int y_max = y_ori + bitmap->rows;

73

74 memcpy(buffer,bitmap->buffer,buffer_size);

75

76 for ( y = y_ori, y_count = 0; y < y_max; y++, y_count++ ){

77 for ( x = x_ori, x_count = 0; x < x_max; x++, x_count++ ){

78

79 if ( x < 0 || y < 0 || x >= buf.width || y >= buf.height )

80 continue;

81 //buf 内の画像は 8 ビットの階調値で、色に変換する必要がある、そうでなければ青色になる

82 show = buffer[y_count * bitmap->width + x_count];

83 //階調が 0 より大きい場合、同じ強度の白色に変換

84 if(show > 0)

85 color = (show&0xff)|((show&0xff)<<8)|((show&0xff)<<16);

86 //黒色にすることもできるが省略可能

87 else
```

株式会社日昇テクノロジー

```
88 color=0;

89 //ピクセル表示関数

90 show_pixel(x, y , color);

91 }

92 }

93

94 }

95

96

97 int freetype_set_char(FT_Face face , int lcd_x,int lcd_y ,int font_size,int angle_degree,uint16_t word)

98 {

99 FT_Matrix matrix;

100 FT_Vector pen;

101 int charIdx;

102 FT_GlyphSlot slot= face->glyph;;

103 double angle;

104 int error;

105

106 FT_Set_Pixel_Sizes(face, font_size, 0);

107 //回転設定

108 angle = (1.0 * angle_degree/360) * 3.14159 * 2;

109 /* マトリックス設定 */
```

株式会社日昇テクノロジー

```
110 matrix.xx = (FT_Fixed)( cos( angle ) * 0x10000L );

111 matrix.xy = (FT_Fixed)(-sin( angle ) * 0x10000L );

112 matrix.yx = (FT_Fixed)( sin( angle ) * 0x10000L );

113 matrix.yy = (FT_Fixed)( cos( angle ) * 0x10000L );

114

115 /* 変換：transformation */

116 FT_Set_Transform(face, &matrix, 0);

117

118 /* 以下の 3 行は FT_Load_Char の代わりに使用可能 */

119 // charIdx = FT_Get_Char_Index(face,word);

120 // FT_Load_Glyph(face,charIdx, FT_LOAD_DEFAULT);

121 // FT_Render_Glyph(face->glyph, FT_RENDER_MODE_NORMAL);

122

123 error = FT_Load_Char( face, word, FT_LOAD_RENDER );

124 if (error){

125 printf("FT_Load_Char エラー¥n");

126 return -1;

127 }

128 draw_bitmap( &slot->bitmap,lcd_x, lcd_y);

129 return 0;

130 }

131
```

株式会社日昇テクノロジー

```
132 int main(int argc, char **argv)

133 {

134 int i,j,count;

135 FT_Library library;

136 FT_Face face;

137 int error;

138 int font_size = 180;

139 int angle_degree = 0;

140

141 unsigned char str1[] = "CSUN A";

142 unsigned char str2[] = "www.embedfire.com";

143 unsigned char str3[] = "abcdefghijklmnopq";

144 uint16_t unicode[20];

145 int unicode_size = 0;

146

147 drm_init();

148

149 /* ベクターフォント表示 */

150 error = FT_Init_FreeType( &library ); /* ライブラリ初期化 */

151 /* エラー処理は省略 */

152 error = FT_New_Face( library, "file/simsun.ttc", 0, &face ); /* フェイスオブジェクト作成 */

153 unicode_size = utf_8_to_unicode_string(str1,unicode);
```

株式会社日昇テクノロジー

```
154

155

156 for(i = 0 ; i<unicode_size;i++)

157 freetype_set_char(face,0+font_size*i,640,font_size,angle_degree,unicode[i] );

158

159 // 公式ウェブサイト表示

160 unicode_size = utf_8_to_unicode_string(str2,unicode);

161 for(i = 0 ; i<unicode_size;i++)

162 freetype_set_char(face,0+font_size/6*i,820,font_size/3,angle_degree,unicode[i] );

163

164 getchar();

165 drm_exit();

166

167 return 0;

168 }
```

- 第 23 行、Freetype の初期化

- 第 24-25 行、字形ファイルを開き、face を作成

- 第 26、33、38 行、それぞれ UTF-8 でエンコードされた str1、str2、str3 を 2 バイトの Unicode エンコーディングの配列に変換

- 第 29-30、34-35、39-40 行、サイズを設定し、ベクトルテキストをループで印刷

株式会社日昇テクノロジー

リスト 9: freetype_set_char

```
1 int freetype_set_char(FT_Face face, int lcd_x, int lcd_y, int font_size, int angle_degree, uint16_t word)
2 {
3    FT_Matrix matrix; // 変換行列
4    FT_Vector pen; // ペンの位置
5    int charIdx;
6    FT_GlyphSlot slot = face->glyph; // グリフスロット
7    double angle;
8    int error;
9
10    FT_Set_Pixel_Sizes(face, font_size, 0); // フォントサイズの設定
11    // 回転の設定
12    angle = (1.0 * angle_degree / 360) * 3.14159 * 2;
13    // 変換行列の設定
14    matrix.xx = (FT_Fixed)(cos(angle) * 0x10000L);
15    matrix.xy = (FT_Fixed)(-sin(angle) * 0x10000L);
16    matrix.yx = (FT_Fixed)(sin(angle) * 0x10000L);
17    matrix.yy = (FT_Fixed)(cos(angle) * 0x10000L);
18
19    // 変換の適用
20    FT_Set_Transform(face, &matrix, 0);
21
```

株式会社日昇テクノロジー

```
22    // 以下のコードは FT_Load_Char を使用する代わりになります
23    // charIdx = FT_Get_Char_Index(face, word);
24    // FT_Load_Glyph(face, charIdx, FT_LOAD_DEFAULT);
25    // FT_Render_Glyph(face->glyph, FT_RENDER_MODE_NORMAL);
26
27    error = FT_Load_Char(face, word, FT_LOAD_RENDER); // 文字の読み込みとレンダリング
28    if (error) {
29        printf("FT_Load_Char error¥n");
30        return -1;
31    }
32    draw_bitmap(&slot->bitmap, lcd_x, lcd_y); // ビットマップの描画
33    return 0;
34 }
```

- 第 10 行、フォントサイズを設定

- 第 11-20 行、回転を設定し、angle_degree を変更することで回転角度を変えることができます。これらの行のコードは設定不要で実行可能ですが、説明のために追加されています。

- 第 27 行、文字を読み込み、レンダリングします。この関数は第 23-25 行の機能を含み、文字リストの読み込み、グリフデータのロード、グリフデータのレンダリングを行います。

- 第 32 行、取得したグリフファイルを描画します。

株式会社日昇テクノロジー

リスト 10: draw_bitmap

```
1 void draw_bitmap(FT_Bitmap* bitmap, FT_Int x_ori, FT_Int y_ori)
2 {
3     FT_Int x, y;
4     FT_Int x_count, y_count;
5     unsigned char show;
6     uint32_t buffer_size = bitmap->width * bitmap->rows;
7     uint8_t buffer[buffer_size];
8     uint32_t color;
9     FT_Int x_max = x_ori + bitmap->width;
10    FT_Int y_max = y_ori + bitmap->rows;
11
12    memcpy(buffer, bitmap->buffer, buffer_size);
13
14    for (y = y_ori, y_count = 0; y < y_max; y++, y_count++) {
15        for (x = x_ori, x_count = 0; x < x_max; x++, x_count++) {
16
17            if (x < 0 || y < 0 || x >= buf.width || y >= buf.height)
18                continue;
19 //buf 内の画像は 8 ビットの階調値を格納しており、表示するには色に変換する必要がある、
そうでないと青色として表示される
20 show = buffer[y_count * bitmap->width + x_count];
```

株式会社日昇テクノロジー

```
21 //階調が 0 以上の場合、同じ強度の白色に変換
22 if(show > 0)
23 color = (show&0xff)|((show&0xff)<<8)|((show&0xff)<<16);
24 //直接黒色にすることもできるが、省略可能
25 else
26 color=0;
27 //ピクセル表示関数
28 show_pixel(x, y , color);
29 }
30 }
31
32 }
```

・第 6 行、レンダリングされた buffer が占めるメモリのサイズを取得

・第 7 行、同じサイズのメモリスペースを作成して格納するため

・第 9-10 行、x_ori と y_ori をメモリの起点とした x 軸と y 軸の最大値を取得

・第 12 行、buffer を作成したばかりのメモリにコピー

・第 14-32 行、buffer 内の点をループして表示

・第 21-26 行、freetype で生成された文字形のメモリには色成分がなく、8 ビットの強度階調のみであり、文字の端が暗く、文字の密集した部分が最大値になるため、他の色が必要な場合は自分で置換する必要がある、

または変更せずに点描関数に直接渡す（青色として表示される）

株式会社日昇テクノロジー

32.3.2.3 総括

Freetype を使用してベクターテキストを生成する手順は以下の通りです。

1. 生成したいベクターテキストのコード（GBK、Unicode など）を取得します。
2. Freetype を初期化します。
3. Freetype で、生成したいテキストのコードに対応するフォントファイルを開きます。
4. テキストのサイズを設定します。
5. 回転角（省略可能）と原点の位置を設定します。
6. ビットマップをロードします（3 つの関数で置き換え可能）。
7. ビットマップのバッファを使用して、描画関数でドットを描きます。

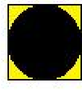
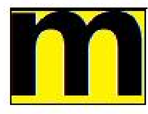
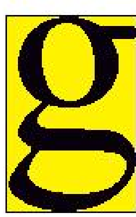
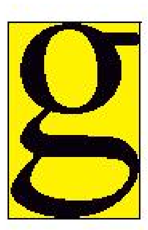
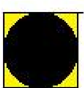
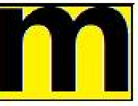
32.3.3 Freetype で文字列を表示

32.3.3.1 問題の復習

前節の実験現象で、英文字が y 軸上で位置が不正確になる現象がありました。これはなぜでしょうか？

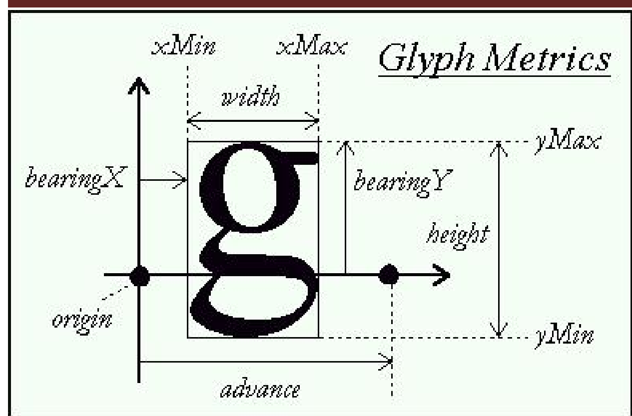
原因分析：

- 同じフォントサイズでレンダリングされたテキストのサイズは同じではありません。draw_bitmap() 関数で背景色を追加してみると（例：else { color = RED; }）、異なる文字のバッファサイズが異なり、各文字が左上隅から表示されることがわかります。

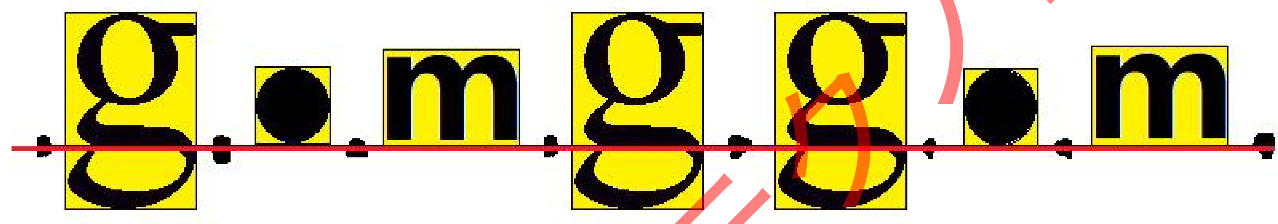
     

方法：

株式会社日昇テクノロジー



- 前述したように、原点はすべてのフォントレンダリングの基点です。つまり、各文字の原点が前の文字の（原点+advance）と一致すれば、それらを整然とさせることができます。



したがって、原点を揃える方法を見つければ、y 軸上で文字を正しく表示することができます。上記の文字形には、非常に重要なパラメータである yMax があり、その位置は原点からの相対的なオフセットです。したがって、各文字の表示エリアを yMax の高さだけ上に移動させることで、各文字形の原点を設定した y 軸に沿って揃えることができます。上に移動した後、各文字形が揃うことがわかります。では、yMax をどのようにして得るのでしょうか？原点が設定されていない場合、または原点が 0 の場合、yMax は slot->bitmap_top に相当するため、draw_bitmap 関数を変更することで、先ほど述べた効果を達成できます。

```
1 draw_bitmap( &slot->bitmap,lcd_x, lcd_y-slot->bitmap_top);
```

上に移動した後、連続する原点軸からのみフォントを生成でき、左上角からテキストの位置を確定できなくなる問題が生じますが、以下の実験ではこの問題を解決します。

株式会社日昇テクノロジー

32.3.3.2 文字列表示実験

上記の問題を解決し、文字列を正しく表示する方法を示す実験です。

リスト 11: コードの位置

```
1 lubancat/base_linux/screen/char/freetype_str/
```

リスト 12: main 関数

```
1 int main(int argc, char **argv)
2 {
3 FT_Library library;
4 FT_Face face;
5 FT_BBox bbox;
6 int i,j,error;
7 //フォントサイズ
8 int font_size = 180;
9
10 uint8_t str[] = "CSUN";
11 uint8_t str1[] = "www.embedfire.com";
12 uint16_t unicode[10];
13 uint16_t unicode1[10];
14 uint32_t u_len;
15 uint32_t u1_len;
16
17 //内容をクリア
```

株式会社日昇テクノロジー

```
18 memset(f_name.unicode,0,sizeof(f_name.unicode));

19 memset(f_str[0].unicode,0,sizeof(f_str[0].unicode));

20 memset(f_str[1].unicode,0,sizeof(f_str[1].unicode));

21

22 drm_init();

23 //ファイルを開く、Linux ではファイルは utf-8 エンコーディングで存在

24 f_name.fd = open("file/name.txt", O_RDONLY);

25 //ファイルサイズを取得

26 fstat(f_name.fd, &f_name.stat);

27 //ファイルの内容を全てマッピング

28 f_name.mem = (unsigned char *)mmap(NULL , f_name.stat.st_size, PROT_READ, MAP_SHARED,
f_name.fd, 0);

29 //utf_8 から unicode へ変換

30 f_name.len=utf_8_to_unicode_string(f_name.mem,f_name.unicode);

31 u_len = utf_8_to_unicode_string(str,unicode);

32 u1_len = utf_8_to_unicode_string(str1,unicode1);

33

34 //freetype を初期化

35 error = FT_Init_FreeType( &library );

36 //文字ファイルを読み込み、face を作成

37 error = FT_New_Face( library, "file/simsun.ttc", 0, &face );

38
```

株式会社日昇テクノロジー

```
39 //「CSUN」を表示

40 FT_Set_Pixel_Sizes(face, 80, 0);

41 compute_string_bbox(face, unicode,u_len,&bbox );

42 display_string(face, unicode,u_len, 200, 1000,&bbox);

43

44 //公式ウェブサイトを表示

45 FT_Set_Pixel_Sizes(face, font_size/3, 0);

46 compute_string_bbox(face,unicode1,u1_len,&bbox);

47 display_string(face, unicode1,u1_len ,120, 1100,&bbox);

48

49

50 //「CSUN へようこそ」を表示

51 FT_Set_Pixel_Sizes(face, 60, 0);

52 compute_string_bbox(face, f_name.unicode,f_name.len,&bbox);

53 display_string(face, f_name.unicode, f_name.len,135, 500,&bbox);

54

55 getchar();

56 drm_exit();

57

58 return 0;

59 }
```

株式会社日昇テクノロジー

- 第 17-20 行、Unicode のバッファをクリアします。

- 第 23-30 行、ファイルを開き、ファイルから UTF-8 エンコードされたテキストを取得し、それを Unicode コードに変換してバッファに渡します。

- 第 31-32 行、2 つの UTF-8 エンコードされた文字列を Unicode エンコーディングに変換します。

- 第 34-37 行、freetype を初期化します。

- FT_Set_Pixel_Sizes でテキストサイズを設定します。

- compute_string_bbox で全てのテキストの境界ボックスを取得し、後で使用するために全てのテキストを含むことができる境界ボックスレイヤーを計算します。これにより、後で平移を容易に行えます。

- display_string でテキストを表示します。

リスト 13：compute_string_bbox

```
1 int compute_string_bbox(FT_Face face,uint16_t *str,int len, FT_BBox *abbox)
2 {
3 int i;
4 int error;
5 FT_BBox bbox;
6 FT_BBox glyph_bbox;
7 FT_Vector pen;
8 FT_Glyph glyph;
9 FT_GlyphSlot slot = face->glyph;
10
11 /* 初期化 */
12 bbox.xMin = bbox.yMin = 32000;
```

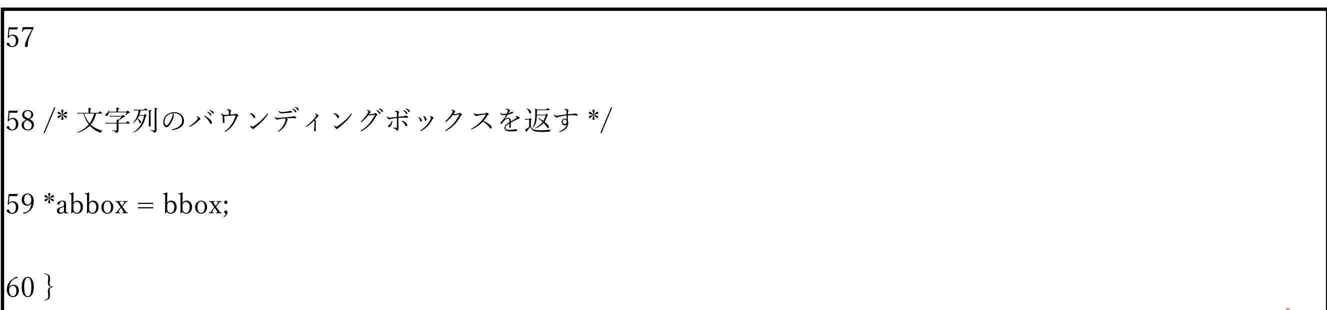
株式会社日昇テクノロジー

```
13 bbox.xMax = bbox.yMax = -32000;

14 /* 原点を(0, 0)に設定 */

15 pen.x = 0;

16 pen.y = 0;

17

18 /* 各文字のバウンディングボックスを計算 */

19 /* まず変換を行い、その後 char をロードすると、その外枠が得られる */

20 for (i = 0; i < len; i++)

21 {

22 /* 変換：transformation */

23 FT_Set_Transform(face, 0, &pen);

24 /* ビットマップをロード：スロットにグリフ画像をロード（前のを消去） */

25 error = FT_Load_Char(face, str[i], FT_LOAD_RENDER);

26 if (error){

27 printf("FT_Load_Char エラー¥n");

28 return -1;

29 }

30

31 /* グリフを取り出す */

32 error = FT_Get_Glyph(face->glyph, &glyph);

33 if (error){

34 printf("FT_Get_Glyph エラー！¥n");
```

株式会社日昇テクノロジー

```
35 return -1;

36 }

37

38 /* グリフから外枠を取得：bbox */

39 FT_Glyph_Get_CBox(glyph, FT_GLYPH_BBOX_TRUNCATE, &glyph_bbox);

40

41 /* 外枠を更新 */

42 if (glyph_bbox.xMin < bbox.xMin)

43 bbox.xMin = glyph_bbox.xMin;

44

45 if (glyph_bbox.yMin < bbox.yMin)

46 bbox.yMin = glyph_bbox.yMin;

47

48 if (glyph_bbox.xMax > bbox.xMax)

49 bbox.xMax = glyph_bbox.xMax;

50

51 if (glyph_bbox.yMax > bbox.yMax)

52 bbox.yMax = glyph_bbox.yMax;

53

54 /* 次の文字の原点を計算：ペン位置を増やす */

55 pen.x += slot->advance.x;

56 }
```

株式会社日昇テクノロジー



```
57

58 /* 文字列のバウンディングボックスを返す */

59 *abbox = bbox;

60 }
```

この関数は、すべてのフォントの境界ボックスを取得し、すべての文字形を含むことができる最小のボックスを計算するためのものです。

下図のように

ymax

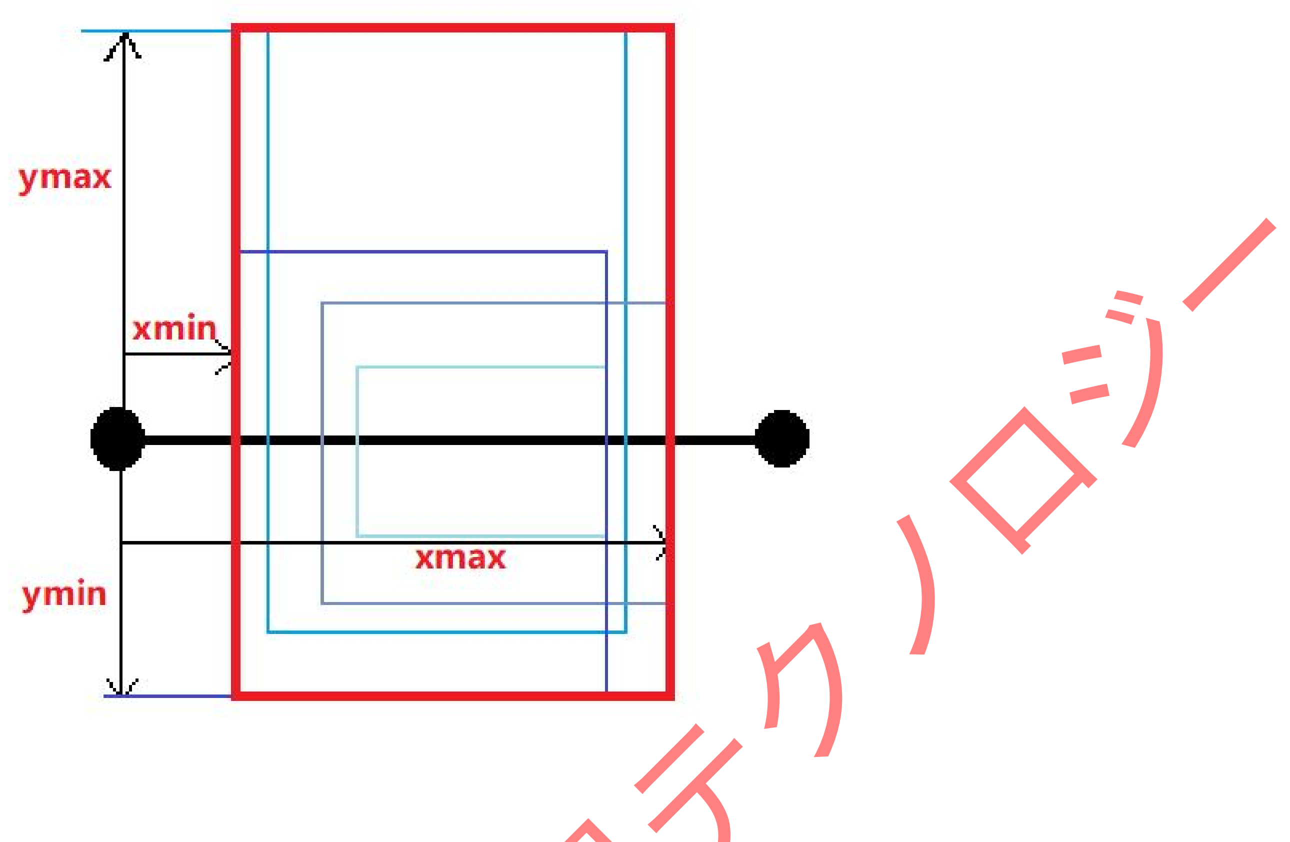
xmin

xmax

ymin

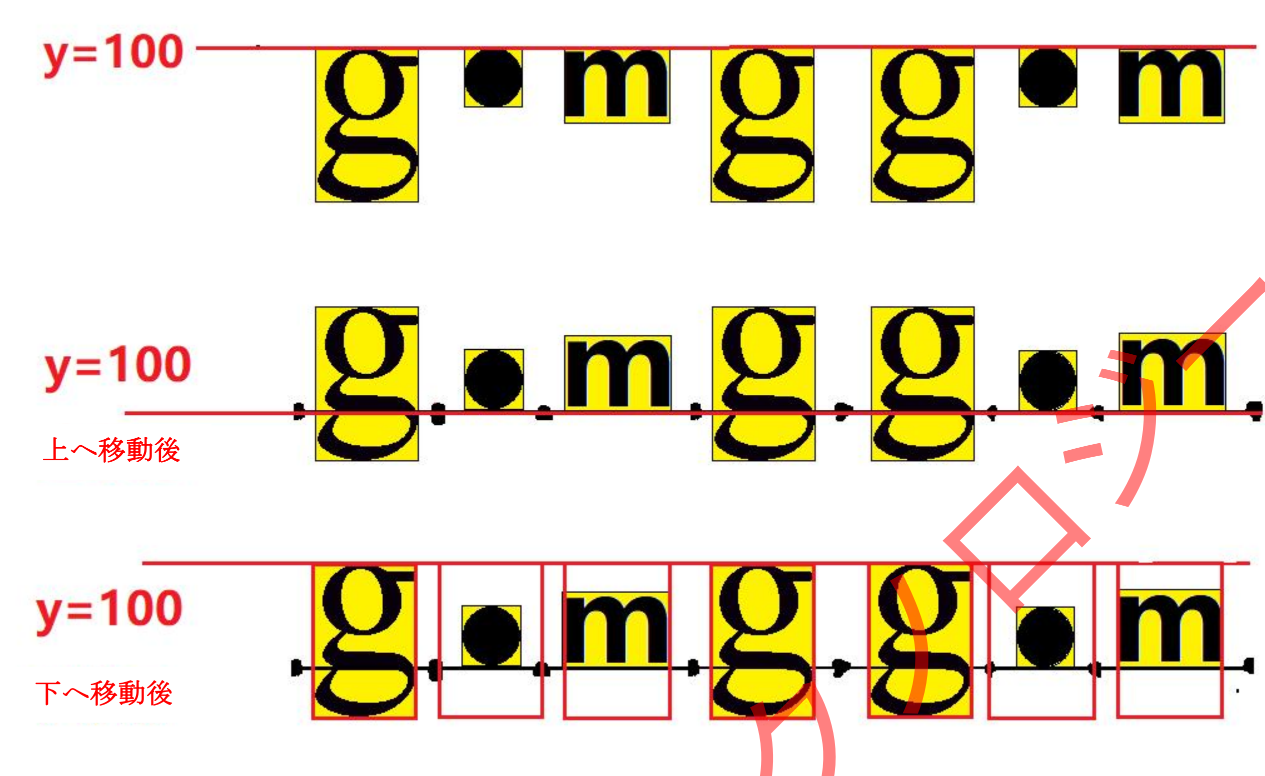
最小のボックス

株式会社日昇テクノロジー

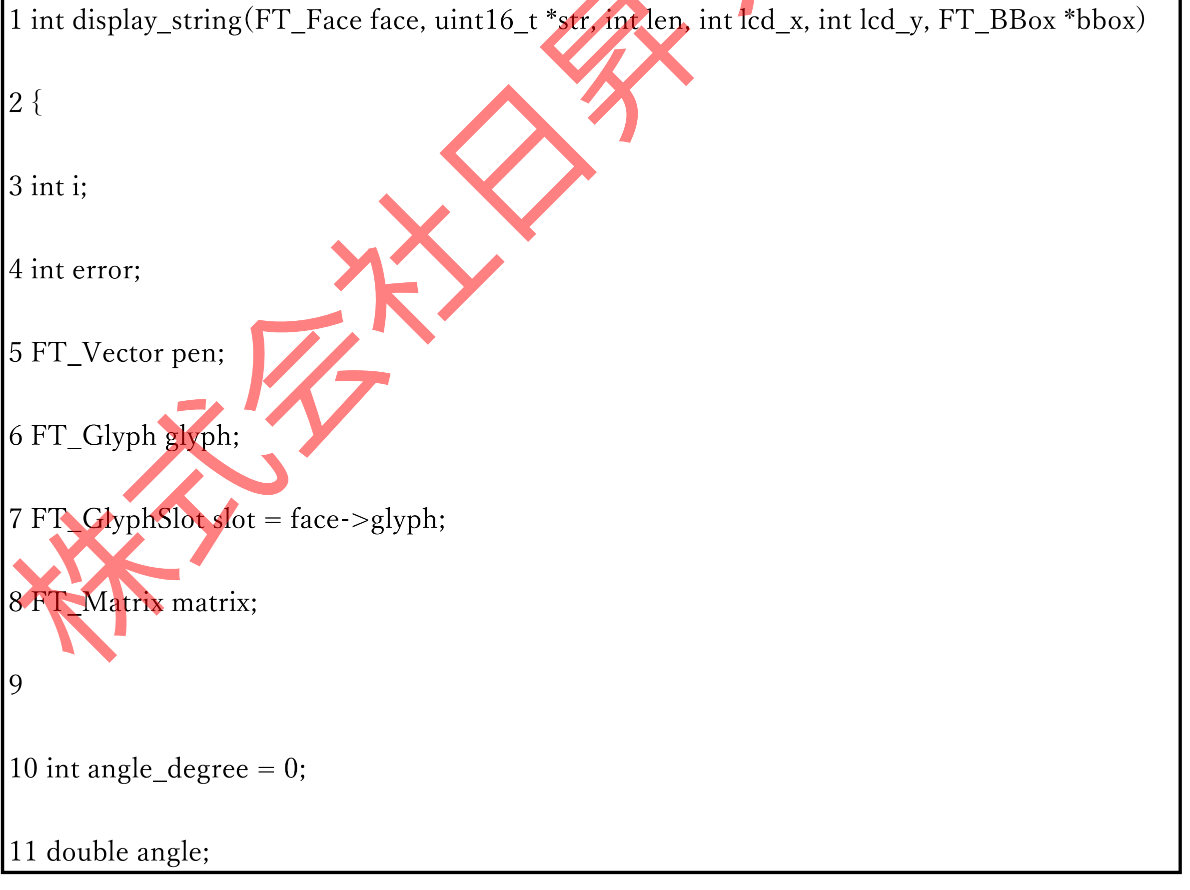


これらのボックスの機能：あるボックスが文字列内の全ての文字のサイズを満たす能力を持つ場合、各文字のボックスをこのボックスを基準として、ymax だけ下に移動します。そうすることで、左上角から文字列の位置を確定できるようになります。

株式会社日昇テクノロジー



リスト 14：display_string()



```
1 int display_string(FT_Face face, uint16_t *str, int len, int lcd_x, int lcd_y, FT_BBox *bbox)

2 {

3 int i;

4 int error;

5 FT_Vector pen;

6 FT_Glyph glyph;

7 FT_GlyphSlot slot = face->glyph;

8 FT_Matrix matrix;

9

10 int angle_degree = 0;

11 double angle;
```

株式会社日昇テクノロジー

```
12
13 /* LCD 座標をデカルト座標系に変換 */
14 int x = lcd_x;
15 int y = buf.height - lcd_y;
16 //原点を設定
17 pen.x = (x - bbox->xMin) * 64; /* 単位: 1/64 ピクセル */
18 pen.y = (y - bbox->yMax) * 64; /* 単位: 1/64 ピクセル */
19
20 /* 各文字を処理 */
21 for (i = 0; i < len; i++){
22 //回転を設定
23 angle = (1.0 * angle_degree / 360) * 3.14159 * 2;
24 /* マトリクスを設定 */
25 matrix.xx = (FT_Fixed)(cos(angle) * 0x10000L);
26 matrix.xy = (FT_Fixed)(-sin(angle) * 0x10000L);
27 matrix.yx = (FT_Fixed)(sin(angle) * 0x10000L);
28 matrix.yy = (FT_Fixed)(cos(angle) * 0x10000L);
29
30 /* 変換：transformation */
31 FT_Set_Transform(face, &matrix, &pen);
32
33 /* ビットマップを読み込み：スロットにグリフ画像をロード（前のものを消去） */
```

株式会社日昇テクノロジー

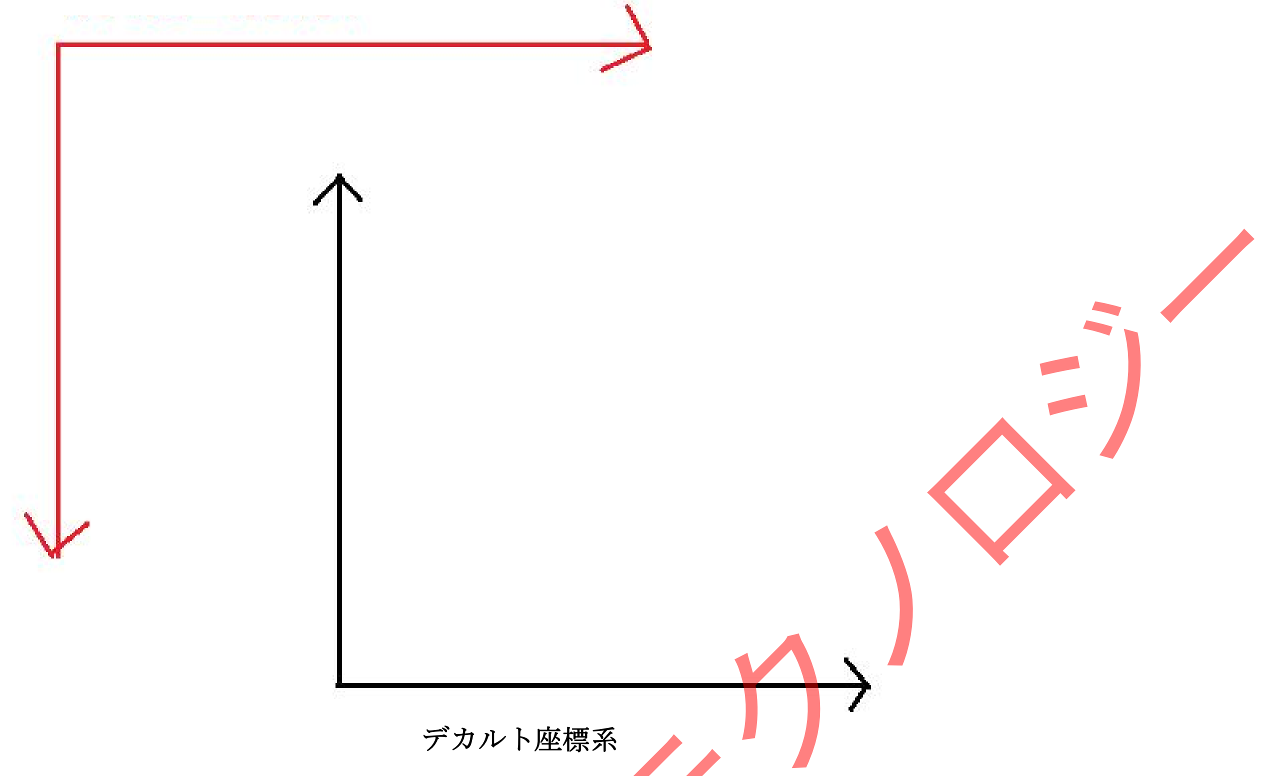
```
34 error = FT_Load_Char(face, str[i], FT_LOAD_DEFAULT | FT_LOAD_RENDER);

35

36 /* 内容を表示 */

37 draw_bitmap(&slot->bitmap, slot->bitmap_left,

38 buf.height - slot->bitmap_top);

39

40 /* 次の文字の原点を計算：ペンの位置を増やす */

41 pen.x += slot->advance.x;

42 // 縦書きの場合はこちらを使用

43 // pen.y += slot->advance.y;

44 }

45

46 return 0;

47 }
```

- この関数は、文字列を画面上に表示するためのものです。

- 第 13-15 行、画面の座標系をデカルト座標系に切り替えます。

- これは、原点の位置がデカルト座標系に基づいて構築されるためです：切り替え方は比較的簡単で、x 軸は変わらず、y 軸は画面の縦の高さから画面の座標系の y 軸を引いたものになります。

株式会社日昇テクノロジー

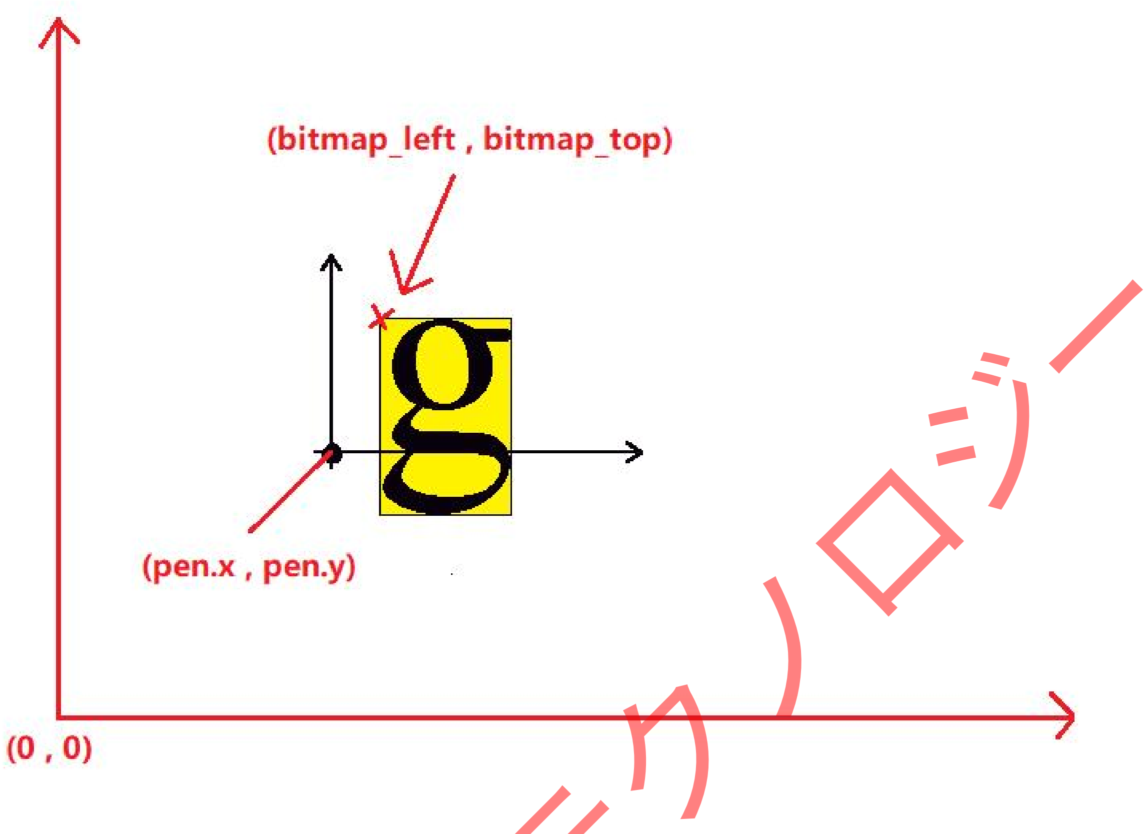


- 第 16-18 行、原点を設定します。x64 は、freetype がレンダリング時にピクセルポイントの x 軸と y 軸を 64 倍に拡大してからレンダリングするためです（そうでなければ、これほど細かいテキストを見ることはできません）。bbox->xMin と bbox->yMax は、上で計算された外枠から得られたデータで、ここでは最大の外枠です。デカルト座標系では、正数を引くと左に移動することと同じであり、下に移動することとも同じです。ここで原点を左に bbox->xMin ユニット移動させるのは、最初の文字の左側の一部の空行を消去するためで、原点を下に移動させるのは、前に述べたように、文字を y 軸の下に表示させ、y 成分以下で文字が表示されるようにするためです。

- 第 20-36 行、通常のフォント設定

- 第 37 行、デカルト座標系を画面の座標系に戻してから描画します。以下は、原点が（0、0）の場合の bitmap_left と bitmap_top の関係図です。bitmap_left = xMin、bitmap_top = yMax です。

株式会社日昇テクノロジー



第 41 行、原点を移動させ、原点の位置を次の文字の原点に移動させます。

32.3.3.3 まとめ

freetype を使用してベクターフォント文字列を生成する手順

1. 生成したいベクターフォントのエンコーディング（GBK、Unicode など）を取得します。

2. freetype を初期化します。

3. freetype で、生成した文字のエンコーディングをサポートする字形ファイルを開きます。

4. 文字サイズを設定します。

5. 文字列を収容する外枠を計算します。

6. 回転角（省略可能）と原点の位置を設定します。

7. ビットマップをロードします（3 つの関数で置き換え可能）。

8. ビットマップのバッファを使用して、ドット描画関数でドットを描画します。

株式会社日昇テクノロジー

32.3.4 freetype の上級

freetype の公式ウェブサイトには、非常に詳細なドキュメントがあり、私の説明は公式ドキュメントに比べると、まるで大きな剣を振るう関羽の前で遊ぶようなものです。

freetype の公式ドキュメントには、以下のチュートリアルや豊富な例が含まれています。

- 字間の微調整
- 中央揃え
- 変形された字形シーケンスのレンダリング
- など

第 33 章 画像表示

```
1 # リポジトリの取得
2
3
4 # この章のコードの位置
5 lubancat_rk_code_storage/base_linux/screen/image/
```

この章では、一般的に使用される 3 種類の画像ファイルの表示方法について紹介します。

1. bmp ファイル
2. jpeg ファイル
3. png ファイル

注意：この章では、これら 3 種類のファイルの基本的な画像表示について導入します。より詳細な操作については、関連するウェブサイトを参照し、関連するソースコードをダウンロードして、より高度な操作を行ってください。

株式会社日昇テクノロジー

33.1 BMP

33.1.1 BMP ファイルの取得

まず、生成したい画像ファイルを見つけます。以下の例を参照してください。

画像 evlove.png

https://convertio.co/ja/png-bmp/を開き、png ファイルを bmp ファイルに変換します。

33.1.2 ファイルフォーマット解析

ファイルのバイナリ形式を見ることができるツールが必要です。ここでは UltraEdit を使用して解説します。ウェブサイトから学習版（グリーン版）をダウンロードして使用することができますが、ダウンロードとインストールのプロセスはここでは説明しません。

BMP は一般的な画像フォーマットで、BMP ファイルは 4 つの部分から構成されています。

1. ビットマップファイルヘッダ（bitmap-file header）
2. ビットマップ情報ヘッダ（bitmap-information）
3. カラーパレット（color palette）
4. ビットマップデータ

株式会社日昇テクノロジー

33.1.2.1 ビットマップファイルヘッダ

合計 14 バイト

リスト 1: ビットマップファイルヘッダ

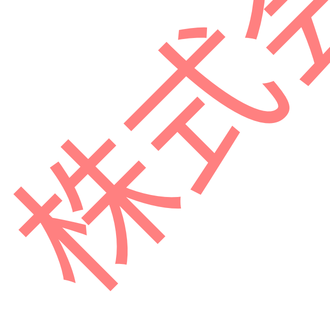
名前	バイト数	オフセット	機能
bfType	2 バイト	0	0x42,0x4D('B','M'), BMP ファイルかどうかを区別する
bfSize	4 バイト	2	BMP ファイルのサイズ
bfReserved1	2 バイト	6	予約されており、0 に設定する必要がある
bfReserved2	2 バイト	8	予約されており、0 に設定する必要がある
bfOffBits	4 バイト	10	ファイルヘッダから実際の画像データまでのバイトオフセット。

bfOffBits 情報ヘッダとカラーパレットの長さは状況によって異なるため、このオフセット値を使用して、ファイルから迅速にビットマップデータを読み取ることができる。

bfOffBits = ビットマップ情報ヘッダ + カラーパレット

```
evlove.bmp   MERCURY.bmp
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 42 4D 46 7A 07 00 00 00 00 00 36 00 00 00 28 00 ; BMFz......6...(.
00000010h: 00 00 5E 01 00 00 5E 01 00 00 01 00 20 00 00 00 ; ..^...^..... ...
00000020h: 00 00 00 00 00 00 C4 0E 00 00 C4 0E 00 00 00 00 ; ......?..?....
00000030h: 00 00 00 00 00 00 2F 1F 0A FF 2F 20 0D FF 2D 1E ; ....../..  / .  -.
00000040h: 0B FF 2A 1C 0A FF 2A 1C 0A FF 29 1C 0C FF 28 1B ; .  *..  *..  )..  (.
```

上の図は UltraEdit で解析されたバイナリファイル、下の図は Windows で見たファイルです。



株式会社日昇テクノロジー

BMP ファイルの最初の 2 バイトはそれぞれ 0x42、0x4D であることが分かります。

- ファイルサイズ：0x467A0700 は、10 進数で 4,618,759 となりますが、明らかにファイルサイズが一致しません。これは、BMP ファイルがリトルエンディアン方式を採用しているためで、正しいファイルサイズは 0x00077A46 で、10 進数で 490,054 となり、Windows で見たサイズと全く同じです。

- 4 バイトが予約されています。

- bfOffBits = 0x36、10 進数で 54、つまりファイルヘッダから画像情報までのオフセットは 54 です。

1. リトルエンディアン：データが複数のバイトでリストされる場合、低位データが低位アドレスに、高位データが高位アドレスに存在します。

2. ビッグエンディアン：データが複数のバイトでリストされる場合、低位データが高位アドレスに、高位データが低位アドレスに存在します。

株式会社日昇テクノロジー

33.1.2.2 ビットマップ情報ヘッダ

リスト 2: ビットマップ情報ヘッダ

名前	バイト数	オフセット	機能
biSize	4 バイト	14	全ビットマップ情報ヘッダ構造体のサイズ
biWidth	4 バイト	18	画像の幅（ピクセル単位）
biHeight	4 バイト	22	画像の高さ（ピクセル単位）
biPlanes	2 バイト	26	色平面数、常に 1
biBitCount	2 バイト	28	1 ピクセルあたりのビット数、値は 1, 4, 8, 16, 24, 32 のいずれか
biCompression	4 バイト	30	データの圧縮タイプ
biSizeImage	4 バイト	34	画像データのサイズ（バイト単位）
biXPelsPerMeter	4 バイト	38	水平解像度（ピクセル/メートル）
biYPelsPerMeter	4 バイト	42	垂直解像度（ピクセル/メートル）
biClrUsed	4 バイト	46	使用する色のインデックス数
biClrImportant	4 バイト	50	画像に重要な影響を与える色のインデックス数

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 42 4D 46 7A 07 00 00 00 00 00 36 00 00 00 28 00 ; BMFz......6...(.
00000010h: 00 00 5E 01 00 00 5E 01 00 00 01 00 20 00 00 00 ; ..^...^..... ...
00000020h: 00 00 00 00 00 00 C4 0E 00 00 C4 0E 00 00 00 00 ; ......?..?....
00000030h: 00 00 00 00 00 00 2F 1F 0A FF 2F 20 0D FF 2D 1E ; ....../..  / .  -.
00000040h: 0B FF 2A 1C 0A FF 2A 1C 0A FF 29 1C 0C FF 28 1B ; .  *..  *..  )..  (.
```

- biSize = 40(0x28)、情報ヘッダのサイズは 40 バイト

- biWidth = 350(0x015E)、画像の幅は 720 ピクセル

- biHeight = 350(0x015E)、画像の高さは 720 ピクセル

注意: biHeight の正負で画像が正向きか逆向きかを判断できます。正の場合は正向き、負の場合は逆向きです。

- biPlanes = 1 色平面数、常に 1

- biBitCount = 32(0x20) 1 ピクセルあたり 32 ビット

- biCompression = 0 圧縮なし

- biSizeImage = 0、一部の bmp ファイルにはこのパラメータがない場合がありますが、bfSize - bfOffBits から計算できます

株式会社日昇テクノロジー

- biXPelsPerMeter = 3780(0x0EC4) 水平解像度 3780 ピクセル/メートル

- biYPelsPerMeter = 3780(0x0EC4) 垂直解像度 3780 ピクセル/メートル

- biClrUsed = 0 色インデックス数は 0（カラーパレット用）

- biClrImportant = 0 重要な色インデックス数は 0（カラーパレット用）

33.1.2.3 カラーパレット

24 ビット真色を使用しているため、カラーパレットは使用しておらず、上記の biClrUsed と biClrImportant は 0 です。これらはカラーパレットの重要なパラメータです。

以下のデータはカラーパレットです。前述のとおり、カラーパレットは色インデックス番号とそれがリストす色の対応関係を示すマッピングテーブルです。ファイル内のレイアウトは二次元配列 palette[N][4] のようで、N は総色インデックス数をリストし、各行の 4 つの要素はそれぞれ B、G、R、Alpha の値をリストし、各成分は 1 バイトを占めます。

透過チャネルを設定しない場合、Alpha は 0 です。インデックス番号は行番号で、対応する色はその行の 4 つの要素です。

- インデックス：(青、緑、赤、Alpha)

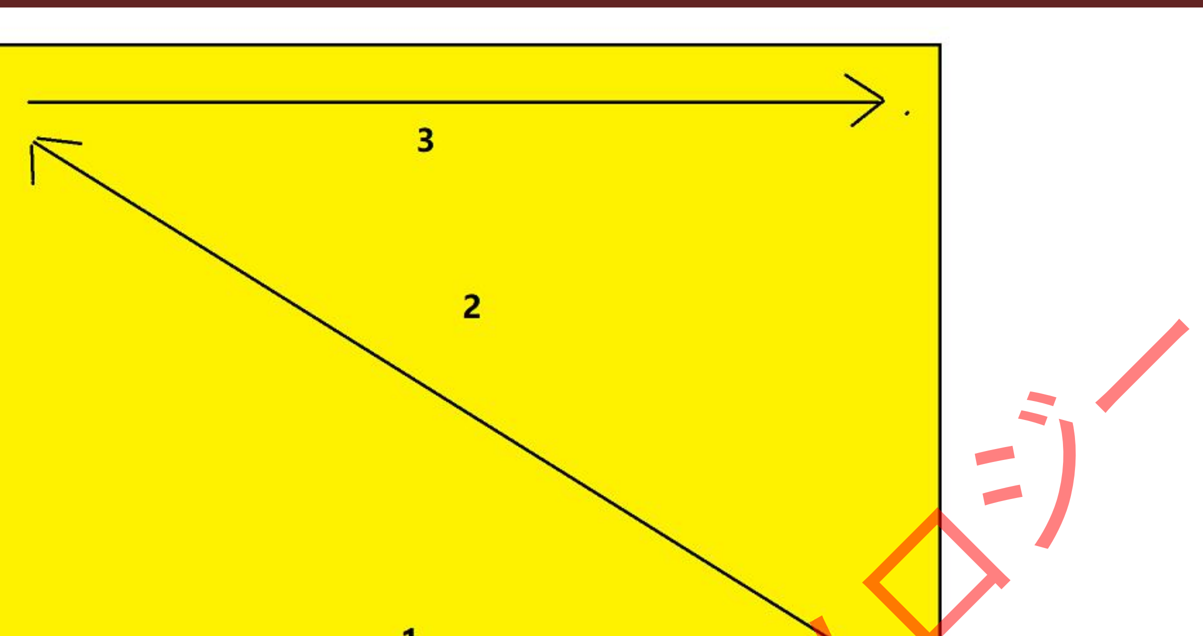
- 0 番：(fe,fa,fd,00)

- 1 番：(fd,f3,fc,00)

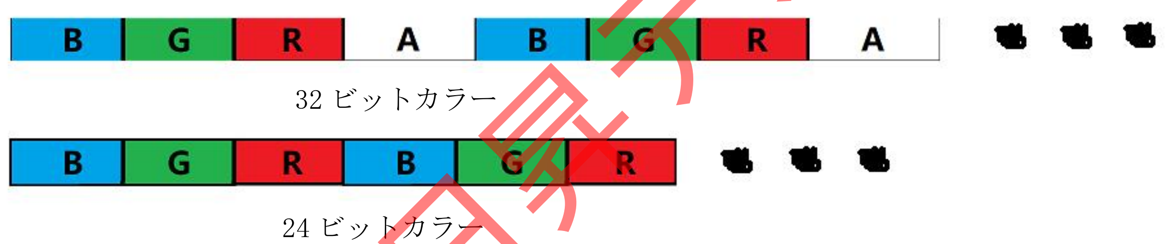
33.1.2.4 ビットマップデータ

biHeight が正であれば、以下のようになります。

株式会社日昇テクノロジー



BMP ファイルがビットマップ情報を保存する方向



32 ビットカラー

24 ビットカラー

- ビットマップデータは左下から右上にかけて格納されているため、画面表示時にデータを調整する必要があります。

- BMP ファイルはリトルエンディアンを使用しているため、24 ビット色と 32 ビット色のデータは BGR および BGRA 形式になります。

- 256 色以下を使用する場合はカラーパレットを使用し、カラーパレットのインデックス番号から色データを取得できます。

33.1.3 コードの位置

```
1 base_linux/screen/image/bmp
```

株式会社日昇テクノロジー

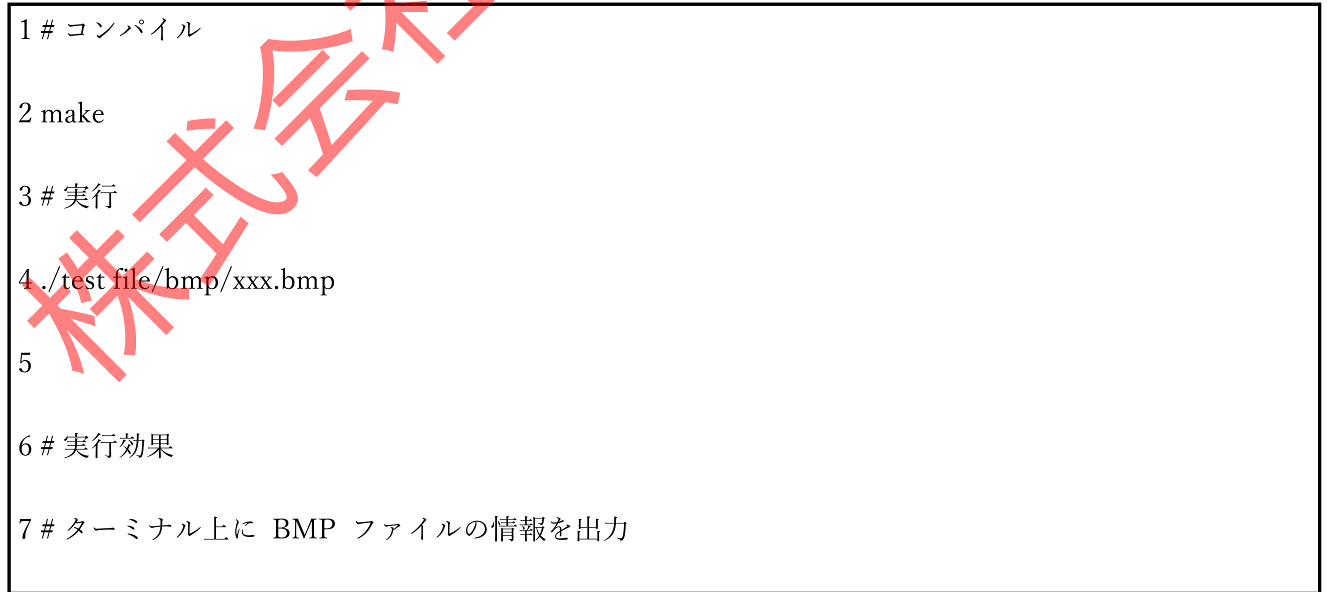
33.1.4 コード構造

```
1 # コード構造

2

3 bmp/

4 |-- Makefile #makefile

5 |-- build

6 | |-- drm-core.o

7 | |-- main.o

8 | `-- test # 実行ファイル

9 |-- file # 画像ファイル

10 | |-- bmp

11 | | |-- MERCURY.bmp

12 | | |-- Night_Visions.bmp

13 | | |-- SILENCE.bmp

14 | | |-- SMOKE+MIRRORS.bmp

15 | | `-- evlove.bmp

16 | |-- jpeg

17 | | |-- MERCURY.jpg

18 | | |-- Night_Visions.jpg

19 | | |-- SILENCE.jpg

20 | | |-- SMOKE+MIRRORS.jpg
```

株式会社日昇テクノロジー

```
21 | | `-- evlove.jpg

22 | `-- png

23 | |-- MERCURY.png

24 | |-- Night_Visions.png

25 | |-- SILENCE.png

26 | |-- SMOKE+MIRRORS.png

27 | `-- evlove.png

28 |-- includes

29 | `-- drm-core.h

30 |-- sources

31 | |-- drm-core.c

32 | `-- main.c

33 `-- test -> build/test # 実行ファイル
```

33.1.5 コンパイル & 実行



```
1 # コンパイル

2 make

3 # 実行

4 ./test file/bmp/xxx.bmp

5

6 # 実行効果

7 # ターミナル上に BMP ファイルの情報を出力
```

株式会社日昇テクノロジー

```
8 # 画面上に画像を表示

9 # キー操作

10 # プログラムを終了
```

33.1.6 コード分析

リスト 1: main()

```
1 int main(int argc, char **argv)

2 {

3 int i,j;

4 int fd_bmp;

5 int ret;

6 struct bmpfile cbf;

7 //BMP ファイルを取得

8 if(argc <2 ){

9 printf("使用方法が間違っています!!!!¥n");

10 printf("使用法: %s xxx.bmp¥n",argv[0]);

11 goto fail1;

12 }

13 //drm を初期化

14 ret = drm_init();

15 if(ret < 0){

16 printf("drm の初期化に失敗しました¥n");
```

株式会社日昇テクノロジー

```
17 return -1;

18 }

19 //ファイルが bmp かどうかを判断

20 ret = judge_bmp(argv[1],&cbf);

21 if(ret < 0){

22 printf("bmp ファイルに問題があります %d¥n",fd_bmp);

23 goto fail1;

24 }

25 //BMP ファイルの各種情報を取得

26 ret = get_bmp_file(argv[1],&cbf);

27 if(ret < 0){

28 printf("bmp ファイルに問題があります %d¥n",fd_bmp);

29 goto fail1;

30 }

31 //情報を表示

32 show_bmp_info(&cbf);

33 //画面情報を中央に表示

34 ret = show_bmp(&cbf , buf.width/2 - cbf.biWidth/2, buf.height/2 - cbf.biHeight/2);

35 if(ret < 0){

36 printf("show_bmp に問題があります！¥n");

37 goto fail2;

38 }
```

株式会社日昇テクノロジー

```
39 //キー入力を取得し、プログラムを終了
40 getchar();
41 //drm を終了
42 drm_exit();
43 //メモリスペースを解放
44 free_bmp_file(&cbf);
45 return 0;
46
47 fail2:
48 drm_exit();
49 free_bmp_file(&cbf);
50 fail1:
51 printf("問題が発生しました、チェックしてください！¥n");
52 return -1;
53
```

- 第 7-12 行、開くべき bmp ファイルを取得します。bmp ファイルがない場合はプログラムを終了します。

- 第 13-18 行、DRM を初期化して表示に使います。drm_init をフレームバッファの初期化に置き換えることも可能です。

- 第 19-24 行、ファイルが bmp であるかを判断し、そうであればファイルサイズとオフセットを取得します。

- 第 25-30 行、bmp ファイルのビットマップ情報ヘッダとビットマップ情報を取得し、BMP のビットマップ形式をスクリーン座標系の RGB 情報に変換します。

株式会社日昇テクノロジー

- 第 31-32 行、bmp ファイルに関するすべての情報を印刷します、以下の図のように。

```
1 struct bmpfile {
2 FILE* fp;
3 int fd;
4 int file_size; //ファイル内の画像のサイズ - 4 バイト
5 int bmp_offset; //画像のオフセット - 4 バイト
6 int biSize; //ビットマップ情報ヘッダのサイズ - 4 バイト
7 int biWidth; //画像の幅 - 4 バイト
8 int biHeight; //画像の高さ - 4 バイト
9 short biPlanes; //カラープレーン数 - 2 バイト
10 short biBitCount; //ピクセルあたりのビット数 - 2 バイト
11 int biCompression; //データ圧縮タイプ - 4 バイト
12 int biSizeImage; //画像データのサイズ - 4 バイト
13 int biXPelsPerMeter; //ピクセル/メートル - 4 バイト
14 int biYPelsPerMeter; //ピクセル/メートル - 4 バイト
15 int biClrUsed; //カラーパレットのインデックス数 - 4 バイト
16 int biClrImportant; //重要なインデックス数 - 4 バイト
17 unsigned char *mem_buf; //メモリマップ、ファイル全体
18 unsigned char *bmp_buf; //BMP ファイルのフォーマットを画像の RGB フォーマットに変更
19 unsigned char bpp; //ピクセルごとに使用されるバイト数
20 };
```

株式会社日昇テクノロジー

- 第 33-38 行、画面の正中央に画面情報を表示します。

- 第 39-40 行、キー情報を取得します。取得するまでプログラムはブロック状態になります。

- 第 41-42 行、drm を登録解除します。

- 第 43-44 行、メモリスペースを解放します。

- 第 46-53 行、エラー情報の実行方向フロー制御。

```
1 int judge_bmp(char *filename, struct bmpfile *pfd)

2 {

3 int offset = 0;

4 int count;

5 char file_head[14];

6 //ファイルを開く

7 pfd->fp = fopen(filename, "rb");

8 if(pfd->fp == NULL){

9 printf("ファイルオープン失敗¥n");

10 return -1;

11 }

12 //最初の 14 バイトを取得

13 count = fread(file_head, 1, 14, pfd->fp);

14 if(count != 14){

15 printf("ファイル読み込み失敗¥n");

16 return -1;
```

株式会社日昇テクノロジー

```
17 }
18 //BMP ファイルかどうかを判断
19 if(file_head[0] == 0x42 && file_head[1] == 0x4d){
20 //パラメータを解析
21 offset = 2;
22 memcpy(&pfd->file_size, file_head+offset, 4);
23 offset += 8; //中間に 4 バイトの空白領域がある
24 memcpy(&pfd->bmp_offset, file_head+offset, 4);
25 fclose(pfd->fp);
26 return 1;
27 }
28 else{
29 fclose(pfd->fp);
30 return -1;
31 }
32
33 }
```

- 第 6-11 行、提供されたファイルを開きます。

- 第 12-17 行、ファイルヘッダの最初の 14 バイトを読み取ります。

- 第 19 行、ファイルが bmp であるかを判断します。

- 第 21-24 行、ファイルのサイズとファイルヘッダから実際の画像データまでのオフセットを取得します。

- 第 28-31 行、ファイルが bmp でなければプログラムを終了します。

株式会社日昇テクノロジー

```
1 int get_bmp_file(char *filename, struct bmpfile *bf)
2 {
3
4 int i;
5 int word = 0;
6 char bmp_head[14];
7 int size;
8 int offset = 14;
9
10 bf->fd = open(filename, O_RDWR);
11 if(bf->fd < 0){
12 printf("ファイルを開けません¥n");
13 return -1;
14 }
15
16 bf->mem_buf = (unsigned char *)mmap(NULL, bf->file_size, PROT_READ | PROT_WRITE,
MAP_SHARED, bf->fd, 0);
17 if(bf->mem_buf == NULL){
18 printf("mmap エラーです！¥n");
19 close(bf->fd);
20 return -1;
21 }
```

株式会社日昇テクノロジー

```
22
23 memcpy(&bf->biSize, bf->mem_buf+offset, 4);
24 offset += 4;
25 memcpy(&bf->biWidth, bf->mem_buf+offset, 4);
26 offset += 4;
27 memcpy(&bf->biHeight, bf->mem_buf+offset, 4);
28 offset += 4;
29 memcpy(&bf->biPlanes, bf->mem_buf+offset, 2);
30 offset += 2;
31 memcpy(&bf->biBitCount, bf->mem_buf+offset, 2);
32 offset += 2;
33 memcpy(&bf->biCompression, bf->mem_buf+offset, 4);
34 offset += 4;
35 memcpy(&bf->biSizeImage, bf->mem_buf+offset, 4);
36 offset += 4;
37 memcpy(&bf->biXPelsPerMeter, bf->mem_buf+offset, 4);
38 offset += 4;
39 memcpy(&bf->biYPelsPerMeter, bf->mem_buf+offset, 4);
40 offset += 4;
41 memcpy(&bf->biClrUsed, bf->mem_buf+offset, 4);
42 offset += 4;
43 memcpy(&bf->biClrImportant, bf->mem_buf+offset, 4);
```

株式会社日昇テクノロジー

```
44
45 if(bf->biSizeImage == 0)
46 bf->biSizeImage = bf->file_size - bf->bmp_offset;
47
48 if(bf->biBitCount == 24 || bf->biBitCount == 32)
49 bf->bpp = bf->biBitCount / 8;
50
51 //空間を確保し、ポインタ操作のために空間を割り当てる
52 bf->bmp_buf = malloc(bf->biSizeImage);
53
54 //BMP ファイルの形式を RGB に適した形式に変更
55 for(i = 0; i < bf->biHeight; i++)
56 memcpy(bf->bmp_buf + i * bf->biWidth * bf->bpp, bf->mem_buf + (bf->biHeight - 1 - i) * bf->biWidth * bf->bpp + bf->bmp_offset, bf->biWidth * bf->bpp);
57
58 close(bf->fd);
59 return 0;
60 }
```

- 第 10-14 行、ファイルを open 形式で開き、mmap に便利です。

- 第 16-23 行、bmp ファイル全体をメモリに mmap します。

- 第 25-45 行、ビットマップ情報ヘッダを取得して解析します。

- 第 47-48 行、一部の bmp ファイルが biSizeImage 情報を持たない場合、ファイル全体のサイズからビットマップ情報のオフセットを引いて計算します。

株式会社日昇テクノロジー

- 第 50-51 行、各ピクセルが占めるバイト数を取得します。24 ビットは 3 バイト、32 ビットは 4 バイトです。

- 第 53-54 行、表示画像のサイズに等しいスクリーン座標系の buf にメモリスペースを割り当てます。

- 第 56-60 行、BMP ファイルのビットマップデータをスクリーン座標系の表示データに変換します。効果は、bmp の最初の行のデータを最後の行に、第二行を最後から二番目の行に変換します。

- 第 62 行、画像ファイルを閉じます。

- この関数は、24 ビットと 32 ビットの bmp 画像を自動的に識別して表示できます。

- 第 13 行、画像が画面の右端および下端を超えない場合に表示します。

- 第 14 行-18 行、BGRA または BGR のビットマップ情報を、画像が識別できる ARGB および RGB に変換します。

- 第 20-21 行、画面の右端および下端を超える場合は表示しません。

- ビットマップ情報ヘッダの詳細情報を印刷します。

- メモリスペースを解放します。

リスト 5: show_bmp_info()

```
1 void show_bmp_info(struct bmpfile *bf)
2 {
3 //ファイルの内容をプリントする
4 printf("file_size = %d¥n",bf->file_size);
5 printf("bmp_offset = %d¥n",bf->bmp_offset);
6 printf("biSize = %d¥n",bf->biSize);
7 printf("biWidth = %d¥n",bf->biWidth);
8 printf("biHeight = %d¥n",bf->biHeight);
```

株式会社日昇テクノロジー

```
9 printf("biPlanes = %d¥n",bf->biPlanes);
10 printf("biBitCount = %d¥n",bf->biBitCount);
11 printf("biCompression = %d¥n",bf->biCompression);
12 printf("biSizeImage = %d¥n",bf->biSizeImage);
13 printf("biXPelsPerMeter = %d¥n",bf->biXPelsPerMeter);
14 printf("biYPelsPerMeter = %d¥n",bf->biYPelsPerMeter);
15 printf("biClrUsed = %d¥n",bf->biClrUsed);
16 printf("biClrImportant = %d¥n",bf->biClrImportant);
17 }
```

• ビットマップ情報ヘッダの詳細をプリントする

リスト 6: free_bmp_file()

```
1 int free_bmp_file(struct bmpfile *bf)
2 {
3 munmap(bf->mem_buf, bf->file_size);
4 free(bf->bmp_buf);
5 }
```

• メモリ空間を解放する

33.2 JPEG

33.2.1 前言

JPEG は、現在生活で最も一般的に使用されている画像フォーマットであり、損失圧縮方法を使用して保存される画像フォーマットです。圧縮の結果として、出力画像は品質とサイズを同時に満たすことができません。ユーザーは圧縮レベルを調整して所望の品質レベルを達成し、同時にストレージサイズを減ら

株式会社日昇テクノロジー

すことができます。画像に 10:1 の圧縮を適用すると、画像品質に与える影響は無視できます。圧縮値が高いほど、画像品質の劣化は大きくなります。

33.2.2 コード位置

```
1 base_linux/screen/image/jpeg
```

33.2.3 コード構造

```
1 # コード構造
2 .
3 |-- Makefile
4 |-- build
5 | |-- drm-core.o
6 | |-- main.o
7 | `-- test
8 |-- file
9 | |-- bmp
10 | | |-- MERCURY.bmp
11 | | |-- Night_Visions.bmp
12 | | |-- SILENCE.bmp
13 | | |-- SMOKE+MIRRORS.bmp
14 | | `-- evlove.bmp
15 | |-- jpeg
16 | | |-- MERCURY.jpg
```

株式会社日昇テクノロジー

```
17 | | |-- Night_Visions.jpg

18 | | |-- SILENCE.jpg

19 | | |-- SMOKE+MIRRORS.jpg

20 | | `-- evlove.jpg

21 | `-- png

22 | |-- MERCURY.png

23 | |-- Night_Visions.png

24 | |-- SILENCE.png

25 | |-- SMOKE+MIRRORS.png

26 | `-- evlove.png

27 |-- includes

28 | `-- drm-core.h

29 |-- sources

30 | |-- drm-core.c

31 | `-- main.c

32 `-- test -> build/test
```

株式会社日昇テクノロジー

33.2.4 コンパイル＆実行

```
1 # jpeg 解码ライブラリのインストール

2 sudo apt install libjpeg-dev

3

4 # コンパイル

5 make

6

7 # 実行

8 ./test xxx.jpeg

9

10 # 例:

11 ./test file/jpeg/evolve.jpg

12

13 # 実行結果

14 画面中央に画像を表示
```

33.2.5 コード分析

リスト 7: main()

```
1 int main(int argc, char **argv)

2 {

3 int i,j;

4 int x =0, y=0 ;

5 uint32_t word;
```

株式会社日昇テクノロジー

```
6 int ret;

7 struct jpeg_file jf;

8

9 if(argc <2 ){

10 printf("Wrong use !!!!¥n");

11 printf("Usage: %s [xxx.jpg / xxx.jpeg]¥n",argv[0]);

12 goto fail1;

13 }

14

15 ret = drm_init();

16 if(ret < 0 ){

17 printf("drm_init fail¥n");

18 return -1;

19 }

20

21 jf.fp= fopen(argv[1], "rb");

22 if (jf.fp== NULL) {

23 printf("can not open file");

24 return -1;

25 }

26

27 memcpy(jf.filename,argv[1],sizeof(argv[1]));
```

株式会社日昇テクノロジー

```
28

29 ret =judge_jpeg(argv[1],&jf);

30 if(ret < 0 ){

31 goto fail2;

32 }

33

34 ret = decode_jpeg(argv[1],&jf);

35 if(ret < 0 ){

36 printf("decode_jpeg wrong¥n");

37 goto fail2;

38 }

39

40 show_jpeg(&jf , buf.width/2 - jf.width/2, buf.height/2 - jf.height/2);

41

42 getchar();

43

44 fclose(jf.fp);

45 free_jpeg(&jf);

46 drm_exit();

47 return 0;

48

49 fail2:
```

株式会社日昇テクノロジー

```
50 fclose(jf.fp);

51 free_jpeg(&jf);

52 drm_exit();

53 fail1:

54 printf("Proglem run fail,please check !¥n");

55 return -1;

56 }
```

- 第 9-13 行、第二引数から jpeg ファイルを取得します。第二引数がない場合はプログラムを終了します。
- 第 15-19 行、表示システムを初期化します。
- 第 21-27 行、ファイルを開きます。
- 第 29-32 行、ファイルが JPEG かどうかを判断します。
- 第 34-38 行、jpeg ファイルをデコードします。
- 第 40 行、画像を表示します。

```
1 //ファイルが jpeg かどうかを判断する

2 int judge_jpeg(char *filename, struct jpeg_file *pfd)

3 {

4 int ret;

5 uint32_t word;

6

7 pfd->cinfo.err = jpeg_std_error(&pfd->jerr);
```

株式会社日昇テクノロジー

```
8 //jpeg_compress_struct 構造体を作成する

9 jpeg_create_decompress(&pfd->cinfo);

10

11 //jpeg 解凍のソースファイルを指定する

12 jpeg_stdio_src(&pfd->cinfo, pfd->fp);

13

14 //jpeg ファイルを解析し、解析完了後に画像の形式を取得する

15 ret = jpeg_read_header(&pfd->cinfo, TRUE);

16 if(ret < 0){

17 printf("file is not jpg ...¥n");

18 jpeg_destroy_decompress(&pfd->cinfo);

19 return -1;
```

- 第 7-9 行、jpeg_compress_struct 構造体を割り当てて初期化します。

- 第 11-12 行、jpeg の解凍元ファイルを指定します。

- 第 14-19 行、jpeg のヘッダを解析し、JPEG ファイルかどうかを判断します。

リスト 9: decode_jpeg()

```
1 int decode_jpeg(char *filename, struct jpeg_file *jf)

2 {

3     int ret;

4     uint32_t word;

5     unsigned char *buf_cpy;

6     // 一時変数行 buffer
```

株式会社日昇テクノロジー

```
7   uint8_t *pucbuffer;
8   // 拡大率
9   jf->cinfo.scale_num = 2;
10  // 縮小率
11  jf->cinfo.scale_denom = 1;
12
13  // cinfo で指定されたソースファイルを解凍し、解凍後のデータを cinfo 構造体のメンバ変数に格納
します。
14  jpeg_start_decompress(&jf->cinfo);
15  // 画像の基本情報を取得
16  jf->row_size = jf->cinfo.output_width * jf->cinfo.output_components;
17  jf->width = jf->cinfo.output_width;
18  jf->height = jf->cinfo.output_height;
19  jf->Bpp = jf->cinfo.output_components;
20  jf->size = jf->row_size * jf->cinfo.output_height;
21  // メモリ空間を割り当てる
22  pucbuffer = malloc(jf->row_size);
23  jf->buffer = malloc(jf->size);
24
25  printf("size: %d w: %d h: %d row_size: %d ,Bpp: %d¥n",
26  jf->size, jf->width, jf->height, jf->row_size, jf->Bpp);
27  // バッファポインタが buffer を指す
```

株式会社日昇テクノロジー

```
28  buf_cpy = jf->buffer;

29

30  while (jf->cinfo.output_scanline < jf->cinfo.output_height) {

31    // RGB データを buffer に読み込むことができます、パラメータ 3 は読み込む行数を指定できま
す

32    jpeg_read_scanlines(&jf->cinfo, &pucbuffer, 1);

33    // メモリにコピー

34    memcpy(buf_cpy, pucbuffer, jf->row_size);

35    buf_cpy = buf_cpy + jf->row_size;

36  }

37

38  // デコードを完了

39  jpeg_finish_decompress(&jf->cinfo);

40  free(pucbuffer);

41  // 構造体を解放

42  jpeg_destroy_decompress(&jf->cinfo);

43

44  return 0;

45 }
```

- 第 8-11 行、拡大または縮小の倍率を設定します。

- 第 13-14 行、解凍を開始し、解凍された内容を cinfo 構造体に格納します。

- 第 15-20 行、画像の基本情報を取得します（各行のバイト数、幅、高さ、Bpp、サイズなど）。

- 第 21-23 行、画像情報を取得するためのメモリスペースを割り当てます。

株式会社日昇テクノロジー

- 第 25-26 行、関連情報を印刷します。

- 第 30-36 行、jpeg_read_scanlines()を使用してデータをメモリに読み込みます。

- 第 39-42 行、デコードを完了し、メモリを解放し、構造体を破壊します。

リスト 10: show_jpeg()

```
1 int show_jpeg(struct jpeg_file *jf,int x ,int y)

2 {

3

4 int i,j;

5 uint32_t word;

6

7 for(j=0; j<jf->height; j++){

8 for(i = 0 ; i<jf->width;i++){

9 if((j+y) < buf.height && (i+x)<buf.width){

10 word = 0;

11 word = (word | jf->buffer[(j*jf->width+i)*jf->Bpp+2]) |

12 (word | jf->buffer[(j*jf->width+i)*jf->Bpp+1])<<8 |

13 (word | jf->buffer[(j*jf->

,→width+i)*jf->Bpp])<<16;

14 buf.vaddr[(j+y)*buf.width+(x+i)] = word;

15 }

16 else

17 continue;

18 }
```

株式会社日昇テクノロジー

```
19 }
20 }
21
```

- 24 ビット画像、jpeg デコードの RGB 形式は RGB888 です。

- 32 ビット画像、jpeg デコードの RGB 形式は ARGB8888 です。

33.2.6 まとめ

デコードの手順：

1. jpeg_std_error(), jpeg_create_decompress()で jpeg_compress_struct 構造体を作成します。
2. jpeg_stdio_src()で jpeg の解凍元ファイルを指定します。
3. jpeg_read_header()で jpeg ファイルを解析します。
4. jpeg_start_decompress()で cinfo が指定するソースファイルを解凍します。
5. jpeg_read_scanlines()で画像データを読み取ります。
6. jpeg_finish_decompress()でデコードを完了します。
7. jpeg_destroy_decompress()で構造体を解放します。

33.3 PNG

33.3.1 前言

PNG（Portable Network Graphics）は、無損失圧縮アルゴリズムを使用するビットマップ形式で、インデックス、グレースケール、RGB の 3 つの色スキームおよび Alpha チャネルなどの特性をサポートしています。その設計目的は、GIF および TIFF ファイル形式を置き換えることであり、GIF ファイル形式が持たないいくつかの特性を追加します。PNG は LZ77 から派生した無損失データ圧縮アルゴリズムを使用し、通常は JAVA プログラム、ウェブページ、または S60 プログラムで使用されます。その理由は、

株式会社日昇テクノロジー

高い圧縮比と生成されるファイルサイズが小さいことです。PNG ファイルの拡張子は.png です。

33.3.2 コード位置

```
1 base_linux/screen/image/png
```

33.3.3 コード構造

```
1 # 代码结构
2 .
3 |-- Makefile
4 |-- build
5 | |-- drm-core.o
6 | |-- main.o
7 | `-- test
8 |-- file
9 | |-- bmp
10 | | |-- MERCURY.bmp
11 | | |-- Night_Visions.bmp
12 | | |-- SILENCE.bmp
13 | | |-- SMOKE+MIRRORS.bmp
14 | | `-- evlove.bmp
15 | |-- jpeg
16 | | |-- MERCURY.jpg
17 | | |-- Night_Visions.jpg
18 | | |-- SILENCE.jpg
```

株式会社日昇テクノロジー

19 | | |-- SMOKE+MIRRORS.jpg

```
20 | | `-- evlove.jpg

21 | `-- png

22 | |-- MERCURY.png

23 | |-- Night_Visions.png

24 | |-- SILENCE.png

25 | |-- SMOKE+MIRRORS.png

26 | `-- evlove.png

27 |-- includes

28 | `-- drm-core.h

29 |-- sources

30 | |-- drm-core.c

31 | `-- main.c

32 `-- test -> build/test
```

33.3.4 コンパイル&実行

```
1 # コンパイル

2 make

3

4 # 実行

5 ./test xxx.png

6

7 # 例:
```

株式会社日昇テクノロジー

```
8 ./test file/png/evolve.png

9

10 # 実行結果

11 画面の中央に画像が表示される
```

33.3.5 コード分析

リスト 11: main()

```
1 int main(int argc, char **argv)

2 {

3     int i,j;

4     int ret;

5     uint32_t word;

6     struct png_file pfd;

7

8     if (argc < 2) {

9         printf("使用方法が間違っています!!!!¥n");

10        printf("使用法: %s [xxx.png]¥n", argv[0]);

11        goto fail1;

12    }

13

14    ret = drm_init();

15    if (ret < 0) {
```

株式会社日昇テクノロジー

```
16      printf("drm_init 失敗¥n");
17      goto fail1;
18  }
19
20  // ファイルを開く
21  pfd.fp = fopen(argv[1], "rb");
22  if (pfd.fp == NULL) {
23      printf("ファイルを開けません");
24      return -1;
25  }
26
27  memcpy(pfd.filename, argv[1], sizeof(argv[1]));
28
29  ret = judge_png(&pfd);
30  if (ret < 0)
31      goto fail1;
32
33  ret = decode_png(&pfd);
34  if (ret < 0) {
35      printf("%d", ret);
36      goto fail2;
37  }
```

株式会社日昇テクノロジー

```
38
39    // 画面の中央に画像を表示
40    show_png(&pfd, buf.width / 2 - pfd.width / 2, buf.height / 2 - pfd.height / 2);
41
42    getchar();
43    free_png(&pfd);
44    drm_exit();
45    return 0;
46
47 fail2:
48    drm_exit();
49 fail1:
50    printf("問題が発生しました、確認してください！¥n");
51    return -1;
52 }
```

- 第 8-12 行、第二引数から png ファイルを取得します。第二引数がない場合はプログラムを終了します。

- 第 14-18 行、表示システムを初期化します。

- 第 20-25 行、ファイルを開きます。

- 第 29-31 行、ファイルが PNG かどうかを判断します。

- 第 33-37 行、PNG ファイルをデコードします。

- 第 40 行、画像を表示します。

株式会社日昇テクノロジー

リスト 12: judge_png()

```
1 int judge_png(struct png_file *pfd)
2 {
3     int ret;
4     char file_head[8];
5     // ファイルの最初の 8 バイトを読み取る
6     if (fread(file_head, 1, 8, pfd->fp) != 8)
7         return -1;
8     // 8 バイトに基づいて、ファイルが PNG ファイルかどうかを判断する
9     ret = png_sig_cmp(file_head, 0, 8);
10    if (ret < 0) {
11        printf("PNG ファイルではありません¥n");
12        return ret;
13    }
14
15    return PNG_FILE;
16 }
```

- 第 5-7 行、ファイルの最初の 8 バイトを読み取ります。

- 第 8-13 行、8 バイトに基づいてファイルが png かどうかを判断します。

株式会社日昇テクノロジー

```
1 int decode_png(struct png_file *pfd)
2 {
3     int ret;
4     int i, j, k;
5     uint32_t word;
6     unsigned char **pucPngData;
7     unsigned char *buf_cpy;
8
9     // libpng 関連の二つの構造体を割り当てて初期化する
10    pfd->png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
11    pfd->info_ptr = png_create_info_struct(pfd->png_ptr);
12
13    // エラー戻りポイントを設定する
14    setjmp(png_jmpbuf(pfd->png_ptr));
15    // fseek(fp, 0, SEEK_SET);
16    rewind(pfd->fp);
17    // ファイルを指定する
18    png_init_io(pfd->png_ptr, pfd->fp);
19
20    // PNG 画像の情報を取得する
21    png_read_png(pfd->png_ptr, pfd->info_ptr, PNG_TRANSFORM_EXPAND, 0);
22
```

株式会社日昇テクノロジー

```
23  // チャンネル 4-32 ビット/3-24 ビット/...
24  pfd->Bpp = png_get_channels(pfd->png_ptr, pfd->info_ptr);
25  pfd->width = png_get_image_width(pfd->png_ptr, pfd->info_ptr);
26  pfd->height = png_get_image_height(pfd->png_ptr, pfd->info_ptr);
27  pfd->bpp = png_get_bit_depth(pfd->png_ptr, pfd->info_ptr) * pfd->Bpp;
28  pfd->row_size = png_get_rowbytes(pfd->png_ptr, pfd->info_ptr);
29  pfd->size = pfd->width * pfd->height * pfd->Bpp;
30
31  pfd->buffer = (unsigned char*)malloc(pfd->size);
32
33  // 画像を行ごとに一括で取得する
34  pucPngData = png_get_rows(pfd->png_ptr, pfd->info_ptr);
35  buf_cpy = pfd->buffer;
36  // バッファ領域に保存する
37  for (j = 0; j < pfd->height; j++) {
38      memcpy(buf_cpy, pucPngData[j], pfd->row_size);
39      buf_cpy += pfd->row_size;
40  }
41
42  png_destroy_read_struct(&pfd->png_ptr, &pfd->info_ptr, NULL);
43  fclose(pfd->fp);
44 }
```

株式会社日昇テクノロジー

- 第 9-11 行、png ファイルに関連する 2 つの構造体を作成します。後の png 操作に必要です。

- 第 13-14 行、エラーリターンポイントを設定します。

- 第 15-16 行、ファイルポインターをヘッダに置きます。

- 第 17-18 行、デコードするファイルを指定します。

- 第 20-21 行、PNG の画像情報を取得します。

- 第 23-29 行、PNG の画像情報を png_file 構造体に保存します。

- 第 33-34 行、画像情報を二次元配列に保存します。

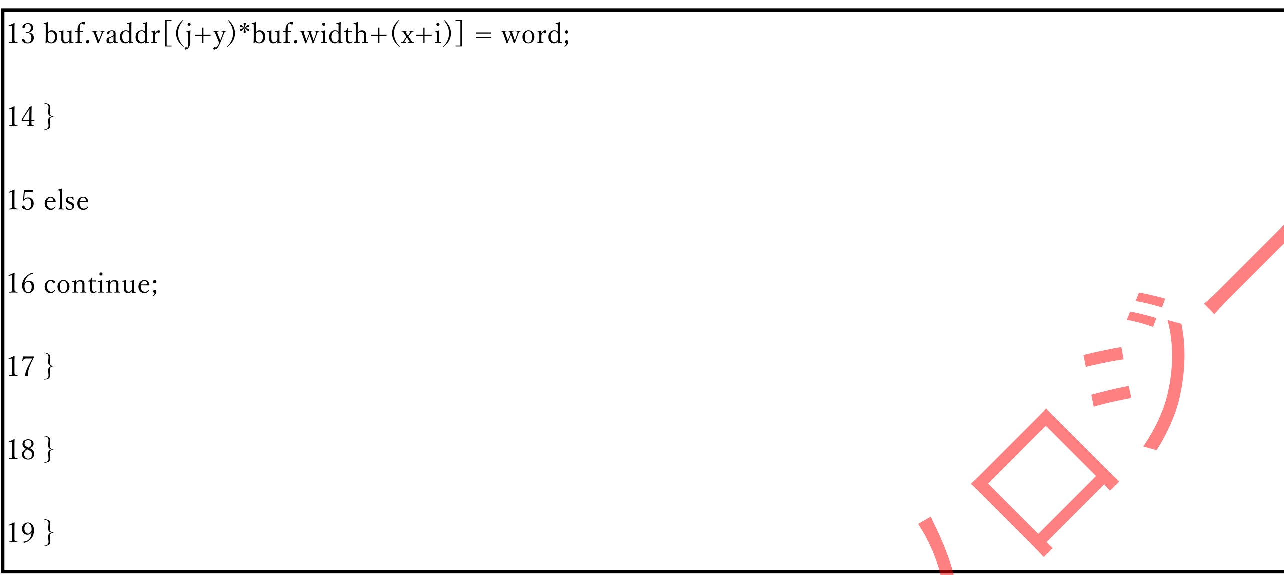
- 第 35-40 行、二次元配列の画像データをバッファに変換します。

- 第 42 行、構造体を破壊します。

リスト 14: show_png()

```
1 int show_png(struct png_file *pfd,int x ,int y)

2 {

3 int i,j;

4 uint32_t word;

5

6 for(j=0; j<pfd->height; j++){

7 for(i = 0 ; i<pfd->width;i++){

8 if((j+y) < buf.height && (i+x)<buf.width){

9 word = 0;

10 word = (word | pfd->buffer[(j*pfd->width+i)*pfd->Bpp+2]) |

11 (word | pfd->buffer[(j*pfd->width+i)*pfd->Bpp+1])<<8 |

12 (word | pfd->buffer[(j*pfd->width+i)*pfd->Bpp])<<16;
```

株式会社日昇テクノロジー



```
13 buf.vaddr[(j+y)*buf.width+(x+i)] = word;

14 }

15 else

16 continue;

17 }

18 }

19 }
```

- 24 ビット画像、jpeg デコードの RGB 形式は RGB888 です。

- 32 ビット画像、jpeg デコードの RGB 形式は ARGB8888 です。

33.3.6 まとめ

デコードの手順：

1. jpeg_std_error(), jpeg_create_decompress()で jpeg_compress_struct 構造体を作成します。
2. jpeg_stdio_src()で jpeg の解凍元ファイルを指定します。
3. jpeg_read_header()で jpeg ファイルを解析します。
4. jpeg_start_decompress()で cinfo が指定するソースファイルを解凍します。
5. jpeg_read_scanlines()で画像データを読み取ります。
6. jpeg_finish_decompress()でデコードを完了します。
7. jpeg_destroy_decompress()で構造体を解放します。

第 34 章プロセス

この章では Linux カーネルが提供するプロセス、パイプ、シグナルなどのカーネルオブジェクトについて学びます。これらのカーネルオブジェクトの応用を理解することで、Linux カーネルがアプリケーショ

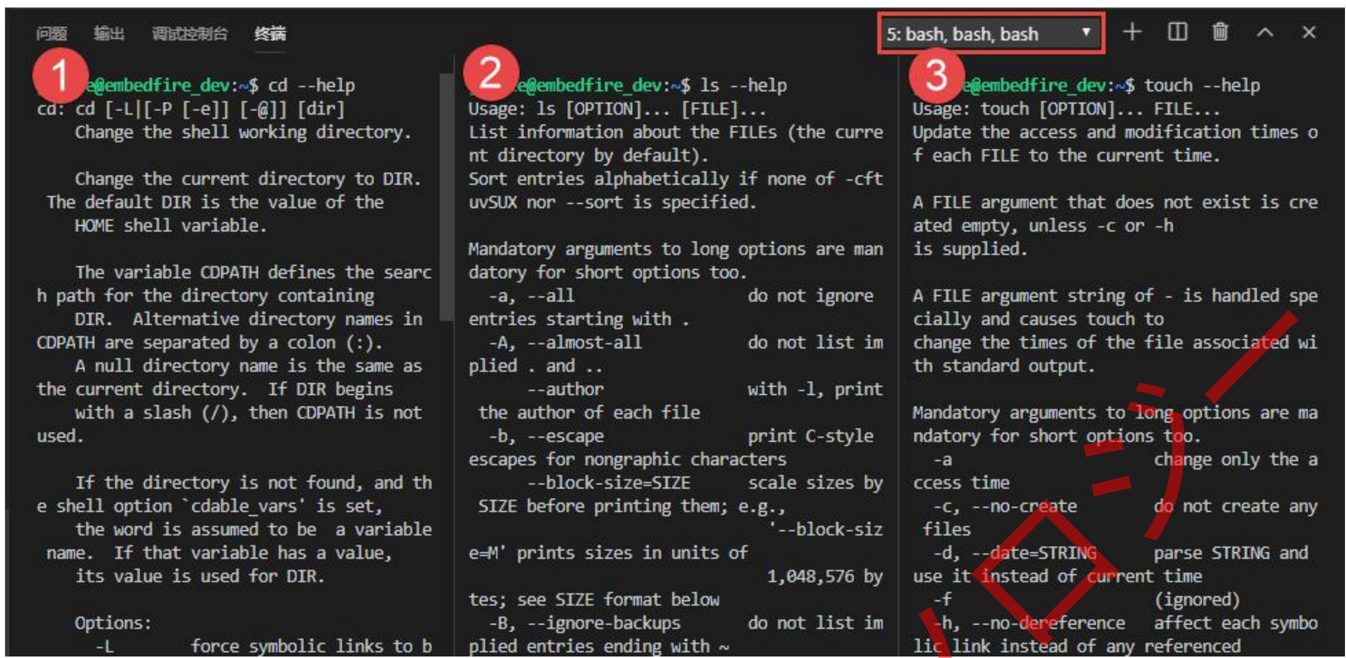
株式会社日昇テクノロジー

ン層に提供する機能についてより深く理解することができるだけでなく、後により低レベルのドライバプログラムを書く際にも役立ちます。

34.1 簡単にプロセスを理解する

本題に入る前に、他の書籍のようにたくさんの原理を語るつもりはありません。むしろ、実験現象を通じて読者にプロセスに関する知識を理解させたいと考えています。その後、プロセスに関連する知識点を補足します。まず、仮想マシンでターミナルを開いてください（この部分を読んでいる読者であれば、既にターミナルが何であるかを理解していると信じています。ただし、著者は会社のサーバーを SSH 方式で接続しているため、以下のスクリーンショットが読者の表示インターフェースと異なる可能性がありますが、これが読者が本章を理解する上で影響を与えることはありません）。一般的に Ubuntu には多くのシェルターミナルがあり、一つのターミナルは一つのプロセスです。もしかすると、読者は複数のターミナルを開いているかもしれませんが、開かれたこれらのターミナルは一連のプロセスとして機能します。これらのプロセスはシステム内で独立して実行され、互いに影響しません。著者は 3 つのシェルターミナルを開きました。これらのターミナルはそれぞれ入出力を持ち、互いに干渉しません。以下の図のようです。実行中の各シェルは個別のプロセスです。もし読者があるシェルからプログラムを呼び出した場合、該当するプログラムは新しいプロセスで実行され、実行が終了するとシェルは作業を続けます。高度なプログラマーはしばしば、プログラムがより多くのタスクを並行して行えるように、またはプログラムをより堅牢にするため、あるいは既存の他のプログラムを直接利用するために、アプリケーション内で複数の協力プロセスを同時に起動します。このようにして、システム内で実行されるプロセスは互いに協力し合い、独立して実行されるのではありません。これはプロセス間通信に関わる内容であり、この部分は後ほど説明します。

株式会社日昇テクノロジー



34.2 プロセスの確認

仮想マシンを起動しても、何もプログラムを実行していなくても、システムにはプロセスが存在します。これは、システムが正常に動作するために必要なプロセスが処理されているためです。Linux では、プロセスに関する情報を記録するためにプロセステーブルを使用しています。プロセステーブルは、メモリにロードされているすべてのプロセスの情報を含むデータ構造で、プロセスの PID、状態、コマンド文字列など、ps コマンドで出力される様々な情報を保持しています。オペレーティングシステムは、これらの PID を使用してプロセスを管理します。これらの PID はプロセステーブルのインデックスであり、現在の Linux システムでは、同時に実行できるプロセスの数は、プロセステーブル項目を作成するために使用されるメモリ容量にのみ関連しており、具体的な数量制限はありません。つまり、システムに十分なメモリがあれば、理論上無限にプロセスを実行することが可能です。

34.2.1 プロセス ID

Linux システム内の各プロセスには、プロセス ID（ProcessID、PID とも呼ばれる）と呼ばれる一意の数字が割り当てられます。プロセス ID は 16 ビットの正の整数で、デフォルトの取得範囲は 2 から 32768 までです（変更可能）。新しいプロセスが開始されると、Linux は自動的に次に使用されていない

株式会社日昇テクノロジー

数字を順番に割り当てます。PID の数値が最大に達すると、システムは次に使用されていない数値を再選択します。新しい PID は 2 から再開されます。これは、PID 番号 1 が通常、システムが実行中に存在する最初のプロセスである特別なプロセス init に予約されているためです。init プロセスは他のプロセスを管理する責任があります。

34.2.2 親プロセス ID

init プロセスを除くすべてのプロセスは、別のプロセスによって開始されます。そのプロセスを開始したプロセスを親プロセスと呼び、開始されたプロセスを子プロセスと呼びます。親プロセスのプロセス ID（PID）は子プロセスの親プロセス ID（PPID）です。ユーザーは`getppid()`関数を呼び出すことで、現在のプロセスの親プロセス ID を取得できます。

これらのプロセスを直感的に見るために、`ps`コマンドを使用してシステム内のプロセス状況を確認できます。`ps`コマンドは、実行中のプロセス、他のユーザーが実行中のプロセス、またはシステム上で現在実行中のすべてのプロセスを表示できます。

#PC や開発ボードで以下のコマンドを実行します：

```
ps –aux
```

```
@embedfire_dev:~$ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0 225648  6708 ?        Ss   9月16   1:46 /sbin/init maybe-ubiquity
root         2  0.0  0.0      0     0 ?        S    9月16   0:00 [kthreadd]
root         4  0.0  0.0      0     0 ?        I<   9月16   0:00 [kworker/0:0H]
root         6  0.0  0.0      0     0 ?        I<   9月16   0:00 [mm_percpu_wq]
root         7  0.0  0.0      0     0 ?        S    9月16   0:47 [ksoftirqd/0]
root         8  0.0  0.0      0     0 ?        I    9月16  27:56 [rcu_sched]
root         9  0.0  0.0      0     0 ?        I    9月16   0:00 [rcu_bh]
root        10  0.0  0.0      0     0 ?        S    9月16   0:11 [migration/0]
root        11  0.0  0.0      0     0 ?        S    9月16   0:04 [watchdog/0]
root        12  0.0  0.0      0     0 ?        S    9月16   0:00 [cpuhp/0]
root        13  0.0  0.0      0     0 ?        S    9月16   0:00 [cpuhp/1]
root        14  0.0  0.0      0     0 ?        S    9月16   0:04 [watchdog/1]
root        15  0.0  0.0      0     0 ?        S    9月16   0:11 [migration/1]
root        16  0.0  0.0      0     0 ?        S    9月16   0:44 [ksoftirqd/1]
root        18  0.0  0.0      0     0 ?        I<   9月16   0:00 [kworker/1:0H]
root        19  0.0  0.0      0     0 ?        S    9月16   0:00 [cpuhp/2]
root        20  0.0  0.0      0     0 ?        S    9月16   0:04 [watchdog/2]
```

出力結果から、プロセス番号 1 のプロセスが init プロセスであることが明確にわかります。それは /sbin/init ディレクトリにあります。もちろん、システムのプロセスはこれだけではありませんが、多すぎ

株式会社日昇テクノロジー

てスクリーンショットに収めることができないため、ここでは一部のプロセスのみを示しています。読者は以下のコマンドを使用して試してみることができます。実際には、Linux の ps コマンドには多くのオプションがあり、多くの異なる UNIX バージョンの ps コマンドとの互換性を試みています。これらのオプションは、どのプロセスを表示し、どの情報を表示するかを決定します。

34.2.3 親プロセスと子プロセス

プロセスが開始されると、開始プロセスは新しいプロセスの親プロセスであり、新しいプロセスは開始プロセスの子プロセスです。

すべてのプロセスには親プロセスがあります（システム内の「ゾンビプロセス」などの特殊なプロセスを除く）。したがって、Linux 内のプロセス構造を木構造として考えることができます。ここで、init プロセスは木の「根」です。または、init プロセスをオペレーティングシステムのプロセスマネージャーと見なすこともできます。それは他のすべてのプロセスの祖先プロセスです。見る他のシステムプロセスは、init プロセスによって開始されるか、init プロセスによって開始された他のプロセスによって開始されます。

要するに、init プロセスの下には多くの子プロセスがあり、これらの子プロセスにはさらに子プロセスが存在する可能性があります。これは家族のようなものです。システム内のすべての親プロセス ID は PPID と呼ばれ、異なるプロセスの親プロセスは異なります。この値は現在のプロセスの親プロセスの ID に過ぎません。システム内の親プロセスと子プロセスは相対的なものであり、祖父 <-> 父 <-> 子の関係のようなものです。父は祖父に対しては子であり、子に対しては父です。

システム内の親プロセスと子プロセスをより直感的に見るために、ここでは`pstree`コマンドを使用してプロセスを木構造でリストアップします。PC や開発ボードで以下のコマンドを実行します：

株式会社日昇テクノロジー

```
pstree

# システムがコマンドを見つけられない場合は、以下の指示でインストールできます：

sudo apt update

sudo apt install psmisc
```

34.3 プログラムとプロセス

ここまでで、プロセスに関する情報を簡単に理解しましたが、多くの読者はまだ疑問を持っているかもしれません。書いたコードはプログラムであり、どのようにしてプロセスになるのでしょうか？このセクションでは、プログラムとプロセスの関係について説明します。

34.3.1 プログラムの概念

プログラム（program）は通常のファイルであり、特定のタスクを完了するために準備された命令列とデータの集合です。これらの命令とデータは、「実行可能イメージ」の形式でディスクに保存されます。書いたコードは、コンパイラによってコンパイルされた後、対応する実行可能ファイルを生成します。これがプログラム、または実行可能プログラムです。

株式会社日昇テクノロジー

```
@embedfire_dev:~$ pstree
systemd─┬─accounts-daemon───2*[{accounts-daemon}]
        ├─agetty
        ├─atd
        ├─bash───sleep
        ├─blkmapd
        ├─cron───cron───sh───run-parts───mlocate───flock───updatedb.mlocat
        ├─dbus-daemon
        ├─dnsmasq
        ├─dockerd─┬─docker-containe─┬─docker-containe─┬─2*[bash]
        │         │                 │                 └─9*[{docker-containe}]
        │         │                 ├─docker-containe─┬─bash
        │         │                 │                 └─9*[{docker-containe}]
        │         │                 └─35*[{docker-containe}]
        │         └─35*[{dockerd}]
        ├─inetd
        ├─irqbalance───{irqbalance}
        ├─2*[iscsid]
        ├─lvmetad
        ├─lxcfs───7*[{lxcfs}]
        ├─networkd-dispat───{networkd-dispat}
        ├─nscd───13*[{nscd}]
        ├─polkitd───2*[{polkitd}]
        ├─privoxy
        ├─rpc.idmapd
        ├─rpc.mountd
        ├─rpcbind
        ├─rsyslogd───3*[{rsyslogd}]
        ├─snapd───27*[{snapd}]
        ├─sshd─┬─sshd───sshd───bash─┬─sh───node─┬─node─┬─3*[bash]
        │      │                    │           │      ├─bash───git───git-remote-http───git─┬─git
        │      │                    │           │      │                                    └─{git}
        │      │                    │           │      ├─bash───pstree
        │      │                    │           │      ├─node───6*[{node}]
        │      │                    │           │      └─18*[{node}]
        │      │                    │           └─11*[{node}]
        │      │                    └─sleep
        │      └─sshd───sshd───3*[sftp-server]
        ├─sslocal
```

34.3.2 プロセスの概念

プロセス（process）は、プログラムの実行の具体的なインスタンスです。例えば、実行可能ファイルが実行される時、それはプロセスになります。プログラムの実行が完了するまで、それはプロセスです。プログラムが実行される過程で、少なくともプロセスの実行環境、CPU、外部デバイス、メモリ、プロセス ID などのシステムリソースを享受します。同じプログラムは、複数のプロセスとしてインスタンス化することができ、Linux システムでは`ps`コマンドを使用して、現在実行中のプロセスを確認できます。この実行可能プログラムが実行完了すると、プロセスもそれに伴い破棄されます（必ずしも即時ではありませんが、最終的には破棄されます）。

プログラムは単独では実行できません。プログラムをメモリにロードし、システムがリソースを割り当て

株式会社日昇テクノロジー

た後にのみ実行できます。このように実行中のプログラムをプロセスと言います。つまり、プロセスはシステムによるリソース割り当てとスケジューリングの独立した単位です。各プロセスには独自のアドレス空間があります。

例えば、`/bin`ディレクトリ下には多くの実行可能ファイルがあります。システムでターミナルを開くと、それはプロセスになります。このプロセスは bash 実行可能ファイル（プログラム）からインスタンス化されます。そして、Linux システムでは複数のターミナルを開くことができ、これらのターミナルはシステム内で独立して実行されます。

```
@embedfire_dev:/bin$ ls -la
total 15356
drwxr-xr-x  2 root root    4096 9月  12 06:21 .
drwxr-xr-x 25 root root    4096 10月  3 06:10 ..
-rwxr-xr-x  1 root root 1113504 4月   4  2018 bash
-rwxr-xr-x  1 root root  716464 3月  12  2018 btrfs
lrwxrwxrwx  1 root root       5 3月  12  2018 btrfsck -> btrfs
-rwxr-xr-x  1 root root  375952 3月  12  2018 btrfs-debug-tree
-rwxr-xr-x  1 root root  371856 3月  12  2018 btrfs-find-root
-rwxr-xr-x  1 root root  396432 3月  12  2018 btrfs-image
-rwxr-xr-x  1 root root  375952 3月  12  2018 btrfs-map-logical
-rwxr-xr-x  1 root root  371856 3月  12  2018 btrfs-select-super
-rwxr-xr-x  1 root root  375952 3月  12  2018 btrfstune
-rwxr-xr-x  1 root root  371856 3月  12  2018 btrfs-zero-log
-rwxr-xr-x  3 root root   34888 7月   4 12:35 bunzip2
-rwxr-xr-x  1 root root 2062296 3月   6  2019 busybox
-rwxr-xr-x  3 root root   34888 7月   4 12:35 bzcat
```

34.3.3 プログラムがプロセスになる過程

Linux システムでは、プログラムは静的なファイルですが、プロセスは動的な実体です。プロセスの状態（後で説明するプロセスの状態）は、実行の過程で変化します。では、プログラムはどのようにしてプロセスになるのでしょうか？

実際、プログラム（実行可能ファイル）を実行するには、通常、Shell でコマンドを入力して実行します。この実行プロセスには、プログラムからプロセスへの変換プロセスが含まれています。この変換プロセスは主に以下の 3 つのステップを含みます：

1. コマンドに対応するプログラムファイルの位置を検索します。

2. fork()関数を使用して新しいプロセスを開始します。

株式会社日昇テクノロジー

3. 新しいプロセスで`exec`族の関数を呼び出してプログラムファイルをロードし、プログラムファイル内の`main()`関数を実行します。

注：具体的な関数については後で詳しく説明します。

34.3.4 まとめ

総じて、プログラムとプロセスの関係は以下のようになります：

1. プログラムは命令シーケンスとデータの集合であり、それ自体には実行の意味はありません。それは静的な実体です。一方、プロセスはプログラムが特定のデータセット上で実行されるプロセスであり、動的に実行される実体で、自身のライフサイクルがあります。それは開始によって生成され、スケジュールによって実行され、リソースやイベントを待って待機状態になり、タスクが完了すると破棄されます。

2. プロセスとプログラムは一対一の関係ではありません。異なるデータセットで実行される同じプログラムは、異なるプロセスになります。プロセス制御ブロックを使用して、システム内の各プロセスを一意に識別できます。これは、プログラムがデータと直接的な関連を持たないため、プログラムができないことです。異なるデータで実行されるプログラムであっても、その命令セットは同じです。したがって、異なるデータセットで実行されるこれらのプログラムを一意に識別することはできません。一般に、プロセスには対応するプログラムがあり、それは一つだけです。しかし、プログラムには対応するプロセスがない場合があります（プログラムが実行されていない場合）、または複数のプロセスが対応している場合があります（プログラムが異なるデータセットで複数回実行されている場合）。

3. プロセスには並行性がありますが、プログラムにはありません。

4. プロセスはコンピュータリソースを競争する基本単位であり、プログラムではありません。

株式会社日昇テクノロジー

34.4 プロセス状態

プロセス状態について学ぶ前に、システム内で一般的に見られるプロセス状態を見てみましょう。これは ps コマンドを使用してシステムで実行中のプロセス情報を出力することで可能です。注目する必要があるのは STAT 列の情報です。プロセスの状態には多くの種類があり、具体的には以下のようになります：

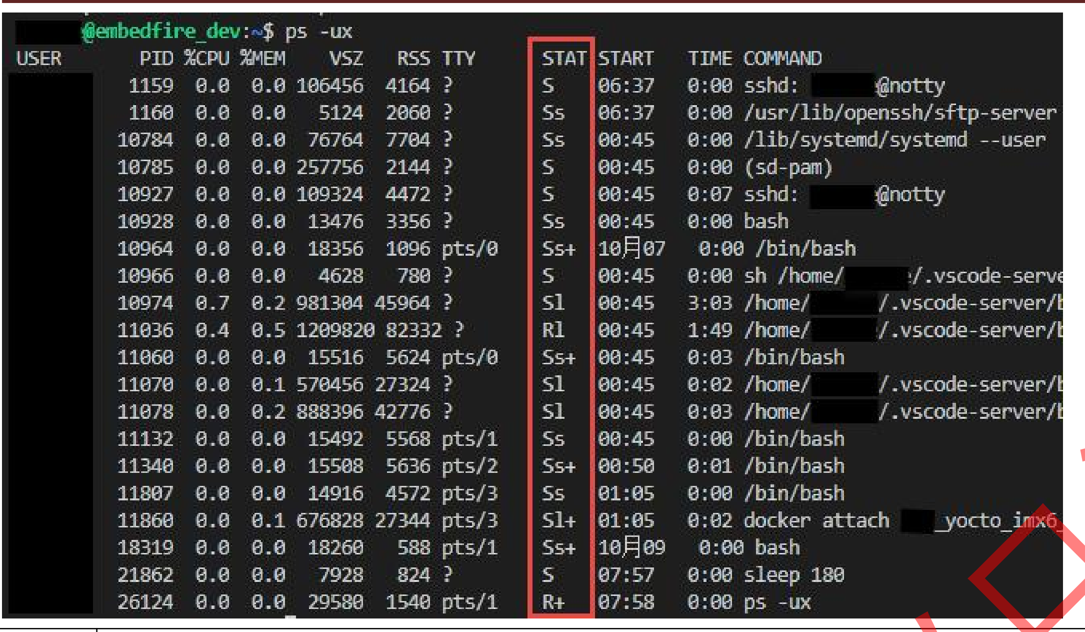
PC や開発ボードで以下のコマンドを実行します：

```
ps -ux
# 出力（短縮版）：
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME  COMMAND
xxx 11132 0.0 0.0 15492 5568 pts/1 Ss 00:45 0:00 /bin/bash
xxx 11340 0.0 0.0 15508 5636 pts/2 Ss+ 00:50 0:01 /bin/bash
xxx 11807 0.0 0.0 14916 4572 pts/3 Ss 01:05 0:00 /bin/bash
xxx 18319 0.0 0.0 18260 588 pts/1 Ss+ 10 月 09 0:00 bash
xxx 21862 0.0 0.0 7928 824 ? S 07:57  0:00 sleep 180
xxx 26124 0.0 0.0 29580 1540 pts/1 R+ 07:58  0:00 ps -ux
```

作者が使用しているのは会社のサーバーなので、ここでは作者のユーザーの現在のプロセス情報のみを出力し、システムのすべてのプロセス情報を出力していません。そのため、`ps`コマンドに`-a`オプションは必要ありません。

上の図から、プロセスの状態には多様なものがあることがわかります。S、Ss、Sl、Rl、R+などの状態がありますが、これらは何を意味しているのでしょうか？実際、これらの状態は Linux システムのプロセスの一部に過ぎず、表示されていないいくつかの状態もあります。これは、作者の現在のユーザー下のすべてのプロセスがそれらの状態にないため、表示されていないからです。以下では、Linux システム内のすべてのプロセス状態について簡単に紹介します。

株式会社日昇テクノロジー



状態	説明
R	実行状態。正確には「実行可能状態」といい、プロセスが実行キュー内にあり、実行中または実行予定の状態を意味します。この状態のプロセスのみが CPU 上で実行される可能性があり、同時に複数のプロセスが実行可能状態にあることがあります。
S	中断可能なスリープ状態。この状態のプロセスは、何らかのイベントの発生を待っているために一時停止されています。例えば、プロセスがシグナルを待っている場合などです。
D	中断不可能なスリープ状態。通常は入出力（I/O）の完了を待っている時に見られ、この状態のプロセスは非同期シグナルに反応できません。
T	停止状態。通常はシェルのジョブコントロールシグナルによるものであるか、デバッガーの制御下にあるために発生します。
Z	終了状態。プロセスがゾンビプロセスになります。
X	終了状態。プロセスが間もなく回収されます。
s	プロセスがセッションリーダーです。
l	プロセスがマルチス
+	プロセスがフォアグラウンドプロセスグループに属しています。
<	高優先度タスクです。

34.5 プロセス状態の変化

前述の紹介からも分かるように、プロセスは動的で活動的なインスタンスです。これは、プロセスが多くの異なる実行状態を持つことを意味し、時にはスリープ状態、一時停止状態、または再実行状態になります。Linux オペレーティングシステムは多ユーザー多タスクのオペレーティングシステムですが、シングルコアの CPU システムでは、ある瞬間に実行状態にあるプロセスは 1 つだけで、他のプロセスは他の

株式会社日昇テクノロジー

状態にあります。これらの状態間でのタスクのスケジュールに従って不断に切り替わります。しかし、CPU の処理速度が非常に速いため、ユーザーはすべてのプロセスが同時に実行されているように感じます。以下の図は、Linux プロセスが開始から終了までのすべての状態、およびこれらの状態の変化条件を示しています。

1. 一般的に、プロセスの開始は親プロセスが`fork`関数を呼び出すことから始まります。したがって、システムが起動すると、`init`プロセスが動作を開始し、システムの実行中に新しいプロセスが継続的に開始されます。これらのプロセスは、`init`プロセスによって開始されるか、または`init`プロセスによって開始された他のプロセスによって開始されます。

2. プロセスが開始されると、実行可能状態（ただし、この時点ではプロセスは CPU を使用していない）になります。この状態にあるプロセスは、プロセス待ち行列で待機しているか、または CPU を使用して実行中です。前者を「準備完了状態」と呼び、後者を「実行状態」と呼びます。

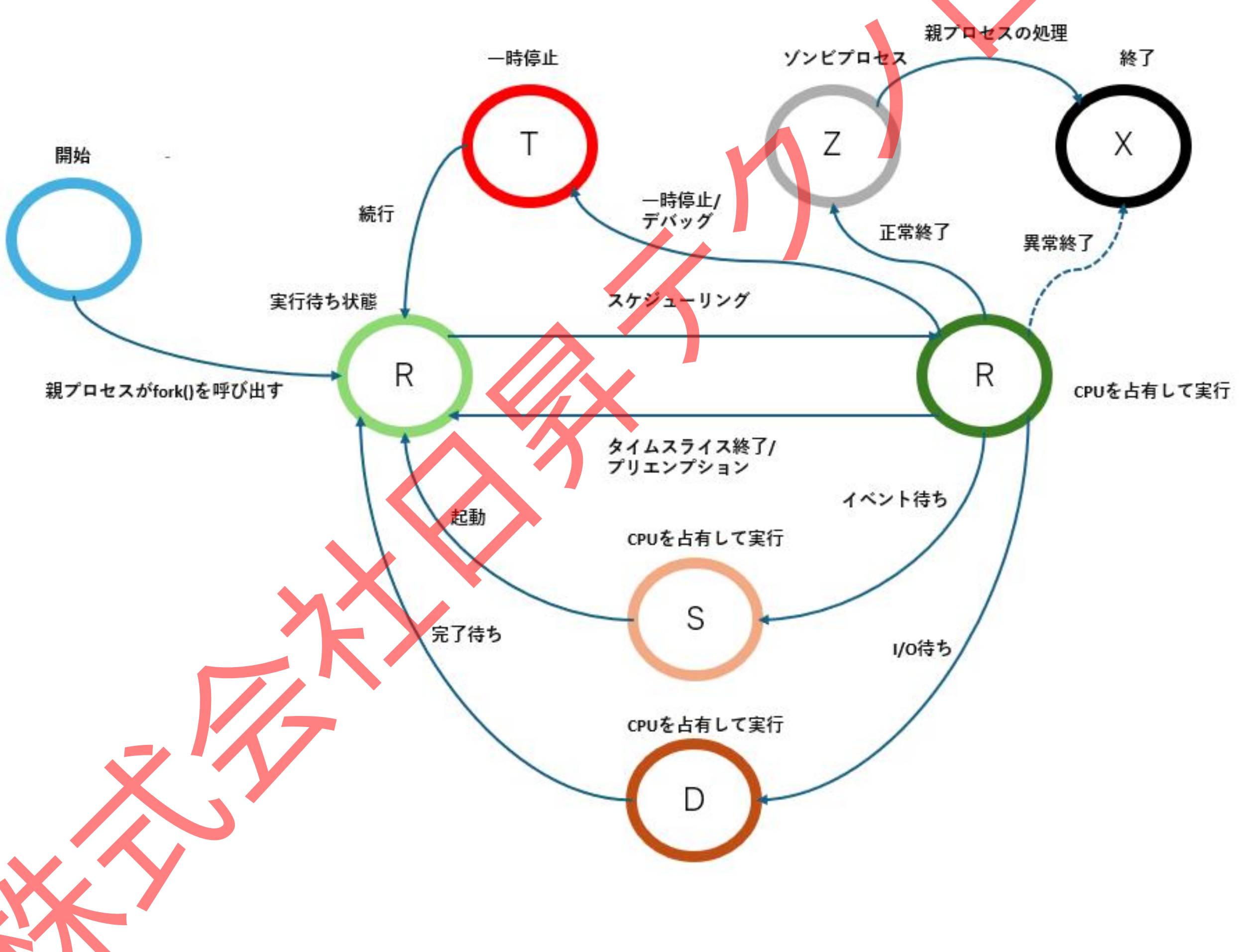
3. システムにプロセススケジュールが発生すると、準備完了状態のプロセスは CPU の使用権を得て、実行状態になります。しかし、各プロセスの実行時間は限られています（例えば 10 ミリ秒）。この実行時間を「タイムスライス」と呼びます。プロセスのタイムスライスが尽きた場合、プロセスがまだ実行を終了していない場合、システムはプロセスを再び待機行列に戻し、プロセスは再び準備完了状態になります。また、実行状態のプロセスは、タイムスライスが尽きていなくても、より高い優先順位を持つ他のプロセスによって「プリエンプト」され、強制的に待機行列に戻されることがあります。

4. 実行状態のプロセスは、あるイベント、シグナル、またはリソースを待つことにより、「割り込み可能なスリープ状態」になることがあります。例えば、プロセスがパイプからデータを読み取ろうとしてパイプが空である場合、またはプロセスがロックを取得しようとしてロックが利用できない場合、またはプロセスが自ら`sleep`関数を呼び出して強制的にスリープ状態に入る場合などです。これらの状況では、プロセスの状態は「割り込み可能なスリープ状態」になります。「割り込み可能なスリー

株式会社日昇テクノロジー

プ状態」とは、中断できる、つまりシグナルに応答できる状態を意味し、特定の条件が満たされた後、プロセスの状態は「準備完了状態」に変わります。例えば、他のプロセスがパイプにデータを書き込んだ後、またはロックリソースが利用可能になった場合、またはスリープ時間が経過した場合などです。

5. もちろん、実行状態のプロセスは「割り込み不可能なスリープ状態」に入ることもあります。この状態のプロセスはシグナルに応答できず、非常に短期間の状態です。通常、この状態のプロセスは入出力（I/O）の完了を待っており、完了後は自動的に「準備完了状態」に戻ります。



6. プロセスが`SIGSTOP`または`SIGTSTP`のいずれかのシグナルを受け取ると、プロセスの状態は「停止状態」に設定されます。この状態のプロセスはスケジュールから外れますが、システムリソースは解放されません。`SIGCONT`シグナルを受け取るまで、「準備完了状態」に戻りません。プロセスがトレースされている場合（デバッガでアプリケーションをデバッグしている場合など）、任意の

株式会社日昇テクノロジー

シグナルを受け取ると、プロセスの状態は`TASK_TRACED`状態になります。これは「停止状態」と同様で、`SIGCONT`シグナルを受け取るまでシステムのプロセススケジュールに参加しません。

7. プロセスがタスクを完了すると終了します。この時、プロセスの状態は「終了状態」になります。これは正常な終了を意味し、例えば`main`関数内で`return`するか、`exit`関数を呼び出すか、スレッドが`pthread_exit`関数を呼び出すことにより発生します。正常な終了を強調する理由は、プロセスが異常終了する場合もあるためです。例えば、プロセスが`kill`シグナルを受け取ると終了しますが、どのような終了であっても、最終的にカーネルは`do_exit`関数を呼び出してプロセスの状態を「ゾンビ状態」にします。ここでの「ゾンビ」とは、プロセスの PCB（Process Control Block、プロセス制御ブロック）を指します。プロセスがなぜ死んだのかについて、終了情報をプロセス制御ブロックに保存する理由は、他の人がどのように死んだのかを明確に知るためです。誰がそれに興味を持つのでしょうか？それはそのプロセスの親プロセスです。親プロセスがそれを起動した主な理由は、そのプロセスに何かをさせるためであり、今その子が死んでいるので、どのように事が運んだかを知る必要があります。したがって、これらの情報はプロセス制御ブロックに保存され、親プロセスがこれらの情報を確認するのを待っています。

8. 親プロセスがゾンビプロセスを処理するとき、このゾンビプロセスの状態は`EXIT_DEAD`、つまり「死亡状態」（終了状態）に設定されます。これにより、システムはゾンビプロセスのメモリスペースを回収できます。そうでなければ、システムにはますます多くのゾンビプロセスが存在し、最終的にシステムメモリが不足してクラッシュすることになります。では、親プロセスが忙しくてゾンビプロセスをすぐに処理できない場合はどうなるのでしょうか？また、子プロセスが「ゾンビ状態」になる前に親プロセスが先に終了した場合、その子プロセスはどうなるのでしょうか？最初のケースでは、プログラマーによって異なる処理方法が考えられます。親プロセスが他のことをしていて、ゾンビプロセスをすぐに処理できない場合、シグナルの非同期通知メカニズムを使用して、子プロセスがゾンビになったときに親プロセスにシグナルを送ることができます。親プロセスはこのシグナルを受

株式会社日昇テクノロジー

け取った後にそれを処理し、それまでの間は何をしていても構いません。もしプロセスの親プロセスが先に終了した場合、その子プロセスは「孤児プロセス」となります（親プロセスがいない）。その後、そのプロセスは祖先プロセスによって養子縁組されます。その祖先プロセスは`init`です（このプロセスはシステムの最初に実行されるプロセスで、カーネルの起動イメージファイルから直接ロードされた PCB を持っています。システムの他のすべてのプロセスは`init`プロセスの子孫です）。したがって、子プロセスが終了すると、`init`プロセスがこれらのリソースを回収します。

34.6 新しいプロセスの起動

Linux で新しいプロセスを起動する方法は複数あります。たとえば、system()関数を使用する方法やfork()関数を使用して新しいプロセスを起動（他の一部の Linux の書籍ではプロセスの生成と呼ばれますが、本書ではすべてをプロセスの起動と呼びます）する方法があります。最初の方法は比較的簡単ですが、使用する前に慎重に考慮する必要があります。なぜなら、それは効率が低く、無視できないセキュリティリスクを持つからです。2 番目の方法ははるかに複雑ですが、より良い柔軟性、効率、安全性を提供します。

34.6.1 `system()`関数によるプロセス実験

こ system()関数は C 標準ライブラリで提供されており、他のプログラムを簡単に呼び出す方法を提供します。system()関数を使用してアプリケーションを呼び出すと、シェルからそのプログラムを実行した場合と基本的に同じ結果が得られます。実際には、system()は/bin/sh を実行する子プロセスを開始し、そのコマンドを実行させます。/bin/sh はシェルの一種です。

例として、提供する system_programing/system ディレクトリ内の system.c ファイルを見てみましょう。それに含まれるアプリケーションルーチンは、system()関数を使用して`ls`という新しいプロセスを起動するものです。具体的なコードは以下の通りです：

株式会社日昇テクノロジー

リスト 1: system()を使用してプロセスを起動する

(system_programing/system/sources/system.c ファイル)

```
1 #include <sys/types.h>

2 #include <unistd.h>

3 #include <stdio.h>

4 #include <stdlib.h>

5

6 int main(void)

7 {

8     pid_t result;

9

10    printf("これはシステムデモです！¥n¥n");

11

12    /* system() 関数を呼び出す */

13    result = system("ls -l");

14

15    printf("完了しました！¥n¥n");

16

17    return result;

18 }
```

コードの第 13 行で、system()関数が呼び出され、"ls -l"コマンドが渡されます。これはシェルで実行した

株式会社日昇テクノロジー

場合と同じ結果になります。system()関数の戻り値は、呼び出されたシェルコマンドの戻り値です。システムのシェル自体が実行できない場合、system()関数は 127 を返します。他のエラーが発生した場合は、system()関数は-1 を返します。この例では、system 呼び出しが実際に機能するかどうかをチェックしていません。なぜなら、system()関数はシェルを介してコマンドを呼び出すため、システムのシェル自体の機能とセキュリティの脆弱性に制限されるため、この方法でプロセスを起動することは推奨されません。

system_programing/system ディレクトリには対応するコンパイル用の Makefile ファイルも提供されています。これは一般的な Makefile ファイルで、この章内のすべての例はこの Makefile ファイルを使用してコンパイルされます。Makefile ファイルの原理は前の章と同様なので、ここでは詳細は省略します。

34.6.1.1 実験操作

この例のコンパイルおよびテストプロセスは以下の通りです：

```
以下の操作は system_programing/system コードディレクトリで行います。
# X86 バージョンのプログラムをコンパイル
make
# X86 バージョンのプログラムを実行
./build_x86/system_demo
# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行うことができます。
make ARCH=ARM
# クロスコンパイルによって生成された armhf アーキテクチャのプログラムは、build_ARM ディレクトリにあります。開発ボードにコピーして実行することができます。
```

このプログラムは、system()関数を呼び出してプロセスを起動した結果を出力します。これは、シェル端末で`ls –l`コマンドを実行した結果と一致します。

株式会社日昇テクノロジー

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/
system_programing/system$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.system.o.d sources/system.c
 -o build_x86/system.o
gcc build_x86/system.o -o build_x86/system_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/
system_programing/system$ ./build_x86/system_demo
This is a system demo!
 合計使用量 12
drwxrwxr-x 2 flyleaf flyleaf 4096 11月  17 15:28 build_x86
-rw-rw-r-- 1 flyleaf flyleaf 1850 8月   12 15:02 Makefile
drwxrwxr-x 2 flyleaf flyleaf 4096 8月   12 15:02 sources
Done!

flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/
system_programing/system$ ls -l
 合計使用量 12
drwxrwxr-x 2 flyleaf flyleaf 4096 11月  17 15:28 build_x86
-rw-rw-r-- 1 flyleaf flyleaf 1850 8月   12 15:02 Makefile
drwxrwxr-x 2 flyleaf flyleaf 4096 8月   12 15:02 sources
```

プログラムの実行結果から、system()関数が完了するまで Done が出力されないことがわかります。これは、プログラムが上から下へ実行され、直接結果を返すことができないためです。system()関数は便利ですが、制限があります。なぜなら、プログラムは system()関数によって起動されたプロセスが終了するまで待たなければならず、そのために他のタスクをすぐに実行することができないからです。

もちろん、コマンドの終了位置に "&" を加えることで "ls -l" コマンドをバックグラウンドで実行させることができます。具体的なコマンドは以下の通りです：

```
ls -l &
```

このコマンドを system()関数内で使用する場合も、バックグラウンドで実行できます。その結果、シェルコマンドが終了するとすぐに system()関数の呼び出しが戻ります。これはバックグラウンドでプログラムを実行するリクエストであるため、ls プログラムが起動するとすぐにシェルが戻ります。system.c のソースコードを以下のように変更してください：

株式会社日昇テクノロジー

リスト 2：コマンドをバックグラウンドで実行するように変更

（/system_programing/system/sources/system.c ファイルに関連）

```
1 #include <sys/types.h>

2 #include <unistd.h>

3 #include <stdio.h>

4 #include <stdlib.h>

5

6 int main(void)

7 {

8 pid_t result;

9

10 printf("これはシステムデモです！¥n¥n");

11

12 /* system() 関数を呼び出す */

13 result = system("ls -l &");

14

15 printf("完了！¥n¥n");

16

17 return result;

18 }
```

make を再実行してプログラムをコンパイルし、実行すると、実験の現象は以下の通りです。

株式会社日昇テクノロジー

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/system$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.system.o.d sources/system.c -o build_x86/system.o
gcc build_x86/system.o -o build_x86/system_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/system$ ./build_x86/system_demo
This is a system demo!

Done!

flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/system$ 总用量 12
drwxrwxr-x 2 flyleaf flyleaf 4096 11月  17 15:41 build_x86
-rw-rw-r-- 1 flyleaf flyleaf 1850 8月   12 15:02 Makefile
drwxrwxr-x 2 flyleaf flyleaf 4096 8月   12 15:02 sources
^C
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/system$
```

ls コマンドがすべての出力結果を表示する前に、system()関数はすでに "完了" を出力して終了しました。system()プログラムが終了した後、ls コマンドは出力を完了します。このような処理方法は、しばしばユーザーに大きな混乱をもたらし、予想した結果と異なることがあります。したがって、プロセスを自分の意図に沿って実行させたい場合は、それらの振る舞いをより細かく制御する必要があります。次に、新しいプロセスを起動する他の方法について説明します。

34.6.2 fork()によるプロセス実験

前のセクションで、init`プロセスが子プロセスを起動する方法として、fork()関数を使用して元のプログラムから完全に独立した子プロセスを作成する方法について学びました。fork()関数の基本的な機能は子プロセスを起動することです。

fork()を呼び出した後、親プロセスには新しい子プロセスの PID が返されます。新しいプロセスは、元のプロセスと同じように実行を続けますが、子プロセス内で`fork()`を呼び出した後に返されるのは 0 です。これにより、親子プロセスは返された値を使用して、どちらが親プロセスでどちらが子プロセスかを判断できます。

株式会社日昇テクノロジー

fork()関数を使用して新しいプロセスを起動する本質は、親プロセスの内容をコピーして子プロセスを作成することです。これは細胞分裂のように、ほぼ同一の 2 つの細胞を得ることに似ています。そのため、起動された子プロセスは基本的には親プロセスのコピーですが、子プロセスと親プロセスにはいくつかの異なる点があります。それらの違いを以下に簡単に列挙します：

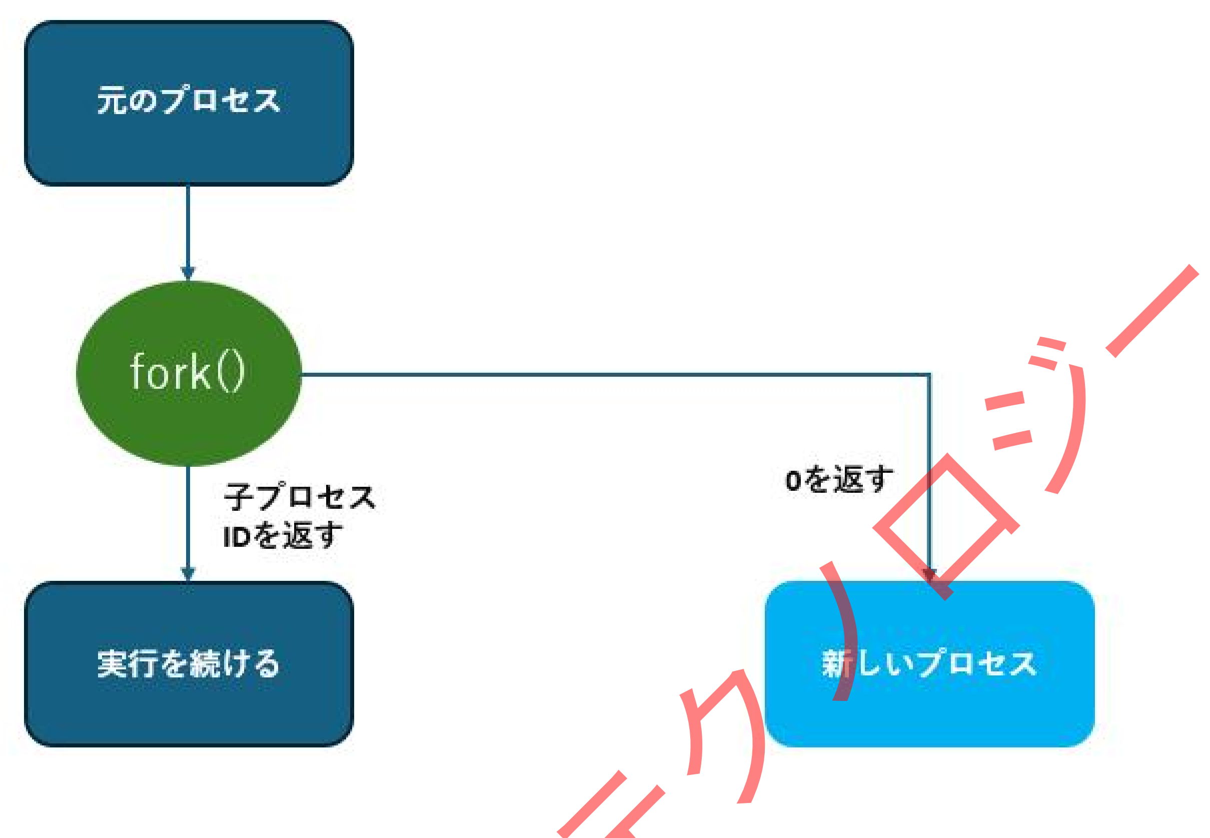
子プロセスと親プロセスが共有する内容：

- プロセスのアドレス空間
- プロセスのコンテキスト、コードセグメント
- プロセスのヒープ空間、スタック空間、メモリ情報
- プロセスの環境変数
- 標準 I/O のバッファ
- 開いたファイルディスクリプタ
- シグナルハンドラ
- 現在の作業ディレクトリ

子プロセス独自の内容：

- プロセス番号 PID（プロセスのユニークな識別子）

株式会社日昇テクノロジー



- ロック記録（親プロセスがファイルにかけたロックは継承されない）

- 保留中のシグナル（処理されていないシグナル、つまり「保留中」のシグナルも継承されない）

子プロセスは親プロセスの完全なコピーであるため、親子のプロセスは同じプログラムを実行しますが、このコピーには大きな問題があります。それは、リソースと時間の両方が大きく消費されることです。fork() システムコールを発行すると、カーネルは親プロセスの全アドレス空間をそのままコピーし、そのコピーを子プロセスに割り当てます。この行為は非常に時間がかかるもので、いくつかの作業が必要です：

- 子プロセスのページテーブル用にページを割り当てる。

- 子プロセスのページ用にページを割り当てる。

- 子プロセスのページテーブルを初期化する。

- 親プロセスのページを子プロセスの対応するページにコピーする。

株式会社日昇テクノロジー

アドレス空間を作成するこの方法は、多くのメモリアクセスを伴い、多くの CPU サイクルを消費し、高速キャッシュの内容を完全に破壊します。したがって、物理メモリを直接コピーすることは、システムへの負担が大きくなります。さらに重要なのは、ほとんどの場合、新しいプログラムをロードして実行を開始する多くの子プロセスでは、継承したアドレス空間が完全に破棄されるため、直接コピーすることは意味がないということです。そのため、Linux には Copy On Write（書き込み時コピー、COW と略される）という技術が導入されました。知っているように、Linux システム内のプロセスはすべて仮想メモリアドレスを使用しており、仮想アドレスと実際の物理アドレスの間には対応関係があります。各プロセスには独自の仮想アドレス空間があり、仮想アドレスを操作する方が物理メモリを直接操作するよりもはるかに簡単で迅速です。したがって、書き込み時コピーは、データのコピーを遅らせることさえ避けることができる技術です。カーネルはこの時、プロセスのアドレス空間全体をコピーするのではなく、親子プロセスが同じアドレス空間（ページ）を共有するようにします。

書き込み時コピーの考え方は、親プロセスと子プロセスがページを共有することにあり、ページをコピーするのではありません。共有されたページは変更できないため、親プロセスと子プロセスがいつ共有ページに内容を書き込もうとするときはいつでも、エラーが発生します。この時、カーネルはそのページを新しいページにコピーし、書き込み可能とマークします。元のページは依然として書き込み保護されています。他のプロセスが書き込みを試みると、カーネルはそのページの唯一の所有者であるかどうかをチェックし、もし所有者であれば、そのプロセスに対して書き込み可能とマークします。

要するに、書き込み時コピーは、実際に書き込みが発生するまでアドレス空間のコピーを遅らせる技術であり、これにより各プロセスが独自のアドレス空間を持つことができます。リソースのコピーは、書き込みが必要な時にのみ行われ、それまでの間、親プロセスと子プロセスはページを読み取り専用で共有します。この技術により、アドレス空間上のページのコピーが実際に書き込みが行われるまで遅らされます。そして、ほとんどの場合、共有されたページは書き込まれることはありません。たとえば、fork()関数の呼び出し後にすぐに exec()を実行する場合、アドレス空間をコピーする必要はありません。これにより、

株式会社日昇テクノロジー

fork()の実際のコストは、親プロセスのページテーブルをコピーし、子プロセスにプロセス記述子を作成することだけになります。

理論に関する知識はこれで十分です。次に、fork()関数の使用法を見てみましょう。この関数のプロトタイプは以下の通りです：

```
pid_t fork(void);
```

fork()によって新しいプロセスが開始されると、子プロセスと親プロセスは並行して実行を開始します。どちらが先に実行されるかは、カーネルのスケジューリングアルゴリズムによって決定されます。fork()関数が成功してプロセスを開始した場合、親子プロセスにそれぞれ一度ずつ戻ります。親プロセスには子プロセスの PID が、子プロセスには 0 が返されます。もし fork()関数が子プロセスの開始に失敗した場合、-1 が返されます。失敗の原因は通常、親プロセスが持つことができる子プロセスの数が規定の限界（CHILD_MAX）を超えたためであり、この時 errno は EAGAIN に設定されます。プロセステーブルに新しいエントリを作成するための十分な空間がない、または仮想メモリが不足している場合、errno 変数は ENOMEM に設定されます。

このセクションで提供される例は、system_programing/fork ディレクトリ下の fork.c にあり、fork()関数を使用して新しいプロセスを開始し、プロセス内で関連情報を印刷する例です。たとえば、親プロセスでは"In father process!!"などの情報が印刷されます。例のソースコードは以下の通りです。

リスト 3: fork()を使用してプロセスを起動する（関連コードリポジトリ /system_programing/fork/sources/fork.c ファイル）

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
```

株式会社日昇テクノロジー

```
5 int main(void)

6 {

7 pid_t result;

8

9 printf("これは fork デモです！¥n¥n");

10

11 /* fork()関数を呼び出す */

12 result = fork();

13

14 /* result の値によって fork()関数の戻り値を判断し、まずはエラー処理を行う */

15 if(result == -1) {

16 printf("Fork エラー¥n");

17 }

18

19 /* 戻り値が 0 の場合は子プロセス */

20 else if (result == 0) {

21 printf("戻り値は%d です。子プロセス内です!! 私の PID は%d です。¥n", result, getpid());

22

23 }

24

25 /* 戻り値が 0 より大きい場合は親プロセス */

26 else {
```

株式会社日昇テクノロジー

```
27 printf("戻り値は%d です。親プロセス内です!! 私の PID は%d です。¥n", result, getpid());

28 }

29

30 return result;

31 }
```

このコードを分析すると：

- 最初に第 12 行で fork 関数が呼び出され、この関数の呼び出しによって子プロセスが起動されます。子プロセスと親プロセスは同じ内容（コードセグメント）を実行しますが、fork()関数の戻り値によって判断できます。

- result の値が-1 の場合、fork()関数の実行に失敗したことを意味します。

- 戻り値が 0 の場合、コードを実行しているのは子プロセスです。その場合、戻り値、"In child process!!"、および子プロセスの PID を印刷します。プロセスの PID は getpid()関数で取得できます。

- 戻り値が 0 より大きい場合、コードを実行しているのは親プロセスです。同様に、戻り値、"In father process!!"、および親プロセスの PID を印刷します。

34.6.2.1 実験操作

fork 例のコンパイルとテストプロセスは以下の通りです：

```
# 以下の操作は system_programing/fork コードディレクトリで行います

# X86 バージョンのプログラムをコンパイル

make

# X86 バージョンのプログラムを実行

./build_x86/fork_demo
```

株式会社日昇テクノロジー

```
# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います

make ARCH=ARM

# クロスコンパイルで生成された armhf アーキテクチャのプログラムは`build_ARM`ディレクトリにあります。

# 開発ボードにコピーして実行します。
```

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/fork$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.fork.o.d sources/fork.c -o build_x86/fork.o
gcc build_x86/fork.o -o build_x86/fork_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/fork$ ./build_x86/fork_demo
This is a fork demo!

The returned value is 2315, In father process!! My PID is 2312

The returned value is 0, In child process!! My PID is 2315

flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/fork$ ./build_x86/fork_demo
This is a fork demo!

The returned value is 37476, In father process!! My PID is 37473

The returned value is 0, In child process!! My PID is 37476
```

この実験で親プロセスの戻り値が子プロセスの PID であること、子プロセスの戻り値が 0 であることに気づくでしょう。また、子プロセスは`fork()`関数より前の内容を再実行しませんが、`fork()`関数以降の内容は親プロセスと子プロセスの両方で実行されます。

34.6.3 exec 系列関数によるプロセス実験

実際、fork()関数を使用して子プロセスを起動するだけではあまり意味がありません。なぜなら、子プロセスは親プロセスと全く同じであり、子プロセスができることは親プロセスもできるからです。そのため、世界中の開発者は子プロセスに異なる作業をさせようと考え、exec 系列関数が生まれました。この関数群は、指定されたファイル名またはディレクトリ名で実行可能ファイルを見つけ、それを使用して元の呼び

株式会社日昇テクノロジー

出しプロセスのデータセグメント、コードセグメント、スタックセグメントを置き換えることで、プロセスの実行プログラムを置き換えます。実行完了後、元の呼び出しプロセスの内容はプロセス番号以外、新しいプログラムの内容に完全に置き換えられます。これらの実行可能ファイルは、バイナリファイルまたは Linux での任意の実行可能スクリプトファイルである可能性があります。簡単に言えば、プロセスの上書きです。

34.6.3.1 実験分析

execl()関数を使用した実験を通じて説明します：

```
int execl(const char *path, const char *arg, ...)
```

execl()関数は、引数`path`文字列で指定されたファイルパス（パスを指定する必要があります）を実行し、可変長引数を使用して、そのファイルを実行する際に渡される`argv[0]`、`argv[1]`...`argv[n]`を指定します。最後の引数は終了のマークとして NULL ポインタを使用する必要があります。

リスト 4: `execl()`関数の例（`system_programing/exec/sources/exec.c`ファイル）

```
1 int main(void)

2 {

3 int err;

4

5 printf("this is a execl function test demo!¥n¥n");

6

7 err = execl("/bin/ls", "ls", "-la", NULL);

8

9 if (err < 0) {

10 printf("execl fail!¥n¥n");
```

株式会社日昇テクノロジー

```
11 }

12

13 printf("Done!¥n¥n");

14 }
```

このコードでは、execl()関数の引数リストを使用して`ls`コマンドプログラムを呼び出し、その後の引数をそのファイルの argv[0]、argv[1]...argv[n]として扱い、最後の引数には終了のマークとして NULL ポインタを使用しています。これは、端末で"ls -la"を実行した場合と同じ結果を生み出します。

この関数の例のコードは`system_programing/exec/sources/exec.c`ファイルにあり、以下のコマンドでコンパイルおよびテストを行うことができます：

```
# 以下の操作は`system_programing/exec`コードディレクトリで行います

# X86 バージョンのプログラムをコンパイル

make

# X86 バージョンのプログラムを実行

./build_x86/exec_demo



# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います

make ARCH=ARM

# クロスコンパイルで生成された armhf アーキテクチャのプログラムは`build_ARM`ディレクトリにあります。

# 開発ボードにコピーして実行します。
```

プログラムは最初にメッセージ"this is a execl function test demo!"を出力し、次に exec 系列関数（実験では execl()関数を使用）を呼び出します。この関数は/bin/ls ディレクトリで ls プログラムを検索し、そ

株式会社日昇テクノロジー

れによって exec_demo 自体のプロセスを置き換えます。プログラムの実行結果は、以下に示すシェルコマンドを端末で使用した場合と同じです。

注意：exec 系列関数は、現在のプロセスを直接置き換えるもので、exec 系列関数を呼び出した後、現在のプロセスは続行されません。そのため、例示プログラムの「Done!」は出力されません。なぜなら、現在のプロセスは既に置き換えられているからです。通常、exec 系列関数はエラーが発生しない限り戻りません。エラーが発生した場合、exec 系列関数は-1 を返し、errno 変数にエラーが設定されます。

合計使用量合計使用量

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/syste
m_programing/exec$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.exce.o.d sources/exce.c -o build
_x86/exce.o
gcc build_x86/exce.o -o build_x86/exec_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/syste
m_programing/exec$ ./build_x86/exec_demo
this is a execl function test demo!

 合計使用量  20
drwxrwxr-x  4 flyleaf flyleaf 4096 11月  20 10:40 .
drwxrwxr-x 27 flyleaf flyleaf 4096 11月   9 09:58 ..
drwxrwxr-x  2 flyleaf flyleaf 4096 11月  20 10:40 build_x86
-rw-rw-r--  1 flyleaf flyleaf 1848 8月   12 15:02 Makefile
drwxrwxr-x  2 flyleaf flyleaf 4096 8月   12 15:02 sources
```

したがって、fork()を呼び出して子プロセスを複製し、その子プロセス内で exec 系列関数を呼び出して子プロセスを置き換えることができます。これにより、fork()と exec 系列関数を組み合わせることで、新しいプロセスを作成するために必要なすべてが得られます。

exec ファミリーには 6 つの異なる exec 関数が実際に含まれており、それらの機能は同じで、主に引数の渡し方が異なります。関数のプロトタイプは以下の通りです：

```
- int execl(const char *path, const char *arg, ...)

- int execlp(const char *file, const char *arg, ...)

- int execle(const char *path, const char *arg, ..., char *const envp[])

- int execv(const char *path, char *const argv[])
```

株式会社日昇テクノロジー

- int execvp(const char *file, char *const argv[])

- int execve(const char *path, char *const argv[], char *const envp[])

これらの関数は、execl、execlp、execle は子プログラムへの可変個数の引数を渡すことができ、例えば「ls -la」の例では、「-la」が子プログラム「ls」の引数です。execv、execvp、execve は引数を配列で装填し、子プログラムへ渡します。いずれの形式でも、引数は NULL ポインタで終了します。

つまり、それらのサフィックスによって彼らの機能を区別することができます：

- l 文字を含む関数（execl、execlp、execle）は、呼び出しプログラムの引数として「list」形式の引数リストを受け取ります。

- p 文字を含む関数（execvp と execlp）は、プログラム名を引数として受け取り、現在の実行パスと環境変数「PATH」でそのプログラムを検索して実行します（つまり、相対パスが使用可能です）；p 文字を含まない関数は、プログラムの完全なパスを指定する必要があります（つまり、絶対パスが必要です）。

- v 文字を含む関数（execv、execvp、execve）は、子プログラムの引数を「vector」形式の配列で装填します。

- e 文字を含む関数（execve と execle）は、他の関数よりも環境変数リストを指定する追加の引数を受け取り、新しいプログラムの環境変数として文字列配列を envp 引数を通じて渡すことができます。この envp 引数は、NULL ポインタで終了する文字列配列であり、各文字列は「environment = variables」の形式であるべきです。

これらの関数の使い方については、exec.c ファイルの例示コードを直接見て実験することができますので、ここでは詳細には触れません。

34.7 プロセスの終了

Linux システムでは、プロセスの終了（またはプロセスの終了と呼ばれますが、統一のために「終了」という言葉を使用します）には 5 つの一般的な方法があります。これらは正常終了と異常終了に分けること

株式会社日昇テクノロジー

ができます：

正常終了

- main 関数からのリターン。

- exit()関数の呼び出しによる終了。

- _exit()関数の呼び出しによる終了。

異常終了

- abort()関数の呼び出しによる異常終了。

- システムシグナルによる終了。

Linux システムでは、exit()関数は stdlib.h に、_exit()は unistd.h に定義されています。exit()と`_exit()`関数はどちらもプロセスを終了させるために使用されますが、プログラムが exit()または_exit()関数に達すると、プロセスは残りのすべての操作を無条件に停止し、PCB を含むさまざまなデータ構造をクリアし、現在のプロセスの実行を終了します。しかし、これら 2 つの関数には違いがあります。具体的な違いは以下の通りです：

実行中のプロセス

_exit()

exit()

終了処理プログラムを呼び出す

I/Oバッファをクリアする

カーネルのシステムコール

プロセスを終了する

株式会社日昇テクノロジー

画像から分かるように、_exit()関数は最もシンプルな作用を持ちます。直接システムコールを使用してプロセスの実行を終了させ、プロセス終了時にはそのプロセスが使用していたメモリ領域をクリアし、カーネル内のさまざまなデータ構造を破棄します。一方、exit()関数はこれらの基本的な動作にいくつかの手順を加えています。例えば、exit システムコールを呼び出す前に、ファイルが開かれている状況をチェックし、ファイルバッファ内の内容をファイルに書き戻します。これは「I/O バッファのクリア」と呼ばれます。

Linux の標準関数ライブラリには、「バッファリング I/O（buffered I/O）」操作と呼ばれるものがあり、

これは各開かれたファイルに対してメモリ内に一つのバッファ領域が存在することが特徴です。ファイルを読む際には、連続した複数のレコードを一度に読み出し、次回ファイルを読む時には直接メモリのバッファ領域から読み出すことができます。同様に、ファイルを書く際にも、実際にはメモリ内のバッファ領域に書き込むだけで、一定の条件（一定量に達するか特定の文字に遭遇するなど）が満たされた時にのみ、バッファ領域の内容を一度にファイルに書き込みます。

この技術はファイルの読み書き速度を大幅に向上させますが、プログラミングにいくつかの問題をもたらします。例えば、プログラムがファイルに書き込まれたと考えているデータが、特定の条件を満たしていないために、実際にはバッファ領域内にのみ保存されている場合があります。この時に_exit()関数を直接使ってプロセスを閉じると、バッファ領域のデータは失われます。したがって、データの完全性を保証したい場合は、exit()関数を使用する必要があります。

どのような終了方法であれ、システムは最終的にカーネル内の同一のコードを実行します。このコードは、プロセスが使用していたファイルディスクリプタを閉じ、使用していたメモリやその他のリソースを解放します。

では、_exit()と exit()関数の使用方法を見てみましょう：

ヘッダーファイル

```
#include <unistd.h>

#include <stdlib.h>
```

関数プロトタイプ：

```
void _exit(int status);

void exit(int status);
```

これらの関数は、プロセスが終了するときの状態コードを示す`status`パラメータを受け取ります。`0`は正常終了を意味し、他の非`0`値は異常終了を意味します。一般的に、`-1`または`1`を使用してリスト現されます。標準 C には`EXIT_SUCCESS`と`EXIT_FAILURE`の 2 つのマクロがあり、正常終了と異常

株式会社日昇テクノロジー

終了をリストします。

これらの関数の使用は非常にシンプルで、終了する必要がある場所で呼び出すだけです。ここでは詳細な説明は省略します。

34.8 プロセスの待機

Linux では、fork()関数を使用して子プロセスを起動すると、子プロセスは独自のライフサイクルを持ち、独立して実行されます。時には、親プロセスが子プロセスがいつ終了するか、または子プロセスの終了状態を知りたい、あるいは子プロセスの終了を待ちたい場合があります。これらの場合、親プロセスで wait()関数や waitpid()関数を呼び出して、子プロセスの終了を待つことができます。

前のセクションで、プロセスが exit()を呼び出した後、プロセスがすぐに完全に消えるわけではなく、ゾンビプロセスになることを学びました。ゾンビプロセスは非常に特殊なプロセスで、ほとんどすべてのメモリスペースを放棄し、実行可能なコードがなく、スケジューリングされることもありません。ただし、プロセスリストに位置を保持し、そのプロセスの終了状態などの情報を記録して他のプロセスが収集できるようにします。いずれにせよ、親プロセスはこれらのゾンビプロセスを回収し、wait()または waitpid()関数を呼び出してゾンビプロセスが占有していたメモリスペースを解放し、プロセスの終了状態情報を得る必要があります。

34.8.1 wait()関数

wait()関数のプロトタイプは以下の通りです：

```
pid_t wait(int *wstatus);
```

wait()関数は呼び出されると、子プロセスが終了するか信号が到着するまで親プロセスの実行を一時停止します。wait()関数が呼び出された時に子プロセスが既に終了している場合は、子プロセスの終了状態がすぐに返されます。子プロセスの終了状態情報は wstatus パラメーターによって返され、同時に子プロセスの PID も返されます。子プロセスの終了状態に興味がない場合は、wstatus を NULL に設定できます。

株式会社日昇テクノロジー

wait()関数に関して注意すべき点は以下の通りです：

- wait()は fork()とセットで使用されるべきです。fork()を使用せずに wait()を呼び出すと、wait()の戻り値は-1 になります。
- wstatus は終了したプロセスの状態を保存するために使用されますが、子プロセスの終了方法に興味がなく、ゾンビプロセスを回収するだけの場合は、このパラメータを NULL に設定できます。

子プロセスの終了状態を判断するためのマクロも Linux システムによって提供されています：

- WIFEXITED(status)：子プロセスが正常に終了した場合、非ゼロ値を返します。
- WEXITSTATUS(status)：WIFEXITED が非ゼロの場合、子プロセスの終了コードを返します。
- WIFSIGNALED(status)：子プロセスがシグナルによって終了した場合、非ゼロ値を返します。
- WTERMSIG(status)：WIFSIGNALED が非ゼロの場合、シグナルのコードを返します。
- WIFSTOPPED(status)：子プロセスが停止した場合、非ゼロ値を返します。
- WSTOPSIG(status)：WIFSTOPPED が非ゼロの場合、シグナルのコードを返します。

34.8.1.1 実験分析

wait()関数の使用例は以下の通りです：

リスト 5: `wait()`関数の例（`system_programing/wait/sources/wait.c`ファイル）

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
```

株式会社日昇テクノロジー

```
7 int main()

8 {

9 pid_t pid, child_pid;

10 int status;

11

12 pid = fork(); //(1)

13

14 if (pid < 0) {

15 printf("fork エラー¥n");

16 }

17 /* 子プロセス */

18 else if (pid == 0) { //(2)

19

20 printf("私は子プロセスです！私の pid は%d です！¥n¥n", getpid());

21

22 /* 子プロセスを 3 秒間停止 */

23 sleep(3);

24

25 printf("プロセスを終了しようとしています！¥n¥n");

26

27 /* 子プロセスを正常に終了 */

28 exit(0); //(3)
```

株式会社日昇テクノロジー

```
29 }

30 /* 親プロセス */

31 else { //(4)

32

33 /* wait を呼び出し、親プロセスをブロック */

34 child_pid = wait(&status); //(5)

35

36 /* 子プロセスの終了を検出したら、対応する状況を出力 */

37 if (child_pid == pid) {

38 printf("終了した子プロセスの ID を取得：%d¥n", child_pid);

39 printf("子プロセスの終了ステータスを取得：%d¥n¥n", status);

40 } else {

41 printf("何らかのエラーが発生しました。¥n¥n");

42 }

43

44 exit(0);

45 }

46 }
```

このコードを分析すると：

- (1)：まず、fork()関数を呼び出して子プロセスを起動します。

- (2)：fork()関数の戻り値 pid が 0 の場合、これは子プロセスが実行中であることを意味します。そのため、子プロセスではメッセージを出力し、3 秒間スリープします。

株式会社日昇テクノロジー

- (3)：スリープが終了した後、exit()関数を呼び出して終了します。終了状態は 0 で、子プロセスが正常に終了したことを意味します。

- (4)：fork()関数の戻り値 pid が 0 ではない場合、これは親プロセスが実行中であることを意味します。そのため、親プロセスでは wait(&status)関数を呼び出して子プロセスの終了を待ちます。子プロセスの終了状態は status 変数に保存されます。

- (5)：子プロセスが終了したことを検出すると wait()関数の戻り値によって子プロセスの pid を判断）、子プロセスの pid と status に関連する情報を出力します。

この例のコードは system_programing/wait ディレクトリ内の wait.c ファイルにあり、wait()関数と waitpid()関数の例が含まれています。マクロを切り替えることで、それぞれの関数を試すことができます。

```
# 以下の操作は system_programing/wait コードディレクトリで実行します
# X86 バージョンのプログラムをコンパイル
make
# X86 バージョンのプログラムを実行
./build_x86/wait_demo

# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルします
make ARCH=ARM
# クロスコンパイルで生成された armhf アーキテクチャのプログラムは`build_ARM`ディレクトリにあります。
# 開発ボードにコピーして実行できます。
```

実行結果は以下の通りです。

株式会社日昇テクノロジー

34.8.2 waitpid()

waitpid()関数は wait()関数と同じ機能を持ちますが、最初に終了した子プロセスを待つ必要はなく、特定の pid の子プロセスを指定して待つ、非ブロッキング版の wait()機能を提供するなど、他にもオプションがあります。実際には、wait()関数は waitpid()関数の特別なケースに過ぎません。Linux の内部実装では、wait 関数は直接 waitpid 関数を呼び出しています。

関数プロトタイプ：

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_prog
raming/wait$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.wait.o.d sources/wait.c -o build_x86/w
ait.o
gcc build_x86/wait.o -o build_x86/wait_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_prog
raming/wait$ ./build_x86/wait_demo
I am a child process!, my pid is 11383!

I am about to quit the process!

Get exit child process id: 11383
Get child exit status: 0
```

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

waitpid()関数には 3 つのパラメータがあり、以下のように説明されています：

- pid：待つ子プロセスの ID で、具体的な意味は以下の通りです：

- pid < -1：pid の絶対値に等しいプロセスグループ ID を持つ任意の子プロセスを待ちます。

- pid = -1：任意の子プロセスを待ちます。この場合、waitpid()関数は wait()関数と同等になります。

- pid = 0：現在のプロセスと同じプロセスグループ ID を持つ任意の子プロセスを待ちます。

- pid > 0：指定されたプロセス ID を持つ子プロセスを待ちます。

- wstatus：wait()関数と同様です。

- options：waitpid()関数の動作を制御するための追加オプションを提供します。これらのオプションを使用しない場合は、このパラメータを 0 に設定できます。

株式会社日昇テクノロジー

- WNOHANG：指定された子プロセスが終了していない場合、`waitpid()`関数は 0 を直ちに返し、この関数上でブロックされることはありません。子プロセスが既に終了している場合は、その子プロセスのプロセス ID と状態情報を直ちに返します。

- WCONTINUED：子プロセスが SIGCONT シグナルによって実行を再開した場合にも直ちに返ります（これはあまり使用されませんが、知っておくと良いでしょう）。

- WUNTRACED：子プロセスが停止状態になった場合（例えば、デバッグ中など）、直ちに返ります。

明らかに、waitpid()関数のパラメータが(子プロセスの pid、status、0)の場合、waitpid()関数は wait()関数と全く同じになります。

wait.c の例のファイルには、waitpid 関数の例が含まれており、wait()関数の実験現象と似ています。

第 35 章 信号

Linux カーネルは、プロセス間通信のために信号、パイプ、メッセージキューなど様々なカーネルオブジェクトを提供しています。本章では、Linux カーネルの信号オブジェクトについて説明します。

35.1 信号の基本概念

35.1.1 概要

信号（signal）、またはソフトウェア割り込みとも呼ばれ、プロセスに非同期イベントが発生したことを通知するために使用されます。これは、Linux システムが特定の条件に応じて生成するイベントであり、ソフトウェアレベルでの割り込みメカニズムの一種のシミュレーションであり、非同期通信方式です。原理的には、プロセスが信号を受信することと、プロセッサが割り込み要求を受信することは同じと言えます。

信号は、プロセス間通信メカニズムにおける唯一の非同期通信メカニズムであり、プロセスは信号の到着を待つ必要がなく、実際には、プロセスは信号がいつ到着するかを知りません。中断サービス関数と同

株式会社日昇テクノロジー

様に、中断が発生すると中断サービス関数に入り、処理を行います。同様に、プロセスが信号を受信すると、それに応じて一定のアクションを取ります。"生成（raise）"は信号の発生を意味し、"捕捉（catch）"はプロセスが信号を受信したことを意味します。

Linux システムでは、信号はシステムのいくつかのエラーによって生成されることもあれば、プロセスが明示的に生成することもあります。例えば、メモリセグメントの衝突、浮動小数点プロセッサエラー、不正な命令など、シェルや端末エミュレータによって生成され、中断を引き起こします。プロセス間で通知を送信したり、動作を変更したりする手段として、プロセスによって明示的に生成される信号もあります。いずれの場合も、それらは生成され、捕捉され、応答され、または無視されることができます。プロセス間で相互に信号を送信することができ、カーネルも内部イベントによってプロセスに信号を送信して、何らかのイベントが発生したことを通知することができます。

35.1.2 システムがサポートする信号

kill -l コマンドを使用して、システムがサポートする信号の種類を確認することができます。

```
@embedfire_dev:~/workdir/base_code/application/process$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
@embedfire_dev:~/workdir/base_code/application/process$
```

Linux システムは 62 種類の信号をサポートしており、各信号名は SIG という 3 文字で始まります。ただし、32 と 33 の信号値は存在しません。

株式会社日昇テクノロジー

これら 62 種類の信号は、1～31 の信号値を持つ信号は非リアルタイム信号（または信頼できない信号）であり、UNIX システムから継承されたものです。34～64 の信号値を持つ信号はリアルタイム信号（または信頼できる信号）です。

特に注意すべき点は以下の通りです：

- 信号の「値」は x86、PowerPC、ARM プラットフォームで有効ですが、他のプラットフォームではこのリストとは異なる場合があります。

- 「説明」に記載されているいくつかの状況が発生すると、対応する信号が生成されますが、それが必ずしもそのイベントが発生したことを意味するわけではありません。実際には、任意のプロセスが kill(`関数を使用して任意の信号を生成することができます。

- SIGKILL と SIGSTOP は特別な信号であり、無視、ブロック、または捕捉することはできず、デフォルトのアクションに従って応答する必要があります。

- 一般的に、信号の応答処理は次のように行われます：その信号がブロックされている場合、その信号は保留され、何の処理も行われず、ブロックが解除されるまで待ちます。その信号が捕捉されている場合、捕捉のタイプに応じてさらに判断され、応答関数が設定されている場合はその関数を実行し、無視が設定されている場合はその信号を直接破棄します。最後に信号のデフォルト処理が実行されます。

35.1.3 "非リアルタイム信号とリアルタイム信号"

Linux システムには多くのシグナルがあり、その中で最初の 31 個のシグナルには特別な名前が付けられ、特別なイベントに対応しています。例えば、シグナル値が 1 の SIGHUP（Signal Hang UP）は、システムに中断を閉じるよう通知するシグナルで、システムの制御端末が閉じられた（つまり、ハングアップした）時にこのシグナルが生成されます。

シグナル値が 1～31 のシグナルは非リアルタイムシグナルで、このクラスのシグナルはキューイングを

株式会社日昇テクノロジー

サポートしていないため、シグナルが失われる可能性があります。例えば、同じシグナルを複数回送信しても、プロセスは一度しか受け取れず、一度しか処理されないため、残りの同じシグナルは破棄されます。一方、リアルタイムシグナル（シグナル値が 34～64 のシグナル）はキューイングをサポートしているため、プロセスに送信されたシグナルの数だけ処理されます。

シグナルに信頼性と不信頼性の区別があるのは、シグナルの処理プロセスに由来します。一般に、プロセスがシグナルを受信した後、すぐには処理されず、適切なタイミングで処理されます。通常は、割り込みが戻る時や、カーネルモードからユーザーモードに戻る時に処理されます（このケースが一般的な処理方法です）。

つまり、これらのシグナルが到着しても、プロセスがすぐに処理するわけではなく、システムは現在実行中のプロセスを一時停止してシグナルを処理することはありません。これは、システムのリソース消費が大きすぎるためです。緊急のシグナルでない限り、すぐには処理されず、システムは一般にカーネルモードからユーザーモードに切り替える際にシグナルを処理します。例えば、プロセスが休眠状態にあるが、シグナルを受け取った場合、システムはそのシグナルをプロセスの唯一の PCB（プロセス制御ブロック）に保存する必要があります。

非リアルタイムシグナルはキューイングをサポートしていないため、この時に別のシグナルが到着した場合、それは破棄され、プロセスはそのシグナルを処理できません。そのため、これは不信頼性のあるものです。リアルタイムシグナルにはこのような心配はなく、キューイングをサポートしているため、シグナルは破棄されず、各到着したシグナルは効果的に処理されます。

35.2 信号の処理

信号を生成するイベントは一般的に三つの大きなカテゴリに分けられます：プログラムエラー、外部イベント、明示的なリクエストです。

- プログラムエラー：ゼロ除算、不正なストレージアクセスなどがあります。このタイプの状況は通常、Linux カーネルではなく、ハードウェアによって検出されますが、カーネルはこのエラーが発生した

株式会社日昇テクノロジー

プロセスに対応する信号を送信します。

- 外部イベント：ユーザーが端末で特定のキーを押した時に中断が生成する信号、プロセスが CPU またはファイルサイズの制限を超えた時に、カーネルがプロセスに通知するために信号を生成します。

- 明示的なリクエスト：プロセスが`kill()`関数を使用して、他のプロセスやプロセスグループに任意の信号を送信する場合です。

信号の生成は同期的または非同期的なものがあります。

- 同期信号は、プログラム実行中に特定のエラーが発生して生成されるものがほとんどです。プロセスが自分自身に明示的に要求して生成した信号も同期的です。

- 非同期信号は、受信プロセスの制御外のイベントによって生成される信号です。このタイプの信号は、プロセスが制御できないものであり、受動的に受け取るしかなく、信号がいつ発生するかを知ることはできません。外部イベントは常に非同期的に信号を生成し、非同期信号はプロセスの実行中の任意の時点で発生する可能性があります。プロセスは信号の到着時刻を予測することはできず、Linux カーネルに信号が生成された場合にどのようなアクションを取るかを指示することしかできません（これは信号に対する処理の登録に相当します）。

同期であれ非同期であれ、信号が発生したときには、Linux カーネルに以下の 3 つのアクションのいずれかを取るよう指示することができます：

- 信号を無視する。ほとんどの信号は無視できますが、SIGSTOP と SIGKILL は例外です。これら 2 つの信号を無視できない理由は、スーパーユーザーに任意のプロセスを終了させるまたは停止させる手段を提供するためです。さらに、他の信号は無視できますが、中には無視すべきでないものもあります。例えば、ハードウェア例外（不正命令）信号を無視すると、プロセスの挙動が不定になる可能性があります。

- 信号をキャッチする。この処理は Linux カーネルに、信号が現れたときに特定の関数を呼び出すよう指示します。この関数は信号処理関数と呼ばれ、信号を生成したイベントを処理します。

株式会社日昇テクノロジー

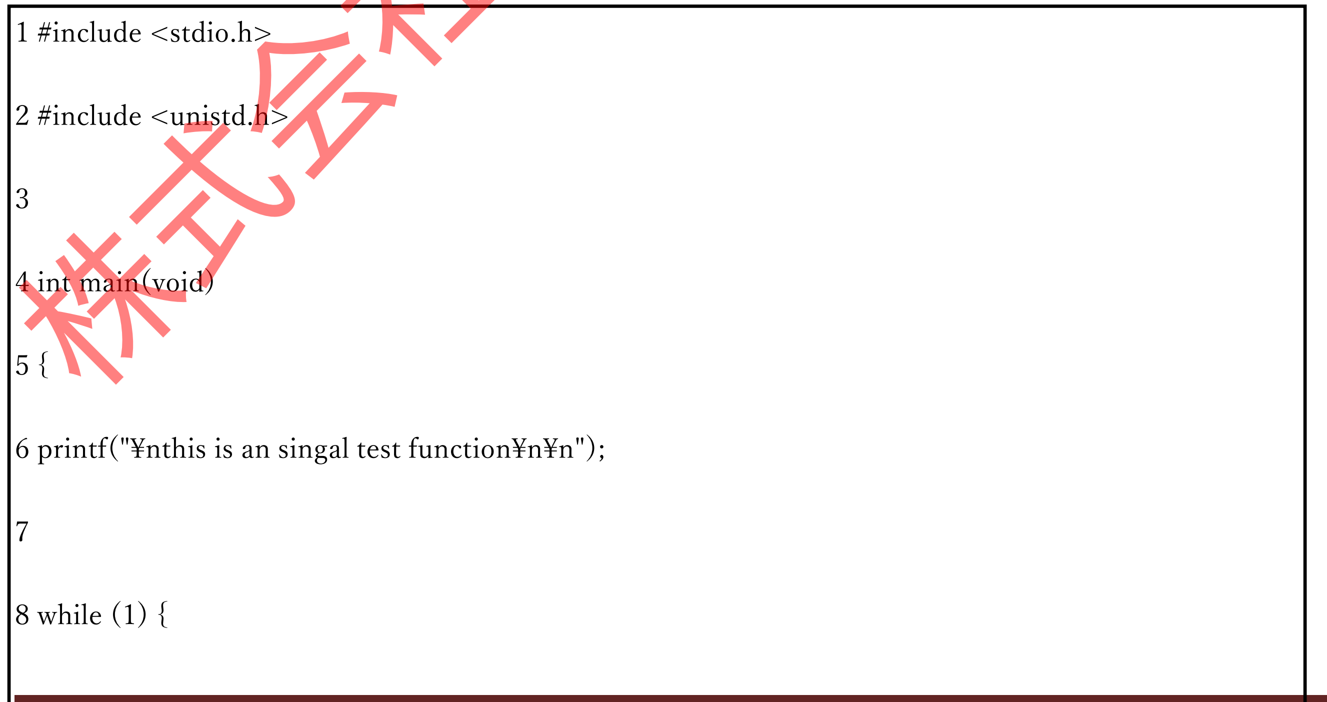
信号のデフォルトアクションを有効にする。システムは各信号に対してデフォルトアクションを定めており、このアクションは Linux カーネルによって完了される。以下のようなデフォルトアクションが考えられる：

- プロセスを終了させ、メモリダンプファイルを生成する。つまり、プロセスのアドレス空間の内容とレジスタコンテキストをプロセスの現在のディレクトリに core というファイル名で書き出す；
- プロセスを終了させるが、core ファイルは生成しない。
- 信号を無視する。
- プロセスを一時停止する。
- プロセスが一時停止状態の場合は、プロセスを再開する。そうでない場合は、信号を無視する。

35.2.1 実験分析

いくつかの制御信号を検証するための小さなプログラムを書くことができます。たとえば、SIGINT（プログラム終了信号）です。プログラムのコードは以下の通りです。

- リスト 1: デフォルト signal 例（対応するコードリポジトリ /system_programing/signal/sources/signal.c ファイル）



```
1 #include <stdio.h>

2 #include <unistd.h>

3

4 int main(void)

5 {

6 printf("¥nthis is an singal test function¥n¥n");

7

8 while (1) {
```

株式会社日昇テクノロジー

9 printf("waiting for the SIGINT signal , please enter ¥"ctrl + c¥"...¥n");

10 sleep(1);

```
11 }

12

13 exit(0);

14 }
```

この実例コードは system_programing/signal ディレクトリ下にあり、signal.c ファイルには上記のデフォルト信号処理の例と、次の小節で説明する signal handler を使用した手動処理の例が含まれています。マクロを使用して切り替えることができます。

```
以下の操作は system_programing/signal コードディレクトリで行います。

# X86 版プログラムをコンパイルする

make

# X86 版プログラムを実行する

./build_x86/signal_demo

# Ctrl+C キーを押して SIGINT シグナルを送信すると、プログラムは終了します



# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行うことができます

make ARCH=ARM

# クロスコンパイルによって生成された armhf アーキテクチャのプログラムは build_ARM ディレクトリにあり、

# それを開発ボードにコピーして実行することができます
```

コンパイル後の実行結果を見ると、端末が継続して「please input "Ctrl + C" to terminate the test

株式会社日昇テクノロジー

process!」と表示されます。Ctrl + C キーを押すと、このプロセスを終了できます。これは、Ctrl+C キーコンビネーションを押すと SIGINT 信号が生成され、プロセスが終了するためです。

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/signal$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.signal.o.d sources/signal.c -o build_x86/signal.o
gcc build_x86/signal.o -o build_x86/signal_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/signal$ ./build_x86/signal_demo

this is an alarm test function

waiting for the SIGINT signal , please enter "ctrl + c"...
waiting for the SIGINT signal , please enter "ctrl + c"...
waiting for the SIGINT signal , please enter "ctrl + c"...
^C
```

35.3 信号を捕捉するための関連 API 関数

上述の signal 例では、信号に対して特に処理を行っていません。システムがデフォルトの処理方法を使用してプログラムを終了させました。多くの場合、信号を使用するのは、プロセスを殺すためではなく、通知するためであり、殺す前にいくつかの後始末を行いたい場合があります。このような場合、信号を捕捉して処理する必要があります。Linux では、信号を捕捉するための関数がいくつかあります。たとえば、signal()、sigaction()などです。

35.3.1 signal()

signal()は主に信号を捕捉するために使用され、プロセス内の信号のデフォルト動作を変更することができます。この信号を捕捉した後、信号の処理方法を自定義できます。この信号を受信した後、どのように処理するかです。これは Linux 開発で最もよく必要とされる作業です。

signal()を使用する際には、事前にコールバック関数を設定する必要があります。つまり、プロセスが信号を受信した後に実行にジャンプする予定の応答関数です。または、特定の信号を無視する設定もできます。これにより、信号のデフォルト動作を変更することができます。このプロセスは「信号の捕捉」と呼ばれます。信号の「捕捉」は繰り返し行うことができますが、signal()関数は前回設定した信号応答関数ポ

株式会社日昇テクノロジー

インタを返します。

signal()関数の使用方法、およびその関数原型については、man コマンドで signal()に関する情報を参照できます。その原型は以下の通りです。

```
- typedef void (*sighandler_t)(int);

- sighandler_t signal(int signum, sighandler_t handler);
```

これは比較的複雑な関数定義で、signal が signum と handler の 2 つのパラメータを持つ関数であることを示しています。捕捉または無視する準備ができている信号は、パラメータ signum によって指定され、指定された信号を受信した後に呼び出される関数は、パラメータ handler によって指定されます。

signum は捕捉する信号の名前を指定します。無効な信号を指定した場合、または捕捉または無視できない信号（SIGKILL など）を処理しようとした場合、errno は EINVAL に設定されます。

handler は関数ポインタで、void(*sighandler_t)(int)型で、int 型のパラメータを持ちます。このパラメータの役割は、受信した信号値を伝達することで、戻り値の型は void です。

signal()関数は sighandler_t 型の関数ポインタを返します。これは、signal()関数を呼び出して信号の動作を変更すると、以前の信号処理動作が何であったかをアプリケーション層に知らせる必要があるためです。信号のデフォルト動作を正しく変更した場合は、対応するエラーコード SIG_ERR を返します。

handler には、信号の処理方法をユーザーが自定義する必要がありますが、以下のマクロを使用することもできます。

- SIG_IGN：その信号を無視します。

- SIG_DFL：システムのデフォルト方法で信号を処理します。

この関数は比較的シンプルですが、処理プログラムの呼び出しによって信号がブロックされた場合、処理プログラムから戻った後、信号はブロック解除されます。SIGKILL と SIGSTOP の信号は捕捉または無視することができません。

株式会社日昇テクノロジー

35.3.1.1 実験分析

この関数を使用して小さな実験を行うことができます。コードは以下の通りです。

リスト 2: signal()関数例（対応するコードリポジトリ

/system_programing/signal/sources/signal.c ファイル）

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <sys/types.h>
6 #include <sys/wait.h>
7
8 /** シグナル処理関数 */
9 void signal_handler(int sig) //(3)
10 {
11 printf("¥n このシグナル番号は %d です¥n", sig);
12
13 if (sig == SIGINT) {
14 printf("SIGINT を受け取りました!¥n¥n");
15 printf("シグナルがデフォルトの処理モードに戻されました!¥n¥n");
16 /** シグナルをデフォルト状態に復元 */
17 signal(SIGINT, SIG_DFL); //(4)
```

株式会社日昇テクノロジー

```
18 }

19

20 }

21

22 int main(void)

23 {

24 printf("¥n これはシグナルテスト関数です¥n¥n");

25

26 /** シグナル処理のコールバック関数を設定 */

27 signal(SIGINT, signal_handler); //(1)

28

29 while (1) {

30 printf("SIGINT シグナルを待っています。¥"ctrl + c¥" を入力してください...¥n");

31 sleep(1); //(2)

32 }

33

34 exit(0);

35 }
```

株式会社日昇テクノロジー

このコードを解析します（まずは 23 行目の main 関数から始めます）。

- (1) ：signal()関数を使用して SIGINT 信号（この信号は CTRL+C を押すことで発生します）を捕捉し、コールバック関数として signal_handler()を設定します。この関数は、信号が発生したときに呼び出され、その信号を処理します。

- (2) ：信号が到着しない間、情報を出力してスリープします。

- (3) ：signal_handler()は信号処理関数で、int 型の信号値を引数に取ります。この関数は、関数が呼び出された原因となった信号値を引数に取ります。複数の信号を同じ関数内で処理する必要がある場合、この引数は非常に便利です。

- (4) ：もし信号が SIGINT であれば、対応する情報を出力し、signal()関数を呼び出して SIGINT 信号の処理をデフォルト（SIG_DFL）に戻します。次に SIGINT 信号を受け取ると、この関数には入らずに直接プロセスが終了します。

この実例コードは system_programing/signal ディレクトリ下にあり、signal.c ファイルには上記のデフォルト信号処理の例と、本節の例のコードを切り替えるためのマクロが含まれています。

```
# コンパイルする前に、ソースファイルを開いて試したいコードへのマクロを切り替えてください！！！
# 以下の操作は system_programing/signal コードディレクトリで行います
# X86 バージョンのプログラムをコンパイル
make
# X86 バージョンのプログラムを実行
./build_x86/signal_demo
# Ctrl+C キーを押して SIGINT 信号を送信し、signal_handler がその信号を捕捉して情報を出力して処理します
# もう一度 Ctrl+C キーを押して SIGINT 信号を送信すると、Linux はデフォルトの方法で処理し、プロセスを終了します
# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います
make ARCH=ARM
# クロスコンパイルで生成された armhf アーキテクチャのプログラムは build_ARM ディレクトリにあります。
# 開発ボードにコピーして実行することができます。
```

株式会社日昇テクノロジー

実験現象は以下の通りです。"CTRL+C"を押すと、signal_handler()信号処理関数に入り、対応する情報を出力して SIGINT 信号の処理をデフォルトに戻します。そのため、次に"CTRL+C"を押すとプロセスは直接終了します。

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/s
_programing/signal$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.signal.o.d sources/signal.c
-o build_x86/signal.o
gcc build_x86/signal.o -o build_x86/signal_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/s
ystem_programing/signal$ ./build_x86/signal_demo

this is an singal test function

waiting for the SIGINT signal , please enter "ctrl + c"...
waiting for the SIGINT signal , please enter "ctrl + c"...
waiting for the SIGINT signal , please enter "ctrl + c"...
^C
this signal number is 2
I have get SIGINT!

The signal has been restored to the default processing mode!

waiting for the SIGINT signal , please enter "ctrl + c"...
waiting for the SIGINT signal , please enter "ctrl + c"...
waiting for the SIGINT signal , please enter "ctrl + c"...
^C
```

35.3.2 sigaction()

実際には、signal()関数インターフェイスの使用はお勧めしません。前の節でそれを紹介したのは、多くの古いプログラムでその使用法を見かけるかもしれないし、比較的簡単だからです。後ほど、より明確な定義と、より信頼性の高い実行を提供する sigaction()関数を紹介します。この関数の機能は signal()関数と同じですが、API インターフェイスが少し異なります。今後はすべてのプログラムでこの関数を使用して信号を操作することをお勧めします。

sigaction()関数のプロトタイプは次のとおりです：

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

この関数のパラメータは signal()関数よりも少し多いです。パラメータの違いは以下の通りです：

- signum：捕捉する信号値を指定します。

株式会社日昇テクノロジー

- act：構造体で、その内容は以下の通りです：

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

- sa_handler は関数ポインタで、シグナルをキャッチした後の処理関数です。int 型のパラメータを持ち、シグナルの値を渡します。この関数は標準のシグナル処理関数です。

- sa_sigaction は拡張シグナル処理関数で、関数ポインタですが、標準のシグナル処理関数よりもはるかに複雑です。実際、拡張インターフェイスを選択する場合、シグナルを受信するプロセスは int 型のシグナル値だけでなく、- siginfo_t 型の構造体ポインタと void 型のポインタも受け取ることができます。sa_handler と sa_sigaction を同時に使用しないように注意してください。これら二つの処理関数は共用体（ユニオン）の部分があります。siginfo_t 型の構造体については後ほど説明します。

- sa_mask はシグナルマスクで、シグナル処理関数の実行中にブロックされるシグナルのマスクを指定します。このマスクに設定されたシグナルは、プロセスがシグナルに応答している間、一時的にブロックされます。SA_NODEFER フラグを使用しない限り、現在処理中の応答シグナルが再度到着してもブロックされます。

- sa_restore は既に廃止されたメンバ変数で、使用しないでください。

- sa_flags はシグナル処理過程の振る舞いを変更するためのフラグの組み合わせを指定します。0 個以上の以下のフラグで構成されます：

* SA_NOCLDSTOP：signum が SIGCHLD の場合、子プロセスが停止または再開したときに、

株式会社日昇テクノロジー

sigaction()関数を呼び出したプロセスにシグナルを送信しません。つまり、SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU（停止）のいずれかを受け取ったとき、または SIGCONT（再開）を受け取ったときに、親プロセスに通知されません。このフラグは、SIGCHLD の処理プログラムを設定するときにのみ意味を持ちます。

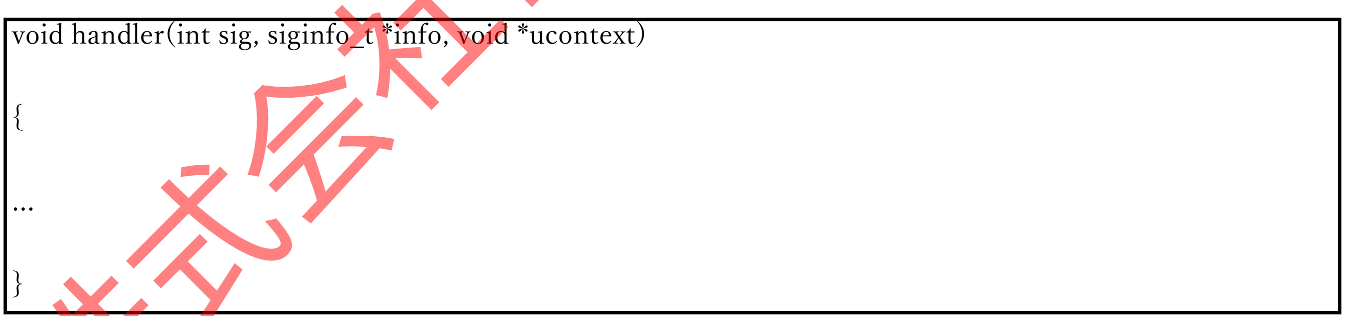
* SA_NOCLDWAIT：親プロセスが子プロセスの終了時に SIGCHLD シグナルを受け取らず、子プロセスがゾンビプロセスにならないことを示します。

* SA_NODEFER：自身のシグナル処理プログラムからシグナルを受け取るのをブロックしないようにし、プロセスによるシグナルのマスクを無効にします。つまり、シグナル処理関数の実行中にこのシグナルを引き続き受信できます。このフラグは、シグナル処理プログラムを設定するときにのみ意味を持ちます。

* `SA_RESETHAND`：シグナル処理の後、デフォルトの処理方式にリセットします。

* `SA_SIGINFO`：シグナル処理関数として sa_handler メンバーではなく sa_sigaction メンバーを使用することを示します。

sa_flags に SA_SIGINFO フラグを指定した場合、シグナル処理プログラムのアドレスは sa_sigaction フィールドを通じて渡されます。この処理プログラムは次のような 3 つのパラメータを取ります：



```
void handler(int sig, siginfo_t *info, void *ucontext)
{
...
}
```

info は siginfo_t へのポインタで、シグナルに関するさらに多くの情報を含む構造体です。具体的なメンバ変数は以下の通りです：

株式会社日昇テクノロジー

```
siginfo_t {
int si_signo; /* シグナル番号 */
int si_errno; /* エラー値 */
int si_code; /* シグナルコード */
int si_trapno; /* トラップ番号（ほとんどのアーキテクチャでは使用されていない） */
pid_t si_pid; /* シグナルを送信したプロセス ID */
uid_t si_uid; /* シグナルを送信した実際のユーザ ID */
int si_status; /* 終了値またはシグナル状態 */
clock_t si_utime; /* 使用されたユーザ時間 */
clock_t si_stime; /* 使用されたシステム時間 */
sigval_t si_value; /* シグナル値 */
int si_int; /* POSIX.1b シグナル */
void *si_ptr;
int si_overrun; /* タイマーのオーバーフローカウント */
int si_timerid; /* タイマー ID */
void *si_addr; /* 故障の原因となったメモリ位置 */
long si_band;
int si_fd; /* ファイルディスクリプタ */
short si_addr_lsb; /* 最低有効ビットアドレス（Linux 2.6.32 から存在） */
void *si_lower; /* アドレス競合の下限 */
void *si_upper; /* アドレス競合の上限（Linux 3.19 から存在） */
int si_pkey; /* 保護キーを引き起こした PTE */
void *si_call_addr; /* システムコール命令のアドレス */
```

株式会社日昇テクノロジー

```
int si_syscall; /* 試みられたシステムコールの数 */

unsigned int si_arch; /* 試みられたシステムコールのアーキテクチャ */

}
```

上記のメンバ変数の大部分は、シグナルをシンプルに処理する場合はほとんど使用されません。なぜなら、単純なシグナル処理の場合は sa_handler を使用するだけで十分であり、siginfo_t のような複雑な情報を設定する必要はありません。

- oldact：元のシグナル処理パラメータを返します。通常は NULL に設定します。

35.3.2.1 実験分析

sigaction は一見複雑に見えますが、直接ソースコードを分析することでより明確な理解を得ることができます。

リスト 3: sigaction の例（関連するコードリポジト

/system_programing/sigaction/sources/sigaction.c ファイル）

```
1 #include <unistd.h>

2 #include <stdio.h>

3 #include <stdlib.h>

4 #include <signal.h>

5 #include <sys/types.h>

6 #include <sys/wait.h>

7

8 /** シグナル処理関数 */

9 void signal_handler(int sig) //(1)
```

株式会社日昇テクノロジー

```
10 {
11     printf("¥n このシグナル番号は %d です¥n", sig);
12
13     if (sig == SIGINT) {
14         printf("SIGINT を取得しました！¥n¥n");
15         printf("シグナルは自動的にデフォルトハンドラに復元されました！¥n¥n");
16         /** シグナルは自動的にデフォルト処理関数に復元されます */
17     }
18
19 }
20
21 int main(void)
22 {
23     struct sigaction act;
24
25     printf("これは sigaction 関数テストデモです！¥n¥n");
26
27     /** シグナル処理のコールバック関数を設定 */
28     act.sa_handler = signal_handler; //(2)
29
30     /* シグナル集合をクリア */
31     sigemptyset(&act.sa_mask); //(3)
```

株式会社日昇テクノロジー

```
32
33   /** 処理後にデフォルトのシグナル処理に復元 */
34   act.sa_flags = SA_RESETHAND; //(4)
35
36   sigaction(SIGINT, &act, NULL); //(5)
37
38   while (1)
39   {
40       printf("SIGINT シグナルを待っています。¥"ctrl + c¥" を押してください...¥n¥n");
41       sleep(1);
42   }
43
44   exit(0);
45 }
```

- (1) ：シグナル処理関数`signal_handler()`は`signal`実験のシグナル処理関数とほぼ同じですが、この関数内でシグナルをデフォルト処理に復元することはありません。これは、`sa_flags`メンバ変数を設定し、シグナルを処理した後に自動的にデフォルトの処理に復元するためです。

- (2) ：シグナル処理のコールバック関数を設定し、この実験では`sa_handler`をシグナル処理メンバ変数として使用し、`sa_sigaction`ではありません。

- (3) ：`sigemptyset()`関数を呼び出してプロセスがブロックするシグナル集合をクリアします。つまり、シグナル処理中にどのシグナルもブロックしません。

- (4) ：`sa_flags`メンバ変数を`SA_RESETHAND`に設定し、処理後にデフォルトのシグナル処理に復元します。

株式会社日昇テクノロジー

- (5) ：`sigaction()`関数を呼び出して SIGINT シグナルをキャッチします。

この例のコードは system_programing/sigaction ディレクトリ下にあります。

```
# 以下の操作は system_programing/sigaction コードディレクトリで行います

# X86 版プログラムをコンパイル

make

# X86 版プログラムを実行

./build_x86/sigaction_demo

# Ctrl+C キーを押して SIGINT シグナルを送信すると、`signal_handler`がこのシグナルをキャッチして

情報を出力して処理します

# もう一度 Ctrl+C キーを押して SIGINT シグナルを送信すると、Linux はデフォルトの方法で処理し、

プロセスを終了します

# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います

make ARCH=ARM

# クロスコンパイルによって生成された armhf アーキテクチャプログラムは build_ARM ディレクトリに

あり、

# 開発ボードにコピーして実行することができます
```

実験現象は、"CTRL+C"を押した時、`signal_handler()`シグナル処理関数に入り、対応する情報を出力します。処理後にデフォルトに復元されるため、次に"CTRL+C"を押した時、プロセスは直接終了します。

実験現象は以下のとおりです：

35.4 信号関連 API 関数

これまでの実験で`"Ctrl+C"`を押して信号を送信しましたが、コード内では kill()、raise()、alarm()などの信号送信関数を呼び出して信号を送ることができます。以下では、これらの関数について順に紹介します。

株式会社日昇テクノロジー

35.4.1 kill()

この関数を説明する前に、まず`kill`コマンドを使ってプロセスを終了させてみましょう。具体的な操作は以下の通りです。

- 最初に`ps –ux`コマンドを使用して、終了させるプロセスを確認します。なければ新しいターミナルを開いてください。ターミナルもプロセスの一つですから、このプロセスを終了させることができます。

以下のコマンドを入力します。なければ新しいターミナルをいくつか開いてください

```
ps –ux
```

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/sigaction$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.sigaction.o.d sources/sigaction.c -o build_x86/sigaction.o
gcc build_x86/sigaction.o -o build_x86/sigaction_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/sigaction$ ./build_x86/sigaction_demo
this is sigaction function test demo!

waiting for the SIGINT signal , please enter "ctrl + c"...

waiting for the SIGINT signal , please enter "ctrl + c"...

^C
this signal number is 2
I have get SIGINT!

The signal is automatically restored to the default handler!

waiting for the SIGINT signal , please enter "ctrl + c"...

waiting for the SIGINT signal , please enter "ctrl + c"...

^C
```

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

xxx 22133 0.3 0.0 14916 4820 pts/5 Ss 07:49 0:00 /bin/bash

xxx 22142 0.5 0.0 14916 4768 pts/6 Ss+ 07:49 0:00 /bin/bash

xxx 22151 0.0 0.0 29580 1500 pts/5 R+ 07:50 0:00 ps -ux

株式会社日昇テクノロジー

xxx 24331 0.0 0.0 15156 5244 pts/3 Ss+ 03:12 0:00 /bin/bash

パソコンによって出力内容が異なる可能性がありますが、最後の列のプロセス名（bash）はターミナルを意味しているため、開いている 3 つのターミナルのうち 1 つを閉じることができます。PID が 22142 のターミナル 6 のプロセスを終了させるために`kill 22142`と入力します。これにより、ターミナル 6 が閉じられたことが確認できます。kill コマンドの構文は以下の通りです：

```
kill [シグナルまたはオプション] PID(s)
```

ここで、[シグナルまたはオプション]は省略可能です。PID(s)は対象プロセスの ID で、一つでも複数でも可能です。現在のフォアグラウンドプロセスでないプロセスにシグナルを送信する場合に kill コマンドを使用します。このコマンドには、シグナルコードまたはシグナル名と、シグナルを受信する対象プロセスの PID（この PID は通常、ps コマンドで確認されます）が必要です。例えば、別のターミナルで実行されている PID666 のプロセスに"SIGHUP"シグナル（ハングアップ）を送信する場合、以下のコマンドを使用します：

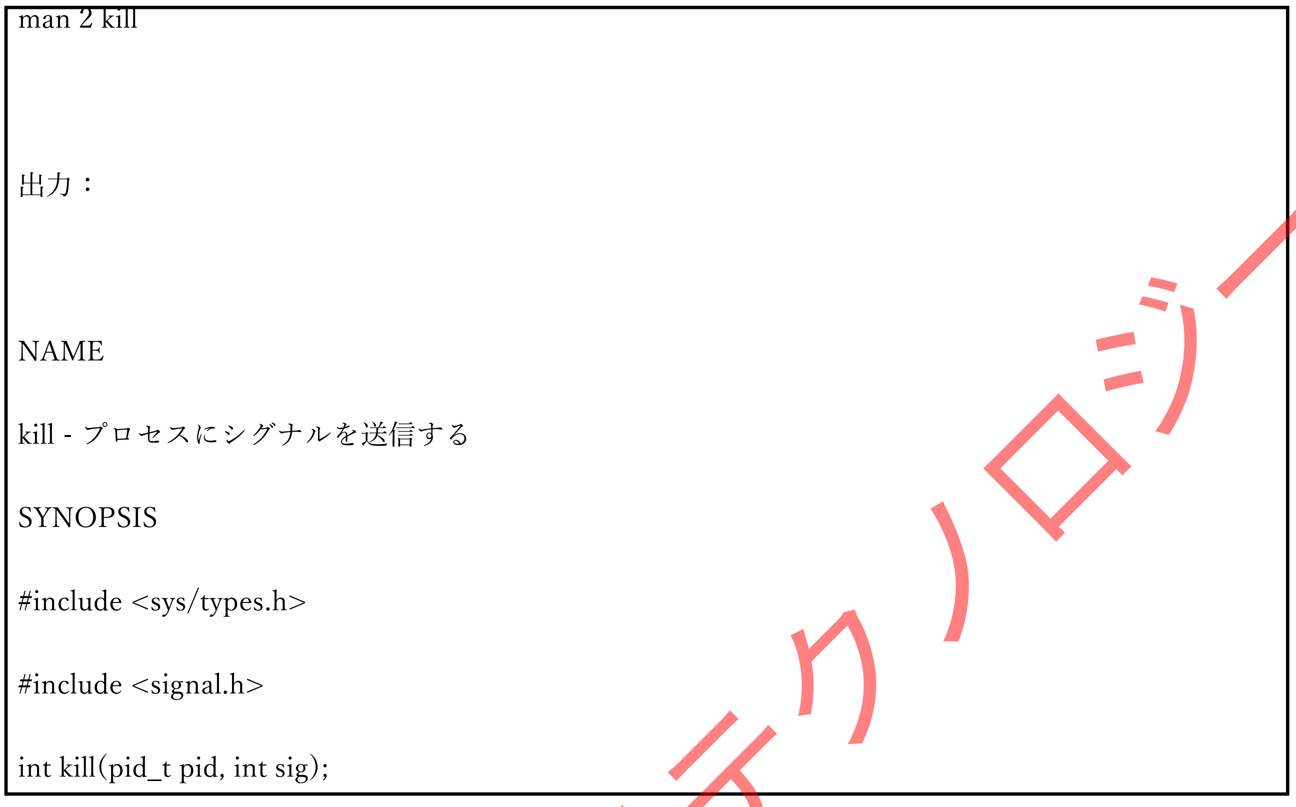
```
kill -SIGHUP 666
# または
kill -1 666
```

ここでの`-1`は SIGHUP シグナルのシグナル値 1 を指しています。

kill コマンドのデモンストレーションの後、Linux システム内のシグナル操作関連の関数について見ていきます。kill()関数は kill システムコマンドと同様に、プロセスまたはプロセスグループにシグナルを送信することができます。実際には、kill システムコマンドは kill()関数のユーザーインターフェースに過ぎません。これは、プロセスを中止するだけでなく（実際には SIGKILL シグナルを発信する）、他のシグナルもプロセスに送信できることを意味します。

株式会社日昇テクノロジー

同様に、システム内の kill()関数に関する説明を man コマンドで確認します：



```
man 2 kill

出力：

NAME

kill - プロセスにシグナルを送信する

SYNOPSIS

#include <sys/types.h>

#include <signal.h>

int kill(pid_t pid, int sig);
```

kill()関数は 2 つのパラメーター、pid と sig を持ち、int 型のエラーコードを返します。

- pid の値は以下の通りです：

- pid > 1：シグナル sig を pid で指定されたプロセス ID のプロセスに送信します。

- pid = 0：シグナルは現在のプロセスと同じプロセスグループにあるすべてのプロセスに送信されます。

- pid = -1：sig をシステム内のすべてのプロセスに送信しますが、プロセス 1（init）は除外されます。

- pid < -1：シグナル sig をプロセスグループ番号が-pid（pid の絶対値）のすべてのプロセスに送信します。

- sig：送信するシグナルの値。

- 関数の戻り値：

- 0：送信成功。

- -1：送信失敗。

株式会社日昇テクノロジー

プロセスは kill()関数を呼び出して自身を含む他のプロセスにシグナルを送信することができます。プログラムがそのシグナルを送信する権限がない場合、kill 関数の呼び出しは失敗します。失敗の一般的な原因は、ターゲットプロセスが別のユーザーに所有されているためです。したがって、シグナルを送信するためには、送信プロセスは適切な権限を持っている必要があります。これは通常、二つのプロセスが同じユーザー ID を持っている必要があることを意味します（つまり、自分のプロセスにのみシグナルを送信できますが、スーパーユーザーは任意のプロセスにシグナルを送信できます）。

kill()関数は失敗すると-1 を返し、errno 変数を設定します。失敗の理由には、指定されたシグナルが無効である（errno が INVAL に設定される）、送信プロセスの権限が不足している（errno が EPERM に設定される）、ターゲットプロセスが存在しない（errno が ESRCH に設定される）などがあります。

35.4.2 raise()

raise()関数についても見てみましょう。これはシグナルを送信する関数ですが、kill()関数と異なり、raise()関数はプロセスが自身に対してのみシグナルを送信します。つまり、kill(getpid(), sig)は raise(sig)と等価です。以下に man コマンドで raise()関数に関する情報を確認します：

```
int raise(int sig);
```

raise()関数には sig という 1 つのパラメーターがあり、送信成功時には 0 を返し、失敗時には-1 を返します。送信失敗の主な原因はシグナルが無効であることですが、自身に対してのみシグナルを送信するため、権限の問題やターゲットプロセスが存在しないという状況は発生しません。

35.4.2.1 実験分析

以下は、raise と kill 関数の例を含む小さな実験です。実験コードは野火が提供する資料の system_programing/kill ディレクトリにあります：

株式会社日昇テクノロジー

リスト 4: raise と kill 関数の例（関連コードリポジトリ

/system_programing/kill/sources/kill.c ファイル）

```
1 #include <unistd.h>

2 #include <stdio.h>

3 #include <stdlib.h>

4 #include <signal.h>

5 #include <sys/types.h>

6 #include <sys/wait.h>

7

8 int main(void)

9 {

10     pid_t pid;

11

12     int ret;

13

14     /* 子プロセスを作成 */

15     if ((pid = fork()) < 0) { // (1)

16         printf("Fork エラー¥n");

17         exit(1);

18     }

19

20     if (pid == 0) { // (2)
```

株式会社日昇テクノロジー

```
21      /* 子プロセスで raise()関数を使って SIGSTOP シグナルを発行し、子プロセスを一時停止 */
22      printf("子プロセス(pid : %d) は任意のシグナルを待っています¥n¥n", getpid());
23
24      /** 子プロセスはここで停止 */
25      raise(SIGSTOP); // (3)
26
27      exit(0);
28   }
29
30   else { // (4)
31      /** 少し待って、子プロセスが先に実行されるようにする */
32      sleep(1);
33
34      /* 親プロセスで子プロセスからのシグナルを受け取り（ブロックしない）し、kill()関数を呼び出し
て適切な操作を行う */
35      if ((waitpid(pid, NULL, WNOHANG)) == 0) { // (5)
36         /** 子プロセスがまだ終了していない場合、0 を返し、SIGKILL シグナルを送って子プロセスを
強制終了させる */
37         if ((ret = kill(pid, SIGKILL)) == 0) {
38            printf("親プロセスが %d を終了させました¥n¥n", pid); // (6)
39         }
40      }
```

株式会社日昇テクノロジー

```
41
42      /** 子プロセスが終了するまでブロックする */
43      waitpid(pid, NULL, 0); // (7)
44
45      exit(0);
46   }
47 }
```

- (1)：fork を使用して子プロセスを開始します。戻り値が 0 未満（-1 の場合）であれば、開始に失敗したことを意味します。

- (2)：戻り値が 0 の場合、現在実行中のプロセスは子プロセスであり、関連情報を印刷します。

- (3)：子プロセスでは、raise()関数を使用して SIGSTOP シグナルを発信し、子プロセスを一時停止させます。

- (4)：親プロセスが実行中の場合は、子プロセスが先に実行されるように少し待ちます。

- (5)：親プロセスでは、waitpid()関数を使用して子プロセスからのシグナルを収集します（ブロックしない）。

- (6)：子プロセスがまだ終了していない場合、kill()関数を使用して子プロセスに終了シグナルを送信します。このシグナルを受け取った子プロセスは終了します。

- (7)：waitpid()関数を使用して子プロセスのリソースを回収します。子プロセスが終了していない場合、親プロセスは子プロセスが終了するまでブロックして待機します。

株式会社日昇テクノロジー

この例のコードは system_programing/kill ディレクトリにあり、以下の手順で実験を実行します。

```
# system_programing/kill コードディレクトリで以下の操作を行います

# X86 バージョンのプログラムをコンパイル

make

# X86 バージョンのプログラムを実行

./build_x86/kill_demo



# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います

make ARCH=ARM

# クロスコンパイルされた armhf アーキテクチャのプログラムは build_ARM ディレクトリにあります。

# 開発ボードにコピーして実行します
```

実行後の実験現象は上記の通りです。

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_prog
raming/kill$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.kill.o.d sources/kill.c -o build_x86/k
ill.o
gcc build_x86/kill.o -o build_x86/kill_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_prog
raming/kill$ ./build_x86/kill_demo
Child(pid : 18174) is waiting for any signal

Parent kill 18174
```

35.4.3 alarm()

alarm()関数、またはアラーム関数は、プロセス内にタイマーを設定し、指定された時間（seconds）が経過するとプロセスに SIGALRM 信号を送信します。関数のプロトタイプは以下の通りです：

```
unsigned int alarm(unsigned int seconds);
```

seconds 秒内に再度 alarm()関数が呼び出され、新しいアラームが設定されると、前の設定は新しい設定に置き換えられます。つまり、以前に設定された秒数は新しいアラーム時間に置き換えられます。戻り値

株式会社日昇テクノロジー

は前のアラームの残り秒数で、以前にアラームが設定されていない場合は 0 を返します。特に、新しい seconds が 0 の場合は、以前に設定されたアラームがキャンセルされ、残り時間が返されます。

35.4.3.1 実験分析

alarm()関数の機能特性と戻り値の特性を理解した後、テストを行うことができます。テストの方向性は 2 つあります：一つは、単独で存在するアラーム関数 alarm()のプログラムをテストすること、もう一つは、複数の alarm()アラーム関数を含むプログラムをテストすることです。そのため、以下の 2 つのプログラムを整理し、比較学習することでより理解を深めます。

リスト 5: alarm()関数の例

```
1 int main()
2 {
3     printf("¥nthis is an alarm test function¥n¥n");
4     alarm(5);
5     sleep(20);
6     printf("end!¥n");
7     return 0;
8 }
```

このテストは SIGALRM 信号のデフォルト処理を検証するためのものです。

- 実際にこのプログラムは、alarm(5)で時計を定義しています。これは、5 秒後に SIGALRM 信号を main()が存在する現在のプロセスに送信することを意味します。

- 次に、sleep(20)を呼び出してプロセスを 20 秒間スリープさせます。

- main()プログラムが 5 秒間サスペンドされた後、alarm によって SIGALRM 信号が生成されますが、キャッチ処理を行わないため、システムはその信号のデフォルト処理関数を呼び出し、exit(0)関数を

株式会社日昇テクノロジー

直接実行してプロセスを終了させ、終了時に自動的に"Alarm clock"（アラームクロック）を出力します。

- デフォルト処理関数の実行後、プロセスが終了するため、コード自体の最後の文 printf("end!¥n")は実行されません。

以下のコード例は system_programing/alarm ディレクトリ内の alarm.c ファイルに含まれており、コンパイルする前にマクロを使ってこのセクションのサンプルコードに切り替える必要があります。

```
# コンパイルする前に、ソースファイルを開いてマクロを使ってテストしたいコードに切り替えてください！！！
# 以下の操作は system_programing/alarm コードディレクトリで行ってください
# X86 バージョンのプログラムをコンパイル
make
# X86 バージョンのプログラムを実行
./build_x86/alarm_demo
# 5 秒後に、ターミナルは "Alarm clock" （アラームクロック）と出力し、プロセスは終了します
# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います
make ARCH=ARM
# クロスコンパイルで生成された armhf アーキテクチャのプログラムは build_ARM ディレクトリ内にあり、
# 開発ボードにコピーして実行することができます
```

実験の現象は以下の通りです。5 秒後に、ターミナルは Alarm clock と出力し、プロセスは終了します：

次に、alarm() 関数のオーバーライド設定実験を行います。

株式会社日昇テクノロジー

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_
programing/alarm$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.alarm.o.d sources/alarm.c -o build
_x86/alarm.o
gcc build_x86/alarm.o -o build_x86/alarm_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_
programing/alarm$ ./build_x86/alarm_demo

this is an alarm test function

 Alarm clock
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_
programing/alarm$
```

次に、alarm()関数の上書き設定に関する実験を行います。

リスト 6: alarm()の上書き例

```
1 int main()

2 {

3 unsigned int seconds;

4

5 printf("¥nthis is an alarm test function¥n¥n");

6

7 seconds = alarm(20);

8

9 printf("last alarm seconds remaining is %d! ¥n¥n", seconds);

10

11 printf("process sleep 5 seconds¥n¥n");

12 sleep(5);

13

14 printf("sleep woke up, reset alarm!¥n¥n");
```

株式会社日昇テクノロジー

```
15

16 seconds = alarm(5);

17

18 printf("last alarm seconds remaining is %d! ¥n¥n", seconds);

19

20 sleep(20);

21

22 printf("end!¥n");

23

24 return 0;

25 }
```

この alarm テストコードは、複数回 alarm を設定すると、前回の設定値が上書きされることを検証するためのものです。コードのロジックは非常にシンプルで、最初に alarm(20)関数を呼び出して 20 秒後に SIGALRM 信号を生成するように設定し、プロセスが 5 秒間スリープから覚めた後、再び alarm(5)関数を設定して 5 秒後に SIGALRM 信号でプロセスを終了させます。この時、前の alarm 設定は上書きされ、前回設定の残り時間（15 秒）が返されます。上書き設定後、プロセスは引き続きスリープし、5 秒後に SIGALRM 信号が到着するのを待ちます。

株式会社日昇テクノロジー

```
# コンパイルする前に、ソースファイルを開いてマクロを使ってテストしたいコードに切り替えてください！！！

# 以下の操作は `system_programing/alarm` コードディレクトリで行ってください。

# X86 バージョンのプログラムをコンパイル

make

# X86 バージョンのプログラムを実行

./build_x86/alarm_demo

# 5 秒待つと、ターミナルが「Alarm clock」（アラームクロック）と出力し、プロセスが終了します。

# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います。

make ARCH=ARM

# クロスコンパイルで生成された armhf アーキテクチャのプログラムは、build_ARM ディレクトリにあります。

# 開発ボードにコピーして実行することができます。
```

具体的な実験現象は以下の画像の通りです。

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/alarm$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.alarm.o.d sources/alarm.c -o build_x86/alarm.o
gcc build_x86/alarm.o -o build_x86/alarm_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/alarm$ ./build_x86/alarm_demo

this is an alarm test function

last alarm seconds remaining is 0!

process sleep 5 seconds

sleep woke up, reset alarm!

last alarm seconds remaining is 15!

Alarm clock
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/system_programing/alarm$
```

株式会社日昇テクノロジー

もし、alarm 信号を自分で処理したい場合は、前のセクションの signal()または sigaction()関数を使用して SIGALRM 信号をキャッチすればよいです。

第 36 章 パイプについて

36.1 パイプの基本概念

正式な学習に入る前に、なぜパイプと呼ばれるのか、また日常生活でパイプに関連するものが何かを考えてみましょう。例えば水道管、水は一方の端から他方の端へと水道管を通って流れます。そのように、プロセス間通信もこの「流れ」の関係を模倣できるのではないでしょうか。明らかに、データは一つのプロセスから別のプロセスへと流れることができます。つまり、一つのプロセスがデータを生成し、パイプを通じて別のプロセスに送信し、もう一方のプロセスがパイプの他方の端からデータを読み取ることで、プロセス間通信が実現されます。

前章で信号について学習しましたが、信号はプロセス内で生成され、別のプロセスに送信されます。これも信号タイプの通信の一種ですが、信号値のみを伝達しデータが伝達されないため、多くの場合でニーズを満たすことができません。したがって、データを伝送するパイプの機能は、ある場合には大きな利点を持ちます。

まず、ターミナルで以下のコマンドを使用して実験を行ってみましょう：

```
ps -aux | grep root

# 出力

root 1 0.0 0.0 225376 6376 ? Ss 10 月  18 0:31 /sbin/init

root 2 0.0 0.0 0 0 ? S 10 月  18 0:00 [kthreadd]

root 4 0.0 0.0 0 0 ? I< 10 月  18 0:00 [kworker/0:0H]

root 6 0.0 0.0 0 0 ? I< 10 月  18 0:00 [mm_percpu_wq]

root 7 0.0 0.0 0 0 ? S 10 月  18 0:02 [ksoftirqd/0]
```

株式会社日昇テクノロジー

```
root 8 0.0 0.0 0 0 ? I 10 月  18 5:35 [rcu_sched]

root 9 0.0 0.0 0 0 ? I 10 月  18 0:00 [rcu_bh]

root 10 0.0 0.0 0 0 ? S 10 月  18 0:00 [migration/0]

root 11 0.0 0.0 0 0 ? S 10 月  18 0:01 [watchdog/0]

root 12 0.0 0.0 0 0 ? S 10 月  18 0:00 [cpuhp/0]

root 13 0.0 0.0 0 0 ? S 10 月  18 0:00 [cpuhp/1]
```

まず ps コマンドは非常によく知られています。これは現在のプロセスをリストアップします。grep コマンドも使用したことがありますが、これは正規リスト現を使用してテキストを検索する強力なテキスト検索ツールです。では、ps と grep コマンドの間の「|」記号は何でしょうか。これは実際にはパイプで、ps コマンドの出力データを grep に流し込むものです。ここでは実際に 2 つのプロセスが開かれています。ps コマンドは本来ターミナルに情報を出力するはずですが、パイプを通じてその出力情報を grep コマンドの入力情報として送信し、検索後に適切な情報を表示します。これにより、ターミナルに表示される情報が形成されます。

次のコマンドを使用して、実際に 2 つのプロセスが開かれているかどうかを確認できます。出力の最後に必ず 2 つのプロセスに関連する情報（ps、grep）が表示されます：

```
ps -ux | grep $USER

出力例（出力の最後に表示されます）：

xxx 29663 0.0 0.0 29580 1460 pts/0 R+ 07:12 0:00 ps -ux

xxx 29664 0.0 0.0 14888 1016 pts/0 S+ 07:12 0:00 grep --color=auto xxx
```

パイプとは何か？データが一つのプロセスから別のプロセスへと流れるとき、その間の接続がパイプ（pipe）です。通常、一つのプロセスの出力を別のプロセスの入力にパイプで接続します。

シェルコマンドにおいて、コマンドの接続はパイプ文字を使って行われます。ps -aux | grep root のようなコマンドでは、「|」文字を使って接続するだけです。

株式会社日昇テクノロジー

それでは、「ps -aux | grep root」コマンドについて詳しく分析してみましょう。実際には以下のプロセスが実行されます：

- シェルが 2 つのコマンドの標準入出力を配置します。
- ps の標準入力はターミナルのマウス、キーボードなどから来ます。
- ps の標準出力は grep に渡され、grep の標準入力となります。
- grep の標準出力はターミナルに接続され、最終的に grep の出力結果を見ることができます。

シェルが行う作業は実際には標準入出力ストリームを再接続し、`ps`コマンドと`grep`の間にデータパイプを構築することです。実際には、パイプも本質的にはファイルであり、上記のプロセスは`ps`プロセスが出力内容をパイプに書き込み、grep プロセスがパイプからデータを読み取るということを抽象化したものです。Linux の「全てはファイルである」という設計思想に従っています。それは VFS（仮想ファイルシステム）を介してアプリケーションに操作インターフェイスを提供し、パイプの機能を実現します。

ただし、パイプがファイルの形をしているとはいえ、パイプ自体はディスクやその他の外部ストレージの空間を占有しません。実際にはメモリ空間を占有しています。したがって、Linux 上のパイプはファイルの操作方法を持つメモリバッファに過ぎません。

36.2 パイプの分類

Linux システム上のパイプには 2 種類あります：

- 匿名パイプ
- 名前付きパイプ

これらのパイプは無名または有名パイプとも呼ばれますが、統一性のために、以下では匿名パイプと名前付きパイプと呼びます。匿名パイプの最も一般的な形は、シェル操作で最もよく使用される「|」です。その特徴は、父子プロセス間でのみ使用できることです。親プロセスは子プロセスを生成する前にパイプファイルを開き、その後 fork で子プロセスを生成します。この方法で子プロセスは親プロセスのプロセスアドレス空間をコピーすることにより、同じパイプファイルのディスクリプタを得て、同じパイプを通じて

株式会社日昇テクノロジー

通信する目的を達成します。この場合、父子プロセス以外にはこのパイプファイルのディスクリプタがわからないため、このパイプを介して他のプロセスに情報を伝達することはできません。これにより、データ転送の安全性が保証されますが、パイプの汎用性も低下します。そのため、システムは名前付きパイプも提供しています。これは本質的にファイルであり、ファイルシステム内に位置しています。名前付きパイプを使用すると、関連のない複数のプロセスが通信を行うことができます。

36.2.1 匿名パイプ（PIPE）

匿名パイプ（PIPE）は特別なファイルの一種ですが、名前がないため、通常のプロセスは open()関数を使用してそのディスクリプタを取得することはできません。これは、プロセス内で作成され、そのファイルディスクリプタを子プロセスに継承によって渡すことでのみ可能です。これが匿名パイプが親子関係のプロセス間通信のみに使用される理由です。また、匿名パイプは通常のファイルと異なり、2 つのファイルディスクリプタを持ちます。一つは読み取り専用、もう一つは書き込み専用です。これを「半二重」通信方式と呼びます。さらに、書き込み操作には保護がありません。つまり、複数のプロセスやスレッドが同時に匿名パイプに書き込む場合、これらのデータは互いに上書きされる可能性があります。したがって、匿名パイプは一対一の親子プロセス通信にのみ使用すべきです。最後に、匿名パイプは lseek()を使用して所定の位置に移動することはできません。データは通常のファイルのようにブロック形式でハードディスクやフラッシュメモリなどのブロックデバイスに保存されるわけではないからです。

匿名パイプの特徴をまとめると以下の通りです：

- 名前がないため、open()関数で開くことはできませんが、close()関数で閉じることは可能です。
- 一方向通信（半二重通信）のみを提供します。つまり、二つのプロセスがこのファイルにアクセスできますが、プロセス 1 がファイルにデータを書き込む場合、プロセス 2 はその内容を読み取ることしかできません。
- 親子関係のあるプロセス間通信にのみ使用できます。

株式会社日昇テクノロジー

- パイプはバイトストリームに基づいて通信します。

- ファイルシステムに依存し、プロセスが終了すると生存期間も終了します。

- 書き込み操作は原子性を持たず、一対一の単純な通信シナリオにのみ適しています。

- パイプは特殊なファイルと見なすことができ、読み書きには通常の read()や write()などの関数を使用できます。しかし、通常のファイルとは異なり、他のどのファイルシステムにも属さず、内核のメモリ空間内にのみ存在するため、lseek()を使用して位置を特定することはできません。

36.2.2 命名パイプ（FIFO）

命名パイプ（FIFO）は匿名パイプ（PIPE）とは異なり、関連のない複数のプロセス間でデータを交換（通信）できます。匿名パイプの通信は通常、共通の先祖プロセスによって開始され、「血縁関係」のあるプロセス間でのみデータ交換が可能です。これは、関連のないプロセス間でデータを交換する際の不便さを解消するため、命名パイプが考案されました。

命名パイプは無名パイプと異なる点は、ファイルシステム内にパス名を持ち、ファイルの形式で存在することです。これにより、命名パイプを作成したプロセスと「血縁関係」のないプロセスでも、命名パイプファイルのパスにアクセスできれば、互いに命名パイプを介して通信することが可能になります。ファイル形式でアクセスできるため、open()、read()、write()、close()などのファイル操作関数を使用できます。命名パイプファイルはファイルシステムに保存されていますが、データはメモリ内に存在する点が異なります。

命名パイプの特徴をまとめると以下の通りです：

- 名前があり、通常のファイルシステム内に保存されます。

- 適切な権限を持つ任意のプロセスが open()を使用して命名パイプのファイルディスクリプタを取得できます。

- 通常のファイルと同様に、統一された read()/write()で読み書きします。

株式会社日昇テクノロジー

- 通常のファイルとは異なり、lseek()を使用して位置を特定することはできません。データはメモリ内に保存されるためです。

- 書き込みの原子性があり、複数のライターが同時に書き込んでもデータが互いに上書きされることはありません。

- 先入れ先出し（FIFO）の原則に従い、FIFO に最初に書き込まれたデータが最初に読み出されます。

36.3 pipe()関数

pipe()関数は匿名パイプを作成するために使用されます。これは、プロセス間通信に使用できる単方向のデータチャネルを作成します。man コマンドで pipe 関数のプロトタイプを確認することができます：

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

この関数プロトタイプは非常にシンプルで、入力パラメータはありません。注意点として、配列 pipefd はパイプの両端のファイルディスクリプタを返すために使用されます。これは、2 つのファイルディスクリプタを含む配列のポインタです。pipefd[0]はパイプの読み取り端を指し、pipefd[1]はパイプの書き込み端を指します。パイプの書き込み端にデータを書き込むと、内核によってバッファリング（メモリに書き込み）され、パイプの読み取り端からデータが読み取られるまで保持されます。また、データは先入れ先出しの原則に従います。pipe()関数は int 型の変数も返し、0 の場合は匿名パイプの作成に成功したことを意味し、-1 の場合は作成に失敗し、errno が設定されます。

匿名パイプの作成に成功した後、その匿名パイプを作成したプロセス（親プロセス）は、パイプの読み取り端と書き込み端の両方を管理します。しかし、親子プロセス間でデータ交換を行いたい場合は、次の操作が必要です：

- 親プロセスは pipe()関数を呼び出して匿名パイプを作成し、2 つのファイルディスクリプタ pipefd[0]、pipefd[1]を取得します。これらはそれぞれパイプの読み取り端と書き込み端を指します。

- 親プロセスは fork()関数を呼び出して子プロセスを起動（作成）します。すると、子プロセスは親プ

株式会社日昇テクノロジー

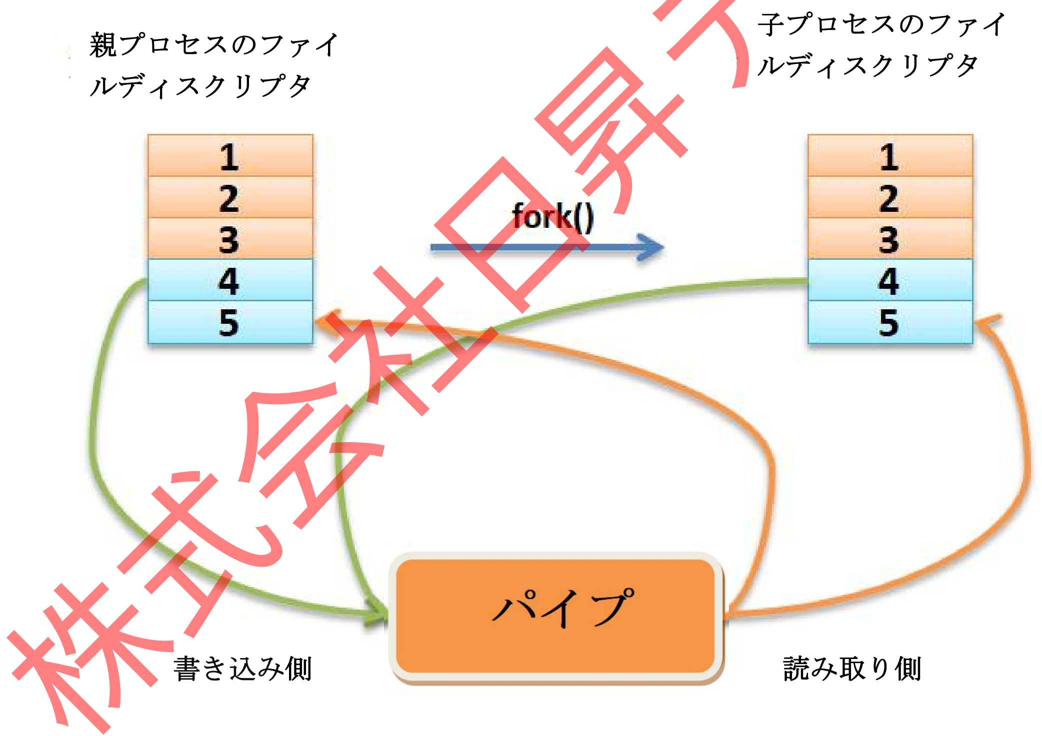
ロセスからこれら 2 つのファイルディスクリプタ pipefd[0]、pipefd[1]を継承し、同じ匿名パイプの読み取り端と書き込み端を指します。

- 匿名パイプは環状キューを使用して実装されており、データは書き込み端からパイプに流れ込み、読み取り端から流れ出ます。これによりプロセス間通信が実現されますが、この時点でパイプには 2 つの読み取り端と 2 つの書き込み端が存在します。したがって、次の操作が必要になります。

- 親プロセスから子プロセスにデータを渡したい場合は、親プロセスが読み取り端を閉じ、子プロセスが書き込み端を閉じる必要があります。

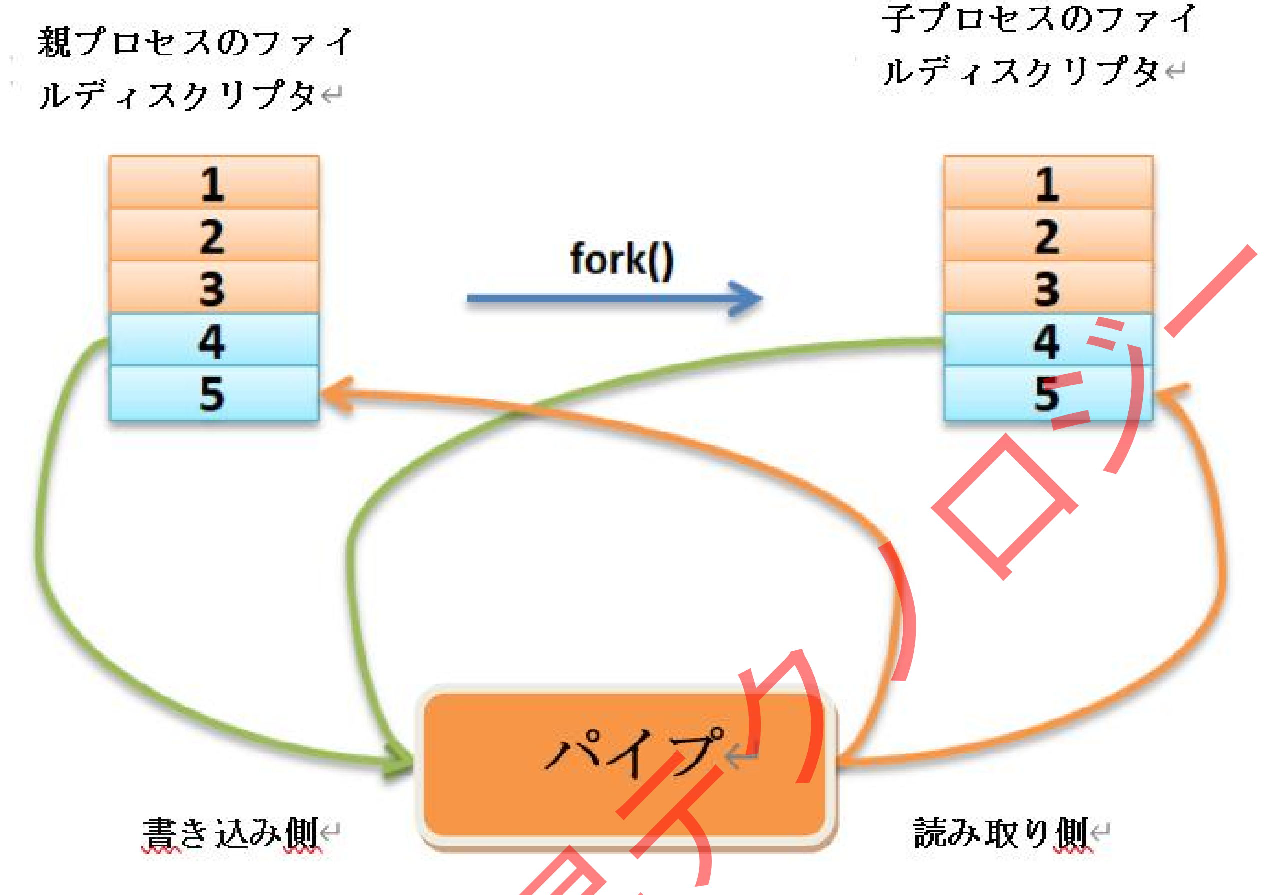
- 子プロセスから親プロセスにデータを渡したい場合は、親プロセスが書き込み端を閉じ、子プロセスが読み取り端を閉じる必要があります。

- パイプが不要になった場合は、プロセス内で開いている端を閉じるだけです。



上記の図：fork 後、子プロセスが親プロセスのファイルディスクリプタを継承します。

株式会社日昇テクノロジー



上記の図：データが親プロセスから子プロセスへ流れます。

上記の図：データが子プロセスから親プロセスへ流れます。

36.3.1 実験分析

pipe()関数を使用してテスト実験を行うことができます。野火の資料では、対応する実験コードが提供されており、system_programing/pipe ディレクトリに pipe.c ファイルが存在します。そのファイルの内容は以下の通りです。

株式会社日昇テクノロジー

親プロセスのファイルディスクリプタ

子プロセスのファイルディスクリプタ

1
2
3
4
5

fork()

1
2
3
4
5

パイプ

書き込み側

読み取り側

リスト 1: パイプ pipe の例

(base_code/system_programing/pipe/sources/pipe.c ファイル)

```
1 #include <unistd.h>

2 #include <sys/types.h>

3 #include <sys/wait.h>

4 #include <errno.h>

5 #include <stdio.h>

6 #include <stdlib.h>

7 #include <string.h>

8
```

株式会社日昇テクノロジー

```
9  #define MAX_DATA_LEN 256
10 #define DELAY_TIME 1
11
12 int main()
13 {
14   pid_t pid;
15   int pipe_fd[2]; // (1) パイプのファイル記述子
16   char buf[MAX_DATA_LEN];
17   const char data[] = "Pipe Test Program";
18   int real_read, real_write;
19
20   memset((void*)buf, 0, sizeof(buf));
21
22   /* パイプの作成 */
23   if (pipe(pipe_fd) < 0) // (2) パイプを作成
24   {
25     printf("パイプの作成に失敗しました¥n");
26     exit(1);
27   }
28
29   /* 子プロセスの作成 */
30   if ((pid = fork()) == 0) // (3) 子プロセスを作成
```

株式会社日昇テクノロジー

```
31 {
32 /* 子プロセスは書き込み用ファイル記述子を閉じ、3 秒間待機して、親プロセスが読み取り用記述子を閉じるのを待つ */
33 close(pipe_fd[1]);
34 sleep(DELAY_TIME * 3);
35
36 /* 子プロセスがパイプからデータを読み込む */ // (4)
37 if ((real_read = read(pipe_fd[0], buf, MAX_DATA_LEN)) > 0)
38 {
39 printf("%d バイトがパイプから読み込まれました: '%s'¥n", real_read, buf);
40 }
41
42 /* 子プロセスが読み取り用ファイル記述子を閉じる */
43 close(pipe_fd[0]); // (5)
44
45 exit(0);
46 }
47
48 else if (pid > 0)
49 {
50 /* 親プロセスは読み取り用ファイル記述子を閉じ、1 秒間待機して、子プロセスが書き込み用記述子を閉じるのを待つ */
```

株式会社日昇テクノロジー

```
51    close(pipe_fd[0]); // (6)

52

53    sleep(DELAY_TIME);

54

55    if((real_write = write(pipe_fd[1], data, strlen(data))) != -1) // (7)

56    {

57      printf("親プロセスが %d バイト書き込みました: '%s'¥n", real_write, data);

58    }

59

60    /* 親プロセスが書き込み用ファイル記述子を閉じる */

61    close(pipe_fd[1]); // (8)

62

63    /* 子プロセスの終了情報を回収する */

64    waitpid(pid, NULL, 0); // (9)

65

66    exit(0);

67 }

68 }
```

株式会社日昇テクノロジー

- (1)：配列 pipe_fd を定義し、匿名パイプを作成した後、配列を通じてパイプのファイルディスクリプタを返します。

- (2)：pipe()を呼び出して匿名パイプを作成し、成功するとファイルディスクリプタ pipe_fd[0]、pipe_fd[1]が得られます。そうでなければ-1 を返します。

- (3)：fork()を呼び出して子プロセスを作成します。戻り値が 0 の場合、実行中のプロセスは子プロ

セスであり、子プロセス内で close()関数を呼び出して書き込みディスクリプタを閉じ、親プロセスが対応する読み取りディスクリプタを閉じるのを 3 秒間待ちます。

- (4)：子プロセスが read()関数を呼び出してパイプの内容を読み取ります。パイプにデータがない場合、子プロセスはブロックされます。データを読み取ると、そのデータを出力します。特に、書き込みディスクリプタが閉じられたパイプを read()関数で読み取ると、read()は 0 を返します（この例では親プロセスの書き込みディスクリプタは閉じていません）。

- (5)：close()関数を呼び出して子プロセスの読み取りディスクリプタを閉じます。

- (6)：fork()関数の戻り値が 0 より大きい場合、実行中のプロセスは親プロセスであり、親プロセスはまずパイプの読み取りディスクリプタを閉じ、1 秒待ちます。この時、子プロセスが親プロセスより先に実行される可能性があるため、少し待ちます。

- (7)：親プロセスが write()関数を呼び出してデータをパイプに書き込みます。

- (8)：親プロセスの書き込みディスクリプタを閉じます。

- (9)：waitpid()関数を呼び出して子プロセスの終了情報を収集し、プロセスを終了します。

パイプの例のコンパイルおよびテストプロセスは以下の通りです：

```
# 以下の操作は system_programing/pipe コードディレクトリで行います
# X86 バージョンのプログラムをコンパイル
make
# X86 バージョンのプログラムを実行
./build_x86/pipe_demo

# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います
make ARCH=ARM
```

株式会社日昇テクノロジー

```
# クロスコンパイルで生成された armhf アーキテクチャのプログラムは build_ARM ディレクトリにあります。
# 開発ボードにコピーして実行します。
```

実行結果は以下の図のように、子プロセスが親プロセスからパイプに書き込まれた内容を読み取ります：

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial
/base_code/system_programing/pipe$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.pipe.o.d sources
/pipe.c -o build_x86/pipe.o
gcc build_x86/pipe.o -o build_x86/pipe_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial
/base_code/system_programing/pipe$ ./build_x86/pipe_demo
Parent write 17 bytes : 'Pipe Test Program'
17 bytes read from the pipe is 'Pipe Test Program'
```

36.4 fifo()関数

これまで、共通の祖先プロセスによって開始されたプログラム間でのみデータを渡すことができました。しかし、関連のないプロセス間でデータを交換したい場合は、FIFO ファイル、または命名パイプを使用してこれを実現することができます。命名パイプは、ファイルシステム内にファイル名の形で存在する特殊なタイプのファイルですが、そのデータはメモリ内に保存されます。端末（コマンドライン）上で命名パイプを作成することも、プログラム内で作成することもできます。

例えば、mkfifo コマンドを使用して命名パイプを作成する場合、man コマンドを使用してその説明を確認することができます。

```
# 次のコマンドを実行します：
man mkfifo
# 出力
名前
        mkfifo - FIFO（名前付きパイプ）を作成する
概要
        mkfifo [オプション]... 名前...
```

株式会社日昇テクノロジー

説明

指定された名前で名前付きパイプ（FIFO）を作成します。

必須の引数は、長短のオプションに同時に適用されます。

-m, --mode=モード

ファイルのパーミッションビットを指定されたモードに設定します（chmod と同様）、a=rw のようにはなりません

-Z　SELinux のセキュリティコンテキストをデフォルトタイプに設定

--context[=CTX]

-Z と同様、または指定された場合は SELinux または SMACK セキュリティコンテキストを指定されたタイプに設定

--help このヘルプ情報を表示して終了

名前付きパイプを作成すると、名前付きパイプファイル test が作成されます。この test ファイルのタイプを file コマンドで確認できます：

```
# 次のコマンドを実行します
mkfifo test
file test
# コマンドの出力から、それが名前付きパイプタイプのファイルであることがわかります
test: fifo (named pipe)
```

この mkfifo コマンドは、実際には Linux システムの同名 API mkfifo に対応しており、ソースコード内で mkfifo 関数を呼び出すことにより、名前付きパイプを作成できます。これは、ファイルを作成するのと似ていますが、このファイルのタイプは名前付きパイプです。

株式会社日昇テクノロジー

mkfifo()の関数プロトタイプは以下の通りです：

```
int mkfifo(const char * pathname, mode_t mode);
```

mkfifo()は、pathname 引数で指定された特殊な FIFO ファイルを作成し、mode 引数でそのファイルのモードとパーミッションを設定します。

mkfifo()によって作成された FIFO ファイルは、他のプロセスが読み書きすることができ、open、read、write、close などの一般的なファイル操作方法で操作できます。

モードおよびパーミッションパラメータの説明：

- O_RDONLY：パイプの読み取り。
- O_WRONLY：パイプの書き込み。
- O_RDWR：パイプの読み書き。
- O_NONBLOCK：非ブロッキング。
- O_CREAT：ファイルが存在しない場合、新しいファイルを作成し、3 番目の引数でそのパーミッションを設定します。
- O_EXCL：O_CREAT を使用してファイルが既に存在する場合、エラーメッセージを返すことができます。この引数はファイルの存在をテストするために使用できます。

関数の戻り値の説明は以下の通りです：

- 0：成功
- EACCESS：filename 引数で指定されたディレクトリパスに実行権限がありません。
- EEXIST：filename 引数で指定されたファイルが既に存在します。
- ENAMETOOLONG：filename 引数のパス名が長すぎます。
- ENOENT：filename 引数に含まれるディレクトリが存在しません。
- ENOSPC：ファイルシステムの残りの空き容量が不足しています。
- ENOTDIR：filename 引数のパスにあるディレクトリが実際にはディレクトリではありません。

株式会社日昇テクノロジー

- EROFS：filename 引数で指定されたファイルが読み取り専用のファイルシステム内にあります。

FIFO を使用する過程で、プロセスがパイプを読み取る操作を行う場合：

- そのパイプがブロック型で、現在 FIFO 内にデータがない場合、読み取りプロセスはデータが書き込まれるまで常にブロックされます。

- そのパイプが非ブロック型の場合、FIFO 内にデータがあるかどうかにかかわらず、読み取りプロセスはすぐに読み取り操作を実行します。つまり、FIFO 内にデータがない場合、読み取り関数はすぐに 0 を返します。

FIFO を使用する過程で、プロセスがパイプに書き込む操作を行う場合：

- そのパイプがブロック型の場合、データが書き込み可能になるまで書き込み操作は常にブロックされます。

- そのパイプが非ブロック型であり、全てのデータを書き込むことができない場合、書き込み操作は部分的に書き込むか、または呼び出しが失敗します。

36.4.1 実験分析

以下は、具体的な例を見てみましょう：

リスト 2: fifo() 例（base_code/system_programing/fifo/sources/fifo.c ファイル）

```
1 #include <sys/wait.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <unistd.h>
5 #include <errno.h>
6 #include <fcntl.h>
7 #include <stdio.h>
```

株式会社日昇テクノロジー

```
8  #include <stdlib.h>

9  #include <limits.h>

10 #include <string.h>

11

12 #define MYFIFO "myfifo" /* 名前付きパイプのファイル名 */

13

14 #define MAX_BUFFER_SIZE PIPE_BUF /* 4096 は limits.h で定義されています */

15

16

17 void fifo_read(void)

18 {

19   char buff[MAX_BUFFER_SIZE];

20   int fd;

21   int nread;

22

23   printf("**************** FIFO の読み込み ***********************¥n");

24   /* 名前付きパイプが既に存在するか確認し、存在しなければ適切な権限で作成する */

25   if (access(MYFIFO, F_OK) == -1) //(4)

26   {

27     if ((mkfifo(MYFIFO, 0666) < 0) && (errno != EEXIST)) //(5)

28     {

29       printf("名前付きパイプファイルの作成に失敗しました¥n");
```

株式会社日昇テクノロジー

```
30      exit(1);
31    }
32  }
33
34  /* 読み取り専用ブロックモードで名前付きパイプを開く */
35  fd = open(MYFIFO, O_RDONLY); //(6)
36  if (fd == -1)
37  {
38    printf("名前付きパイプファイルのオープンに失敗しました¥n");
39    exit(1);
40  }
41
42  memset(buff, 0, sizeof(buff));
43
44  if ((nread = read(fd, buff, MAX_BUFFER_SIZE)) > 0) // (7)
45  {
46    printf("FIFO から '%s' を読み込みました¥n", buff);
47  }
48
49  printf("***************** FIFO のクローズ ***********************¥n");
50
51  close(fd); //(8)
```

株式会社日昇テクノロジー

```
52

53   exit(0);

54 }

55

56

57 void fifo_write(void)

58 {

59   int fd;

60   char buff[] = "これは FIFO テストデモです";

61   int nwrite;

62

63   sleep(2); //子プロセスが先に実行されるのを待つ //(9)

64

65   /* 書き込み専用ブロックモードで FIFO を開く */

66   fd = open(MYFIFO, O_WRONLY | O_CREAT, 0644); //(10)

67   if (fd == -1)

68   {

69     printf("名前付きパイプファイルのオープンに失敗しました¥n");

70     exit(1);

71   }

72

73   printf("FIFO に '%s' を書き込みます¥n", buff);
```

株式会社日昇テクノロジー

```
74
75  /* 文字列をパイプに書き込む */
76  nwrite = write(fd, buff, sizeof(buff)); //(11)
77
78  if(wait(NULL)) //子プロセスの終了を待つ
79  {
80    close(fd); //(12)
81    exit(0);
82  }
83
84 }
85
86
87 int main()
88 {
89  pid_t result;
90  /* fork() 関数を呼び出す */
91  result = fork(); //(1)
92
93  /* result の値によって fork()関数の戻り値を判断し、まずはエラー処理を行う */
94  if(result == -1)
95  {
```

株式会社日昇テクノロジー

```
96    printf("Fork エラー¥n");

97  }

98

99  else if (result == 0) /* 戻り値が 0 なら子プロセス */

100 {

101   fifo_read();

101   fifo_read(); //(2)

102 }

103

104 else /* 戻り値が 0 より大きいなら親プロセス */

105 {

106   fifo_write(); //(3)

107 }

108

109 return result;

110 }
```

この例のプロセスを紹介します。まずは main 関数から始めます：

- (1): まず fork 関数を使って子プロセスを作成します。

- (2): 戻り値が 0 なら子プロセスで、fifo_read()関数を実行します。

- (3): 戻り値が 0 より大きいなら親プロセスで、fifo_write()関数を実行します。

- (4): 子プロセスでは、access()関数を使って名前付きパイプが既に存在するかどうかを確認し、存在しなければ適切な権限で作成します。

- (5): mkfifo()関数を呼び出して名前付きパイプを作成します。

株式会社日昇テクノロジー

- (6): open()関数を使って名前付きパイプを読み取り専用でブロックモードで開きます。

- (7): read()関数を使ってパイプの内容を読み取ります。開いたパイプがブロックモードで、パイプにデータが存在しないため、子プロセスはここでブロックされ、パイプにデータが存在するまで待ち、読み取ったデータを表示します。

- (8): 読み取りが完了したら、close()関数を使ってパイプを閉じます。

- (9): 親プロセスは 2 秒間休止し、子プロセスが先に実行されるのを待ちます。この例では、子プロセスでパイプが作成されます。

- (10): FIFO パイプを書き込み専用でブロックモードで開きます。

- (11): パイプに文字列データを書き込みます。書き込みが完了すると、パイプにデータが存在し、ブロックされていた子プロセスが再開し、文字列データを表示します。

- (12): 子プロセスが終了するのを待ち、パイプを閉じます。

fifo のコンパイルとテストプロセスは以下の通りです：

```
# 以下の操作は system_programing/fifo コードディレクトリで行います
# X86 バージョンのプログラムをコンパイル
make
# X86 バージョンのプログラムを実行
./build_x86/fifo_demo
# 開発ボードで実行したい場合は、以下のコマンドでクロスコンパイルを行います
make ARCH=ARM
# クロスコンパイルで生成された armhf アーキテクチャのプログラムは build_ARM ディレクトリにあり、
# 開発ボードにコピーして実行することができます。
```

株式会社日昇テクノロジー

実行結果は以下の通りです：

```
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/syst
em_programing/fifo$ make
gcc -c  -Iincludes  -I. -MD -MF build_x86/.fifo.o.d sources/fifo.c -o buil
d_x86/fifo.o
gcc build_x86/fifo.o -o build_x86/fifo_demo
flyleaf@ebf-dev:~/workdir/linux/debian/embed_linux_tutorial/base_code/syst
em_programing/fifo$ ./build_x86/fifo_demo
***************** read fifo ************************
Write 'this is a fifo test demo' to FIFO
Read 'this is a fifo test demo' from FIFO
***************** close fifo ************************
```

この例は、2 つのプロセス間の通信問題です（この例では親子プロセスを使用していますが、「血縁関係」のないプロセスでも同じ操作が可能です）。つまり、一方のプロセスが FIFO ファイルにデータを書き込み、もう一方のプロセスが FIFO ファイルからデータを読み取ります。複数のプロセスが同時に同一の FIFO ファイルにデータを書き込み、一方のプロセスのみが FIFO ファイルからデータを読み取る場合、どのような状況が発生するかを考えてみてください。データが相互に混ざり合って混乱すると思われるかもしれません。何も処理をしなければ、実際にそうなりますが、FIFO と PIPE の大きな違いの一つは、FIFO が書き込みの原子性を持っていることです。書き込み操作を原子化するにはどうすればよいでしょうか？答えは簡単です。システムは、O_WRONLY（つまりブロックモード）で開かれた FIFO に対して、書き込みデータの長さが PIPE_BUF 以下の場合、全てのバイトを書き込むか、一切書き込まないかのどちらかにすると規定しています。すべての書き込みリクエストがブロックされた FIFO に対して行われ、各書き込みリクエストのデータ長が PIPE_BUF バイト以下であれば、システムはデータが混在することは決してないと保証できます。この特性により、FIFO に書き込む際にこの問題を心配する必要はありません。

それでは、FIFO の応用シナリオは何でしょうか？一つの典型的な応用は Linux のログシステムです。システムのログ情報は/var/log ディレクトリに統一して格納され、これらのログファイルはすべて普通のテキストファイルです。Linux システムでは、普通のファイルは一つまたは複数のプロセスによって何度も開かれ、それぞれ独立した位置オフセットを持っています。複数のプロセスやスレッドが同時にファイルに書き込む場合、相互に調整することなく混乱が生じます。残念ながら、ログを書き込むプロセスは「調

株式会社日昇テクノロジー

整する」ことができません。ログを書き込むプロセスは無関係であり、互斥ロックやセマフォなどの一般的な互斥手段は効果がありません。異なるプロセスのログ情報がログファイルに完全かつ混乱なく転送されるようにするためには、この問題を解決する必要があります。簡単で効率的な解決策は、FIFO を使用して異なるプロセスからのログ情報を受け取り、専用のプロセスが FIFO からデータを取り出して適切なログファイルに書き込むことです。この方法の利点は、ログ情報の互斥的な書き込みに関する追加のコードを何も書く必要がなく、FIFO に書き込むだけでよいことです。バックグラウンドで静かに働くログシステムサービスルーチンがこれらの情報を一つずつ取り出してログファイルに書き込み、FIFO の書き込み原子性がデータの完全性と混乱のない保証を提供します。

第 37 章 メッセージキュー

Linux におけるプロセス間通信手段は基本的に Unix プラットフォームのプロセス間通信手段を継承しています。Unix の発展に大きな貢献をした二大勢力、AT&T のベル研究所と BSD（カリフォルニア大学バークレー校のバークレーソフトウェア配布センター）は、プロセス間通信の焦点を異にしています。

・前者は Unix 初期のプロセス間通信手段をシステマティックに改善・拡張し、「System V IPC」として、通信プロセスを単一コンピュータ内（同一デバイスの異なるプロセス間通信）に限定しました。

・後者はこの制限を超え、ソケット（Socket）を基にしたプロセス間通信メカニズムを形成しました（主に異なるデバイスのプロセス間通信に使用）。Linux は両者を継承しており、「System V IPC」を 有し、さらに「Socket」もサポートしているため、最も成功したと言えます。

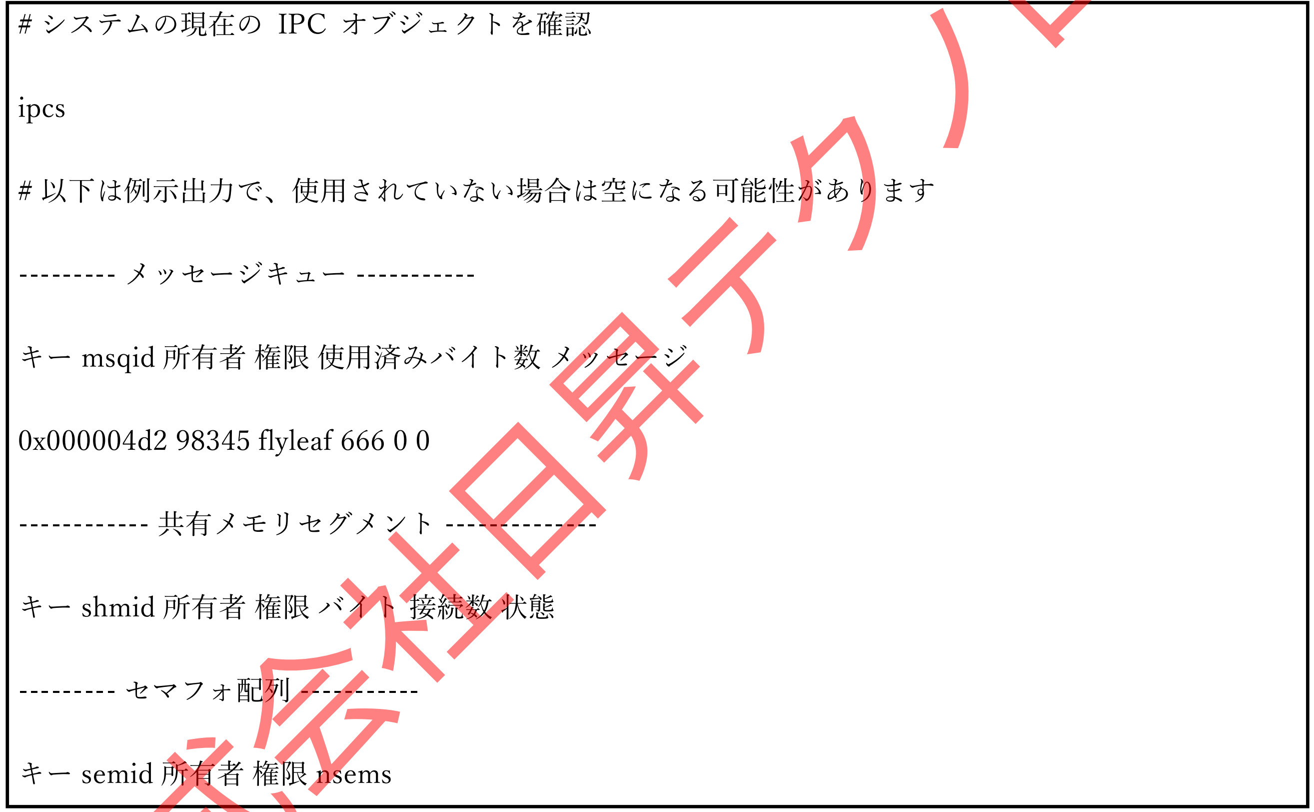
メッセージキュー、共有メモリ、セマフォは「System V IPC」と総称されます。V はローマ数字の 5 で、Unix の AT&T 分岐のバージョンの一つです。通常、これらを IPC オブジェクトと呼び、これらのオブジェクトの操作インターフェースは似ており、システム内で「キー」と呼ばれるキー値を使用して一意に識別されます。また、これらは「持続性」リソースであり、作成後にプロセスが終了しても消えず、特別な関数やコマンドを呼び出して削除するまで持続します。

株式会社日昇テクノロジー

Linux の IPC オブジェクト（メッセージキュー、共有メモリ、セマフォを含む）は内部的にリンクリストで管理され、異なるオブジェクトは IPC 識別子で識別されます。例えば、メッセージキュー識別子 msqid、共有メモリ識別子 shmid、セマフォ識別子 semid です。

ユーザーにとって、カーネルは簡潔なインターフェースを提供し、異なるプロセスは IPC キーワード（key）を通じて具体的なオブジェクトにアクセスできます。

以下のコマンドでシステムの現在の IPC オブジェクトを確認できます。使用されていない場合は空になる可能性があります：



```
# システムの現在の IPC オブジェクトを確認
ipcs
# 以下は例示出力で、使用されていない場合は空になる可能性があります
--------- メッセージキュー -----------
キー msqid 所有者 権限 使用済みバイト数 メッセージ
0x000004d2 98345 flyleaf 666 0 0
------------ 共有メモリセグメント --------------
キー shmid 所有者 権限 バイト 接続数 状態
--------- セマフォ配列 -----------
キー semid 所有者 権限 nsems
```

37.1 メッセージキューの基本概念

メッセージキューは、あるプロセスから別のプロセスへデータブロックを送信する方法を提供します。各データブロックはタイプを持っていると考えられ、受信プロセスは異なるタイプのデータ構造を独立して受信できます。メッセージの送信により、名前付きパイプの同期とブロッキングの問題を回避できます。

株式会社日昇テクノロジー

37.2 メッセージキューとシグナル、パイプの比較

メッセージキューとシグナルの比較：

・シグナルは情報量が少ないが、メッセージキューは大量のカスタマイズされたデータを扱うことができます。

メッセージキューとパイプの比較：

・メッセージキューと名前付きパイプは多くの共通点がありますが、通信するプロセスは関連しなくてもよく、データの送受信方式を通じてデータを伝達します。名前付きパイプでは、データの送信に write()を使用し、受信には read()を使用しますが、メッセージキューでは、データの送信に msgsnd()を使用し、受信には msgrcv()を使用します。メッセージキューでは各データに最大長制限があります。

・メッセージキューは送信および受信プロセスとは独立して存在し、プロセスが終了しても、メッセージキューおよびその内容は削除されません。

・パイプは形式のないバイトストリームのみを扱いますが、メッセージキューは形式化されたバイトストリームを提供し、開発者の作業量を減らすことができます。

・メッセージキューはレコード指向で、メッセージには特定の形式と特定の優先度があり、受信プログラムはメッセージタイプによって選択的にデータを受信できます。これは、名前付きパイプのようにデフォルトで受信するのではなく、メッセージの種類によって受信できるということです。

・メッセージキューはメッセージのランダムアクセスを実現でき、メッセージを先入れ先出しの順序で受信する必要はなく、メッセージのタイプによって受信することもできます。

メッセージキューの実装には、メッセージキューの作成またはオープン、メッセージの送信、メッセージの受信、メッセージキューの制御の 4 つの操作が含まれます。

37.3 メッセージキュー関数説明

Linux カーネルはメッセージキューを使用するための一連の関数を提供しています：

・メッセージキューの作成またはオープンに使用される関数は msgget()で、ここで作成されるメッセ

株式会社日昇テクノロジー

ージキューの数はシステムがサポートできるメッセージキューの数に制限されます。

・メッセージの送信に使用される関数は msgsnd()で、これはメッセージを開かれたメッセージキューの末尾に送信します。

・メッセージの受信に使用される関数は msgrcv()で、これはメッセージキューからメッセージを取り出します。FIFO と異なり、特定のメッセージを取り出すことができます。

・最後に、メッセージキューを制御するために使用される関数は msgctl()で、これは複数の機能を実行することができます。

37.3.1 msgget() 関数

メッセージを送受信する前に、具体的なメッセージキューオブジェクトが必要です。msgget()関数の役割は、メッセージキューオブジェクトを作成または取得し、メッセージキュー識別子を返すことです。関数のプロトタイプは以下の通りです：

```
int msgget(key_t key, int msgflg);
```

成功するとキュー ID を返し、失敗すると-1 を返します。2 つの入力パラメータは以下の通りです：

・key：メッセージキューのキーワード値で、複数のプロセスがこれを使用して同じメッセージキューにアクセスできます。たとえば、送受信プロセスが同じキー値を使用すると、同じメッセージキューを使用して通信できます。ここには IPC_PRIVATE という特別な値があり、これは現在のプロセスのプライベートメッセージキューを作成するために使用されます。

・msgflg：作成されるメッセージキューのモードフラグパラメータを示し、主に IPC_CREAT、IPC_EXCL、および権限モードがあります。

- IPC_CREAT が真の場合：カーネル内に key と等しいキーワードのメッセージキューが存在しない場合、新たにメッセージキューを作成します。もし既にそのようなメッセージキューが存在する場合、そのメッセージキューの識別子を返します。

株式会社日昇テクノロジー

- IPC_CREAT | IPC_EXCL の場合：カーネル内に key と等しいキーワードのメッセージキューが存在しない場合、新たにメッセージキューを作成します。もし既にそのようなメッセージキューが存在する場合は、エラーを報告します。

- mode は IPC オブジェクトのアクセス権限を指します。これは Linux ファイルの数値権限リスト現法を使用します。例えば、0600、0666 などです。

これらのパラメータは "|" 演算子を使用して組み合わせることができます。例えば、msgflag に IPC_CREAT | 0666 を使用した場合、メッセージキューを作成または既存のメッセージキューの識別子を返し、そのメッセージキューのアクセス権限を 0666 に設定します。これは、メッセージの所有者、グループのユーザー、その他のユーザーがメッセージに対して読み書きができることを意味します。

注意：

- msgflg はビットマスクなので、IPC_CREAT、IPC_EXCL、権限 mode をビットまたはで重ね合わせることができます。例えば、msgget(key, IPC_CREAT | 0666);は、key に対応するメッセージキューが存在しない場合にはそれを作成し、権限を 0666 に指定します。既に存在する場合は、メッセージキュー ID を直接取得します。ここでの 0666 は、Linux ファイル権限の数値リスト現法を使用しています。

- 権限には読み取りと書き込みのみが有効で、実行権限は無効です。例えば、0777 と 0666 は等価です。

- key が IPC_PRIVATE に指定された場合、システムは自動的に未使用の key を生成して新しいメッセージキューオブジェクトに対応させます。このメッセージキューは通常、プロセス内部間の通信に使用されます。

- この関数は以下のエラーコードを返す可能性があります：

- EACCES：指定されたメッセージキューは既に存在しますが、呼び出しプロセスにはアクセス権限がありません。

株式会社日昇テクノロジー

- EEXIST：key で指定されたメッセージキューは既に存在し、msgflg に IPC_CREAT と IPC_EXCL フラグが同時に指定されています。

- ENOENT：key で指定されたメッセージキューが存在せず、msgflg に IPC_CREAT フラグが指定されていません。

- ENOMEM：メッセージキューを作成する必要がありますが、メモリが不足しています。

- ENOSPC：メッセージキューを作成する必要がありますが、システムの制限に達しています。

37.4 メッセージの送受信

37.4.1 msgsnd() 送信関数

この関数の主な役割は、メッセージキューにメッセージを書き込むこと、つまりメッセージを送信することです。関数のプロトタイプは以下の通りです：

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

パラメータ説明：

- msqid：メッセージキューの識別子。

- msgp：キューに送るメッセージ。msgp は任意の型の構造体である可能性がありますが、最初のフィールドは long 型でなければなりません。これは、送信されるメッセージのタイプをリストし、msgrcv()関数はこれに基づいてメッセージを受信します。msgp の定義例は以下の通りです：

```
/* msgp の定義例 */
struct s_msg{
  long type; /* 0 より大きい必要があります, メッセージタイプ */
  char mtext[1]; /* メッセージ本文、任意の型が可能 */
} msgp;
```

- msgsz：送信するメッセージのサイズで、メッセージタイプが占める 4 バイトは含まれません。つま

株式会社日昇テクノロジー

り、mtext の長さです。

- msgflg：0 の場合、メッセージキューが満杯のとき、msgsnd()関数はメッセージが書き込めるまでブロックされます。IPC_NOWAIT の場合、メッセージキューが満杯のとき、msgsnd()関数は待たずにすぐに戻ります。IPC_NOERROR の場合、送信されたメッセージが size バイトより大きい場合、そのメッセージは切り捨てられ、切り捨てられた部分は破棄され、送信プロセスには通知されません。

- 戻り値：成功すると 0 を返し、失敗すると-1 を返し、エラー原因は errno に格納されます。エラーコード：

- EAGAIN：msgflg が IPC_NOWAIT に設定され、メッセージキューが満杯です。

- EIDRM：msqid で識別されるメッセージキューが削除されました。

- EACCESS：メッセージキューへの書き込み権限がありません。

- EFAULT：msgp が無効なメモリアドレスを指しています。

- EINTR：メッセージキューが満杯で待機中にシグナルによって中断されました。

- EINVAL：無効な msqid、msgsz、またはメッセージタイプ type が 0 未満です。

msgsnd()はブロッキング関数であり、メッセージキューの容量が満杯またはメッセージ数が満杯の場合にブロックされます。メッセージキューが削除された場合は EIDRM エラーを返し、シグナルによって中断された場合は EINTR エラーを返します。

IPC_NOWAIT が設定されている場合、メッセージキューが満杯またはメッセージ数が満杯のときに-1 を返し、EAGAIN エラーを設定します。

msgsnd()のブロックが解除される条件は以下の 3 つです：

- メッセージキューにそのメッセージを収容できるスペースができた場合。

- msqid で識別されるメッセージキューが削除された場合。

- msgsnd 関数を呼び出したプロセスがシグナルによって中断された場合。

株式会社日昇テクノロジー

37.4.2 msgrcv() 受信関数

msgrcv()関数は、msqid で識別されるメッセージキューからメッセージを読み込み、msgp にメッセージを格納します。読み取った後、このメッセージをメッセージキューから削除します。これは俗に言うメッセージの受信です。関数のプロトタイプは以下の通りです：

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

パラメータ説明：

- msqid：メッセージキュー識別子。

- msgp：メッセージを格納する構造体で、msgsnd()関数で送信されるタイプと同じ構造体タイプでなければなりません。

- msgsz：受信するメッセージのサイズで、メッセージタイプが占める 4 バイトを含みません。

- msgtyp は複数の選択肢があります：0 の場合は最初のメッセージを受信します。0 より大きい場合は、msgtyp と等しいタイプの最初のメッセージを受信します。0 より小さい場合は、msgtyp の絶対値以下のタイプの最初のメッセージを受信します。

- msgflg は受信の処理方法を設定するために使用されます。取り得る値は以下の通りです：

- 0: ブロッキング受信。該当するタイプのメッセージがない場合、msgrcv 関数はブロックされ続けます。

- IPC_NOWAIT：メッセージキューに該当するタイプのメッセージがない場合、関数は即座に戻ります。この時、エラーコードは ENOMSG です。

- IPC_EXCEPT：msgtype と組み合わせて使用し、msgtype とは異なるタイプのメッセージキューの最初のメッセージを返します。

- IPC_NOERROR：キュー内の条件を満たすメッセージの内容が要求されたサイズバイトより大きい場合、そのメッセージを切り詰め、切り詰めた部分は破棄されます。

株式会社日昇テクノロジー

- 戻り値：msgrcv()関数がメッセージの受信に成功すると、実際に読み取られたメッセージデータの長さを返します。失敗した場合は-1 を返し、エラーの原因は error に格納されます。エラーコード：

- E2BIG：メッセージデータの長さが msgsz より大きく、msgflag に IPC_NOERROR が設定されていない場合。

- EIDRM：msqid で識別されるメッセージキューが削除されています。

- EACCESS：そのメッセージキューを読み取る権限がありません。

- EFAULT：パラメータ msgp が無効なメモリアドレスを指しています。

- ENOMSG：パラメータ msgflg に IPC_NOWAIT が設定され、メッセージキューに読み取り可能なメッセージがありません。

- EINTR：メッセージキュー内のメッセージの読み取りを待っている間にシグナルによって中断されました。

msgrcv()関数のブロック解除条件も三つあります：

- メッセージキューに条件を満たすメッセージが存在します。

- msqid で識別されるメッセージキューが削除されます。

- msgrcv()関数を呼び出したプロセスがシグナルによって中断されます。

37.4.3 msgctl() メッセージキュー操作

n メッセージキューはユーザーによって操作することができます。例えば、メッセージキューの関連属性を設定または取得する場合、msgctl()関数を使用して処理します。関数のプロトタイプは以下の通りです：

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

パラメータ説明：

- msqid：メッセージキュー識別子。

- cmd は使用する操作コマンドを設定します。取り得る値は複数あります：

株式会社日昇テクノロジー

- IPC_STAT：その MSG の情報を取得し、取得した情報は msqid_ds 型の構造体 buf に格納されます。

- IPC_SET：メッセージキューの属性を設定します。設定する属性は msqid_ds 型の構造体 buf に事前に格納されている必要があります。設定可能な属性には msg_perm.uid、msg_perm.gid、msg_perm.mode、msg_qbytes が含まれ、これらは msqid_ds 構造体に格納されます。

- IPC_RMID：その MSG を即座に削除し、その MSG 上でブロックされているすべてのプロセスを起こします。第三パラメータは無視されます。

- IPC_INFO：現在のシステム内の MSG の制限値情報を取得します。

- MSG_INFO：現在のシステム内の MSG の関連リソース消費情報を取得します。

- MSG_STAT：IPC_STAT と同じですが、msgid はカーネル内ですべてのメッセージキュー情報を記録する配列のインデックスです。したがって、すべてのインデックスを反復することで、システム内のすべてのメッセージキューの関連情報を取得することができます。

- buf：関連情報構造体バッファ。

- 戻り値：

- 成功：0

- エラー：-1、エラーの原因は error に格納されます。エラーコード：

- EACCESS：cmd が IPC_STAT で、そのメッセージキューを読み取る権限がありません。

- EFAULT：パラメータ buf が無効なメモリアドレスを指しています。

- EIDRM：msqid で識別されるメッセージキューが削除されています。

- EINVAL：無効なパラメータ cmd または msqid です。

- EPERM：cmd が IPC_SET または IPC_RMID で、十分な権限がない場合です。

37.5 メッセージキュー例

次に、メッセージキューの使用例を紹介します。一般的な使用方法は以下の通りです。

株式会社日昇テクノロジー

送信者:

1. メッセージキューの ID を取得します。

2. 識別子を伴う特別な構造体にデータを入れ、メッセージキューに送信します。

受信者:

1. メッセージキューの ID を取得します。

2. 指定された識別子のメッセージを読み出します。

送信者と受信者がメッセージキューを使用しなくなったら、システムリソースを解放するためにそれを削除することが重要です。

この実験では、血縁関係のない 2 つのプロセスがメッセージキューを介してメッセージをやり取りします。一方のプロセスがメッセージを送信し、もう一方のプロセスがメッセージを受信して表示します。

37.5.1 送信プロセス

以下は送信プロセスの例です：

以下は、指定された C 言語のプログラムコードの日本語訳です。このコードはメッセージキューを使用してプロセス間通信を行う送信プロセスの例を示しています。

```
1  #include <sys/types.h>

2  #include <sys/ipc.h>

3  #include <sys/msg.h>

4  #include <stdio.h>

5  #include <stdlib.h>

6  #include <unistd.h>

7  #include <string.h>
```

株式会社日昇テクノロジー

```
8

9

10 #define BUFFER_SIZE 512

11

12 struct message

13 {

14   long msg_type;

15   char msg_text[BUFFER_SIZE];

16 };

17 int main()

18 {

19   int qid;

20   struct message msg;

21

22   /* メッセージキューの作成 */

23   if ((qid = msgget((key_t)1234, IPC_CREAT | 0666)) == -1)

24   {

25     perror("msgget エラー¥n");

26     exit(1);

27   }

28

29   printf("キューを開く %d¥n", qid);
```

株式会社日昇テクノロジー

```
30
31  while(1)
32  {
33    printf("キューに何かメッセージを入力してください：");
34    if ((fgets(msg.msg_text, BUFFER_SIZE, stdin)) == NULL)
35    {
36      printf("¥n メッセージ入力終了。¥n");
37      exit(1);
38    }
39
40    msg.msg_type = getpid();
41    /* メッセージをメッセージキューに追加 */
42    if ((msgsnd(qid, &msg, strlen(msg.msg_text), 0)) < 0)
43    {
44      perror("¥n メッセージ送信エラー。¥n");
45      exit(1);
46    }
47    else
48    {
49      printf("メッセージを送信しました。¥n");
50    }
51
```

株式会社日昇テクノロジー

```
52    if (strncmp(msg.msg_text, "quit", 4) == 0)
53    {
54      printf("¥n メッセージ入力を終了します。¥n");
```

```
55      break;
56    }
57  }
58
59  exit(0);
60 }
```

このコードのポイントは以下の通りです：

- 22 行目で、msgget()関数を呼び出し、キー値 1234 のメッセージキューを作成/取得します。このキューの属性"0666"は、誰でも読み書きできることを意味し、作成/取得したキュー ID は変数 qid に格納されます。

- 47 行目で、msgsnd()関数を呼び出し、プロセス ID とユーザーが入力した文字列を msg 構造体を介して先に取得した qid キューに追加します。

- 51 行目で、ユーザーが"quit"というメッセージを送信した場合、ループを抜けてプロセスを終了します。

株式会社日昇テクノロジー

37.5.2 受信プロセス

受信プロセスの例は以下の通りです：

リスト 2: メッセージキュー受信プロセス（対応するコードリポジトリ/system_programing/msg/msg_recv/sources/msg.c ファイル）

```
1 #include <sys/types.h>

2 #include <sys/ipc.h>

3 #include <sys/msg.h>

4 #include <stdio.h>
```

```
5 #include <stdlib.h>

6 #include <unistd.h>

7 #include <string.h>

8

9 #define BUFFER_SIZE 512

10

11 struct message

12 {

13   long msg_type;

14   char msg_text[BUFFER_SIZE];

15 };

16

17 int main()

18 {

19   int qid;

20   struct message msg;

21

22   /* メッセージキューの作成 */

23   if ((qid = msgget((key_t)1234, IPC_CREAT | 0666)) == -1)
```

株式会社日昇テクノロジー

```
24  {
25    perror("msgget");
26    exit(1);
```

```
27  }
28
29  printf("キューを開く %d¥n", qid);
30
31  do
32  {
33    /* メッセージキューからのメッセージ読み込み */
34    memset(msg.msg_text, 0, BUFFER_SIZE);
35
36    if (msgrcv(qid, (void*)&msg, BUFFER_SIZE, 0, 0) < 0)
37    {
38      perror("msgrcv");
39      exit(1);
40    }
41
42    printf("プロセス %ld からのメッセージ: %s", msg.msg_type, msg.msg_text);
43
44  } while(strncmp(msg.msg_text, "quit", 4));
45
```

株式会社日昇テクノロジー

```
46  /* システムカーネルからメッセージキューを削除 */
47  if (msgctl(qid, IPC_RMID, NULL) < 0)
48  {
49    perror("msgctl");
50    exit(1);
51  }
52  else
53  {
54    printf("メッセージキュー %d を削除しました。¥n", qid);
55  }
56
57  exit(0);
58
59 }
```

このコードの重要な説明は以下の通りです：

- 23 行目で、msgget()関数を呼び出してキュー qid を作成/取得します。送信プロセスと完全に同じであることに注意してください。どちらのプロセスが先に実行されても、キー値 1234 のキューが存在しない場合は作成されます。これにより、実験時の 2 つのプロセスには起動順序の要件がなくなります。

- 36 行目で、ループ内で msgrcv()関数を呼び出して qid キューの msg 構造体メッセージを受信します。ここではブロッキングモードで受信しており、キューにメッセージがない場合はこの行で待機します。

株式会社日昇テクノロジー

- 47 行目で、前に受信したユーザーのメッセージが quit であれば、ループを抜け、この行で msgctl() を呼び出してメッセージキューを削除し、プロセスを終了します。

37.5.3 コンパイル及びテスト

例のコードはそれぞれ、付属のコードリポジトリ/system_programing/msg/の msg_send および msg_recv ディレクトリにあります。2 つのプロセスをコンパイルして実行すると、以下のような実験現象が観察できます：

37.5.3.1 送信プロセス

送信メッセージプロセスを実行すると、送信するメッセージを入力するように促されます。任意のメッセージを入力して Enter キーを押すことでメッセージの入力を完了できます。quit を入力するか、Ctrl+D、Ctrl+C を使用してプロセスを終了できます。

```
# 以下の操作は system_programing/msg/msg_send コードディレクトリで行います
# X86 バージョンの送信プロセスをコンパイル
make
# X86 バージョンの送信プロセスを実行
./build_x86/msg_send_demo
# メッセージを入力してテスト
Open queue 98345
Enter some message to the queue:embedfire
Send message.
Enter some message to the queue:test
Send message.
Enter some message to the queue:hello world
```

株式会社日昇テクノロジー

Send message.

quit メッセージを送信してプロセスを終了

Enter some message to the queue:quit

Send message.

Quit get message.

37.5.3.2 メッセージキューの確認

ipcs -q コマンドを使用して、システムに存在するメッセージキューを確認できます。上記のキューが閉じられていない場合、その結果は以下のようになります：

```
# システムに現存するキューを確認
ipcs -q
# 出力例：
--------- メッセージキュー -----------
キー      msqid  所有者  権限  使用済みバイト数 メッセージ
0x000004d2 98345  flyleaf 666   0          0
# キー値 0x04d2(1234)、qid 98345 はプロセス内で作成されたものと一致します。
```

37.5.3.3 受信プロセス

新しいターミナルを開き、受信メッセージプロセスをコンパイルして実行します。送信メッセージプロセスからメッセージを入力して送信すると（Enter キーを押して送信）、受信メッセージプロセスは入力されたメッセージを表示します。メッセージがない場合は、受信プロセスはブロックされて待機します。quit メッセージを受信するとプロセスが終了します。

```
# 以下の操作は system_programing/msg/msg_recv コードディレクトリで行います
# X86 バージョンの受信プロセスをコンパイル
```


株式会社日昇テクノロジー

make

X86 バージョンの受信プロセスを実行

./build_x86/msg_recv_demo

受信したメッセージ

```
Open queue 98345
The message from process 21023 : embedfire
The message from process 21023 : test
The message from process 21023 : hello world
The message from process 21023 : quit
Delete msg qid: 98345.
```

この例では、送信プロセスが quit メッセージで終了しない場合（例えば Ctrl+C や Ctrl+D を使用するなど）、受信プロセスはメッセージキューをアクティブに削除しません。この場合、ipcs -q コマンドでメッセージキューが依然として存在していることが確認でき、ipcrm -q [メッセージキュー qid]で削除できます。

第 38 章 System V IPC セマフォ

本章では、特に明記されていない限り、「セマフォ」とは System V IPC セマフォのことを指します。これは、後続の章で「POSIX セマフォ」と区別するためです。

38.1 プロセスセマフォの基本概念

セマフォは、以前に紹介されたシグナル、パイプ、FIFO、メッセージキューとは異なり、本質的にはカウンターです。これは、共有データオブジェクトへの複数プロセス間のアクセスを調整するために使用されます。主な目的はデータの送信ではなく、共有リソース（セマフォもクリティカルリソースに属します）

株式会社日昇テクノロジー

を保護し、そのクリティカルリソースが一度に 1 つのプロセスだけが独占的に使用できるようにすることです。なぜグローバル変数を使用しないのでしょうか？それは、プロセス間でグローバル変数を共有することはできないからです。プロセス間は互いに独立しており、参照カウントの原子操作を保証することもできません。したがって、システムが提供するセマフォを使用します。

38.2 セマフォの動作原理

セマフォは、待機（P 操作）とシグナル送信（V 操作）の 2 つの操作のみを行うことができます。ロック動作は P 操作であり、アンロックは V 操作です。P 操作をリソースの要求として、V 操作をリソースの解放として直接理解することができます。PV 操作はコンピュータオペレーティングシステムが提供する基本的な機能の 1 つです。その動作は以下の通りです：

- P 操作：利用可能なリソースがある場合（セマフォ値が 0 より大きい）、リソースを 1 つ使用し（セマフォ値を 1 減らし、クリティカルセクションのコードに入る）; 利用可能なリソースがない場合（セマフォ値が 0）、ブロックされ、システムがそのプロセスにリソースを割り当てるまで待機します（待機キューに入り、リソースがそのプロセスに割り当てられるまで待機します）。これは、車を駐車場に入れる前に、警備員に駐車カードを要求するようなものです。P 操作はリソースの要求であり、要求が成功するとリソース数（空き駐車スペース）が 1 つ減少します。要求が失敗すると、出口で待つか、または去るしかありません。

- V 操作：そのセマフォの待機キューにプロセスが待機している場合、ブロックされたプロセスを 1 つ起こします。プロセスが待機していない場合は、リソースを 1 つ解放します（セマフォ値を 1 増やします）。これは、駐車場から出るときに空き駐車スペースが 1 つ増えるようなものです。

例えば、2 つのプロセスがセマフォ sem を共有しており、sem の利用可能なセマフォの数値が 1（リソース数が 1）である場合、いずれかのプロセスが P 操作を実行した場合、そのプロセスはセマフォを取得し、クリティカルセクションに入ることができ、sem を 1 減らします。一方、2 番目のプロセスはクリテ

株式会社日昇テクノロジー

ィカルセクションに入ることが阻止されます。なぜなら、それが P 操作を試みたときに sem が 0 であるため、1 番目のプロセスがクリティカルセクションを離れて V 操作を実行してセマフォを解放するまで、2 番目のプロセスは中断され待機状態になるからです。この時、2 番目のプロセスは実行を再開することができます。

セマフォでの PV 操作は、クリティカルリソースを保護する必要があるため、すべて原子操作として行われます。

注：原子操作とは、単一の指示で行われる操作で、その実行は中断されません。

簡単に言うと、カーネルはこのセマフォ（カウンター）に対して加算減算操作を行い、操作時にいくつかの基本的な動作原則に従います。すなわち、セマフォ（カウンター）に対する加算操作は即時に返され、減算操作はセマフォ（カウンター）の現在値が減算可能かどうかを判断する必要があります（セマフォの現在値が 1 以上である必要があります）。条件を満たしていれば減算操作を行い、そうでない場合はプロセスをブロックして待機させます。システム内の別のプロセスがそのセマフォに対して V 操作を行うまでです。

38.2.1 セマフォの作成または取得

38.3 semget 作成/取得関数

semget 関数は、セマフォを作成するか、既に作成されているセマフォを取得するための関数です。成功すると対応するセマフォ識別子を返し、失敗すると-1 を返します。関数のプロトタイプは以下の通りです：

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>
```

株式会社日昇テクノロジー

int semget(key_t key, int nsems, int semflg);

パラメータ説明：

- key：メッセージキューと同様、パラメータ key はシステム内のセマフォを識別するために使用されます。指定された key が既に存在する場合は、そのセマフォを開きます。この場合、nsems パラメータは 0 に、semflg パラメータも 0 に設定されます。特に、IPC_PRIVATE を使用して key のないセマフォを作成することができます。

- nsems：このパラメータは、セマフォを作成する際に使用可能なセマフォの数を指定します。

- semflg：semflg パラメータはフラグビットを指定するために使用されます。メッセージキューと同様に、主に IPC_CREAT、IPC_EXCL および権限 mode があります。IPC_CREAT フラグを使用して新しいセマフォを作成すると、そのセマフォが既に存在していても（同じキー値のセマフォがシステムに既に存在していても）エラーにはなりません。IPC_EXCL フラグを同時に使用すると、新しいユニークなセマフォを作成できます。この場合、そのセマフォが既に存在すると、関数はエラーを返します。

セマフォの作成は、以下のシステム情報にも影響されます：

- SEMMNI：システム内のセマフォの総数の最大値。

- SEMMSL：各セマフォ内のセマフォ要素の最大数。

- SEMMNS：システム内のすべてのセマフォのセマフォ要素の総数の最大値。

Linux システムでは、上記の情報は`ipcs -l`コマンドで確認できます。

38.4 セマフォ操作

38.4.1 semop() PV 操作関数

Linux は semop()関数を提供して、セマフォに対して PV 操作を行います。関数のプロトタイプは以下の通りです：

株式会社日昇テクノロジー

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

パラメータ説明：

- semid：System V セマフォの識別子で、セマフォを識別します。

- sops：struct sembuf 構造体の配列へのポインタで、セマフォ操作配列です。構造体のプロトタイプは以下の通りです：

```
struct sembuf
{
   unsigned short int sem_num; /* セマフォの番号、0 ～ nsems-1 */
   short int sem_op; /* セマフォに対する操作、>0, 0, <0 */
   short int sem_flg; /* 操作フラグ：0、IPC_WAIT、SEM_UNDO */
};
```

- sem_num はセマフォの中でのセマフォ番号を識別します。0 は最初のセマフォ、1 は 2 番目のセマフォ、nsems-1 は最後のセマフォをリストします。

- sem_op はセマフォに対する操作のタイプを示します。セマフォに対する操作には 3 つのタイプがあります：

- sem_op が正の場合、プロセスがリソースの使用を終えてリソースを返却することを意味します。つまり、セマフォに対して V 操作を行い、返却するリソースの数は sem_op によって決まります。システムは sem_op の値をそのセマフォの現在値 semval に加算します。特に、sem_flg が SEM_UNDO フラグを指定している場合、このプロセスのそのセマフォに対する調整値から sem_op を減算します。

- sem_op が負の場合、プロセスがリソースを使用したいことを意味します。セマフォの現在値

株式会社日昇テクノロジー

semval が-sem_op 以上の場合、semval から sem_op の絶対値を減算し、そのプロセスに対応する数のリソースを割り当てます。特に、SEM_UNDO が指定されている場合、sem_op の絶対値もそのプロセスのそのセマフォの調整値に加算されます。semval が-sem_op 未満の場合、該当セマフォの待機プロセス数が増加し、呼び出しプロセスはブロックされます。

- sem_op が 0 の場合、プロセスはセマフォの現在値 semval が 0 になるまでブロックされ待機します。

- sem_flg は、セマフォ操作の属性フラグで、指定できるパラメータには IPC_NOWAIT と SEM_UNDO があります。0 の場合は通常操作を意味し、SEM_UNDO が指定された場合、プロセスによるセマフォの調整値を維持し、プロセスが終了するときに自動的にセマフォに対する操作を元に戻します。IPC_WAIT が指定された場合、セマフォの操作が非ブロッキングになります。つまり、このフラグが指定されている場合、セマフォの値が条件を満たさない状況で呼び出しプロセスはブロックされずに、-1 を直接返し、errno を EAGAIN に設定します。

では、セマフォの調整値とは何でしょうか？実際には、特定のプロセスに対するセマフォの調整値を指します。sembuf 構造体の sem_flag が SEM_UNDO に指定された後でのみ、sem_op による更新に伴ってセマフォの調整値が変更されます。もっと簡単に言うと、あるプロセスに対して、SEM_UNDO が指定された後、セマフォの現在の値に対する変更はすべてセマフォの調整値に反映され、そのプロセスが終了するとき、カーネルはセマフォの調整値に基づいて以前のセマフォの値を再び復元します。SEM_UNDO 操作は、プロセスが終了時にセマフォを解放しないことによって発生するデッドロックを防ぐことができます。

- nsops は、上記の sops 配列の数をリストします。例えば、sops 配列が 1 つだけの場合、nsops は 1 に設定されます。

株式会社日昇テクノロジー

38.4.2 semctl() 属性関数

semctl 関数は、セマフォセットに対する一連の制御操作を行います。操作コマンド cmd に応じて異なる操作を実行します。第 4 引数はオプションです。関数のプロトタイプは以下の通りです：

```
int semctl(int semid, int semnum, int cmd, ...);
```

- semid：System V セマフォの識別子；

- semnum：セマフォセット内の semnum 番目のセマフォを示します。その範囲は 0~nsems-1 です。

- cmd：操作コマンドで、主に以下のコマンドがあります：

  - IPC_STAT：このセマフォセットの semid_ds 構造を取得し、第 4 引数の buf に格納します。

  - IPC_SET：第 4 引数の buf を使用して、セマフォセットに関連付けられた semid_ds の中のセマフォセットの権限を sem_perm の uid、gid、mode に設定します。

  - IPC_RMID：システムからそのセマフォセットを削除します。

  - GETVAL：semnum 番目のセマフォの値を返します。

  - SETVAL：semnum 番目のセマフォの値を設定します。その値は第 4 引数の val によって指定されます。

  - GETPID：semnum 番目のセマフォの sempid、最後に操作した pid を返します。

  - GETNCNT：semnum 番目のセマフォの semncnt を返します。semval が現在の値より大きくなることを待っているスレッドの数です。

  - GETZCNT：semnum 番目のセマフォの semzcnt を返します。semval が 0 になるのを待っているスレッドの数です。

  - GETALL：セマフォセット内のすべてのセマフォの値を取得し、array が指す配列に結果を格納します。

  - SETALL：arg.array が指す配列の値に従って、セット内のすべてのセマフォの値を設定します。

株式会社日昇テクノロジー

第 4 引数はオプションで、使用する場合、その型は union semun で、特定のコマンドに対するユニオン体です。具体的には以下の通りです：

```
union semun {
    int val; /* SETVAL 用の値 */
    struct semid_ds *buf; /* IPC_STAT, IPC_SET 用のバッファ */
    unsigned short *array; /* GETALL, SETALL 用の配列 */
    struct seminfo *__buf; /* IPC_INFO 用のバッファ(Linux 固有) */
};
```

38.5 セマフォ例

System V のセマフォ関連の関数呼び出しインターフェースは比較的複雑なので、本例ではそれを単一セマフォのいくつかの基本関数に包装しています。これらの関数は sem.c ファイルの内容として別途実装され、また外部呼び出し用のヘッダファイル sem.h も実装されています。具体的な実装は以下の通りです：

リスト 1: セマフォ操作のカプセル化（対応するコードリポジトリ /system_programing/systemV_sem/sources/sem.c ファイル）

```
1 #include <sys/sem.h>
2 #include <sys/ipc.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <string.h>
7 #include <sys/shm.h>
8 #include <sys/stat.h>
9 #include <fcntl.h>
```

株式会社日昇テクノロジー

```
10 #include <errno.h>

11

12 #include "sem.h"

13

14

15 /* セマフォの初期化（値の設定）関数 */

16 int init_sem(int sem_id, int init_value)

17 {

18     union semun sem_union;

19     sem_union.val = init_value; /* init_value は初期値 */
```

```
20

21     if (semctl(sem_id, 0, SETVAL, sem_union) == -1)

22     {

23         perror("セマフォの初期化エラー");

24         return -1;

25     }

26

27     return 0;

28 }

29

30 /* システムからセマフォを削除する関数 */

31 int del_sem(int sem_id)
```

株式会社日昇テクノロジー

```
32 {
33     union semun sem_union;
34     if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
35     {
36         perror("セマフォの削除エラー");
37         return -1;
38     }
39 }
40
41 /* P 操作関数 */
```

```
42 int sem_p(int sem_id)
43 {
44     struct sembuf sops;
45     sops.sem_num = 0; /* 単一のセマフォの番号は 0 であるべき */
46     sops.sem_op = -1; /* P 操作を示す */
47     sops.sem_flg = SEM_UNDO; /* プロセスが終了すると、システムがセマフォを元に戻す */
48
49     if (semop(sem_id, &sops, 1) == -1)
50     {
51         perror("P 操作エラー");
52         return -1;
53     }
```

株式会社日昇テクノロジー

```
54    return 0;
55 }
56
57 /* V 操作関数 */
58 int sem_v(int sem_id)
59 {
60    struct sembuf sops;
61    sops.sem_num = 0; /* 単一のセマフォの番号は 0 であるべき */
62    sops.sem_op = 1; /* V 操作を示す */
63    sops.sem_flg = SEM_UNDO; /* プロセスが終了すると、システムがセマフォを元に戻す */
```

```
64
65    if (semop(sem_id, &sops, 1) == -1)
66    {
67       perror("V 操作エラー");
68       return -1;
69    }
70    return 0;
71 }
```

それぞれ、セマフォの初期化関数 sem_init()、セマフォを削除する関数 sem_del()、P 操作関数 sem_p()、および V 操作関数 sem_v()です。具体的な説明は以下の通りです：

- sem_init：初期化関数で、与えられたパラメータに基づいてセマフォの初期値を設定し、初期利用可能リソース数を設定します。関数内部では semctl()を呼び出して SETVAL コマンドを使用し、

株式会社日昇テクノロジー

sem_union 型の sem_union 変数にセマフォの初期値を設定します。

- sem_del：セマフォを削除する関数で、semctl()を呼び出して IPC_RMID コマンドを使用して指定されたセマフォを削除します。

- sem_p：P 操作関数で、semop()を呼び出して調整値を設定し、sops.sem_op の値を-1 に設定します。これは、P 操作ごとにセマフォの値を 1 減らすことを意味します。

- sem_v：V 操作関数で、semop()を呼び出して調整値を設定し、P 操作関数との違いは sops.sem_op の値が+1 であることを意味します。これは、V 操作ごとにセマフォの値を 1 増やすことを意味します。

上記の操作関数を使用して、以下のテスト例を作成します。まず、子プロセスを作成し、次にセマフォを使用して 2 つのプロセス（親子プロセス）間の実行順序を制御します。

株式会社日昇テクノロジー

リスト 2: セマフォ操作のヘッダーファイル（付属コードリポジトリ /system_programing/systemV_sem/sources/test.h ファイル）

```
1 #include <sys/types.h>
2 #include <sys/shm.h>
3 #include <sys/sem.h>
4 #include <sys/ipc.h>
5 #include <unistd.h>
6 #include <stdlib.h>
7 #include <stdio.h>
8 #include <string.h>
9 #include <sys/stat.h>
```

```
10 #include <fcntl.h>

11 #include <errno.h>

12

13 #include "sem.h"

14

15 #define DELAY_TIME 3 /* デモンストレーションの効果を高めるため、数秒間待機します */

16

17 int main(void)

18 {

19 pid_t result;

20 int sem_id;
```

```
21

22 sem_id = semget((key_t)6666, 1, 0666 | IPC_CREAT); /* セマフォを作成する */

23

24 init_sem(sem_id, 0);

25

26 /* fork() 関数を呼び出す */

27 result = fork();

28 if(result == -1)

29 {

30 perror("Fork エラー¥n");

31 }
```

株式会社日昇テクノロジー

```
32 else if (result == 0) /* 戻り値が 0 なら子プロセス */

33 {

34 printf("子プロセスは数秒間待機します...¥n");

35 sleep(DELAY_TIME);

36 printf("子プロセス内での戻り値は %d (PID = %d)¥n", result, getpid());

37

38 sem_v(sem_id);

39 }

40

41 else /* 戻り値が 0 より大きいなら親プロセス */

42 {
```

```
43 sem_p(sem_id);

44 printf("親プロセス内での戻り値は %d (PID = %d)¥n", result, getpid());

45

46 sem_v(sem_id);

47

48 del_sem(sem_id);

49 }

50

51 exit(0);

52 }
```

コードの説明は以下の通りです：

株式会社日昇テクノロジー

- 22 行目：semget()を呼び出して、権限 0666（すべてのユーザーが読み書き可能）でセマフォを作成します。

- 24 行目：init_sem()を呼び出してセマフォの値を 0 に初期化します。

- 27 行目：fork 関数を使用して子プロセスを作成します。

- 32～38 行目：子プロセスが一定時間スリープした後、スリープが終わると sem_v を使用してセマフォに 1 を加えます。

- 41～49 行目：親プロセスは sem_p()を使用してセマフォを待ち、セマフォを得た後に情報を出力します。

この例の結果は、親プロセスが子プロセスによってセマフォが解放された後に実行されることを模擬しています。これは、一方のプロセスがリソースを作成し、もう一方のプロセスがリソースを待つという調整プロセスです。

38.5.1 実験操作

この実験のコードは、付属コードリポジトリ/system_programing/systemV_sem ディレクトリに保存されています。コンパイルと実行のプロセスは以下の通りです：

```
# 以下の操作を system_programing/systemV_sem コードディレクトリで行います
# X86 版プログラムをコンパイル
make
# X86 版プログラムを実行
./build_x86/systemV_sem_demo

# 実行結果は以下の通りです
Child process will wait for some seconds...
```

株式会社日昇テクノロジー

子プロセスが一定時間待ってからセマフォを解放

The returned value is 0 in the child process(PID = 16085)

親プロセスが子プロセスによるセマフォの解放後に実行されます

The returned value is 16085 in the father process(PID = 16084)

第 39 章 共有メモリ

39.1 共有メモリの基本概念

共有メモリとは何か？名前の通り、メモリを共有することを指します。これにより、関連しない複数のプロセスが同一の論理メモリにアクセスできるようになります。直接的に一塊の生のメモリをデータ転送が必要なプロセスの前に置き、それらが自由に使用できるようにします。そのため、共有メモリは IPC 通信メカニズムの中で最も効率的な一つであり、プロセス間でデータを共有および転送することができます。プロセス間で共有する必要のあるデータは共有メモリ領域に置かれ、この共有領域にアクセスする必要があるすべてのプロセスは、その共有領域を自プロセスのアドレス空間にマッピングする必要があります。そのため、すべてのプロセスが共有メモリ内のアドレスにアクセスできるようになります。これはまるで、C 言語の関数 malloc によって割り当てられたメモリであるかのようにです。

しかし、このような共有されたメモリはプロセス自身で管理する必要があります。例えば、同期、排他などの作業です。プロセス 1 が共有メモリのデータを読み取っている時に、プロセス 2 が共有メモリ内のデータを変更した場合、データの混乱が生じ、プロセス 1 が読み取るデータは誤りとなります。したがって、共有メモリはクリティカルリソースに属し、ある瞬間には最大で 1 つのプロセスのみが操作（読み/書き）できます。共有メモリは通常、単独で使用されることはなく、セマフォ、ミューテックスなどの調整メカニズムと組み合わせて使用され、プロセス間で効率的にデータを交換しつつ、データの踏みつけや破壊などの事故が発生しないようにします。

共有メモリの概念は非常にシンプルです。プロセス間で仮想メモリ空間は本来互いに独立しており、相互

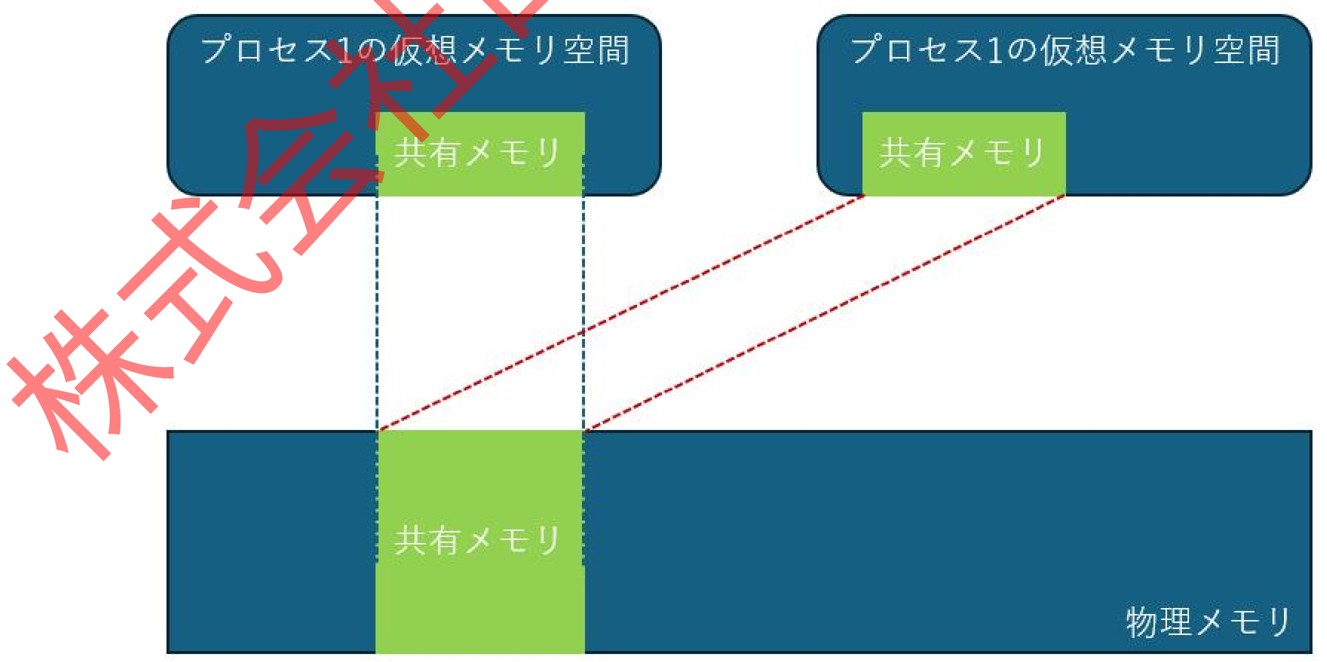
株式会社日昇テクノロジー

にアクセスすることはできませんが、ある方法を通じて、同じ物理メモリを異なるプロセスの仮想空間に複数回マッピングすることが可能になります。これにより、複数のプロセスの仮想メモリ空間が部分的に重なり合う効果が実現されます。

プロセス 1 が共有メモリにデータを書き込んだ後、共有メモリのデータが変更され、プロセス 2 が変更されたデータを即座に読み取ることができます。この間、カーネルのコピーを経由することなく、そのため非常に高い効率が得られます。

共有メモリは次の特徴を持っています：

1. 共有メモリはプロセス間通信の中で最も効率の良い方法の一つです。
2. 共有メモリは、複数のプロセス間の通信を考慮してシステムに予約されたメモリ領域であり、データ転送を目的としています。
3. 共有メモリは 2 つ以上のプロセスが同じメモリ領域にアクセスできるようにし、1 つのプロセスがこのアドレスの内容を変更すると、他のプロセスもその変更を察知します。
4. 共有メモリは同期や排他制御を必要としません。



株式会社日昇テクノロジー

共有メモリの利点と欠点は次のとおりです：

- 利点：共有メモリを使用してプロセス間通信を行うことは非常に便利で、関数のインターフェースもシンプルです。データの共有により、プロセス間のデータ転送が不要になり、プログラムの効率が向上します。また、匿名パイプのように通信するプロセスに一定の「血縁」関係が必要なく、システム内の任意のプロセスが共有メモリを読み書きできます。

- 欠点：共有メモリは同期のメカニズムを提供していないため、プロセス間通信に共有メモリを使用する際には、しばしば他の手段（セマフォ、ミューテックスなど）を使ってプロセス間の同期を行う必要があります。

39.2 shmget() 共有メモリ作成関数

カーネルは共有メモリオブジェクトを作成または取得し、共有メモリ識別子を返す shmget()関数を提供します。関数のプロトタイプは以下の通りです：

```
int shmget(key_t key, size_t size, int shmflg);
```

引数の説明：

- **key**：共有メモリを識別するキー値で、以下の値を取ることができます：

- 0 または IPC_PRIVATE。key の値が IPC_PRIVATE の場合、shmget()関数は新しい共有メモリを作成します。key の値が 0 で、引数 shmflg に IPC_PRIVATE フラグが設定されている場合も、新しい共有メモリが作成されます。

- 0 より大きい 32 ビット整数：引数 shmflg によって操作が決まります。

- **size**：作成する共有メモリのサイズで、全てのメモリ割り当て操作はページ単位で行われるため、たとえ 1 バイトのメモリしか要求していなくても、メモリは完全に 1 ページ割り当てられます。

- **shmflg**：作成する共有メモリのモードフラグを示し、実際の使用時には IPC オブジェクトのアクセス権限 mode（例：0600）と「｜」演算を行って共有メモリのアクセス権限を決定します。

shmflg には複数の場合があります：

株式会社日昇テクノロジー

- IPC_CREAT：カーネル内に key と等しいキーワードの共有メモリが存在しない場合、新しい共有メモリを作成します。そのような共有メモリが存在する場合、その共有メモリの識別子を返します。

- IPC_EXCL：カーネル内に key と等しいキーワードの共有メモリが存在しない場合、新しい共有メモリを作成します。そのような共有メモリが存在する場合、エラーが報告されます。

- SHM_HUGETLB：「大ページ」を使用して共有メモリを割り当てます。「大ページ」とは、カーネルがプログラムの性能を向上させるために、メモリのページ管理を行う際に、デフォルトサイズ（4KB）よりも大きなページサイズを採用することを指します。Linux カーネルは、物理ページの基本単位として 2MB をサポートしています。

- SHM_NORESERVE：この共有メモリのためにスワップ領域に空間を予約しない。

- **返り値**：shmget()関数の返り値は共有メモリの ID です。

shmget()関数の呼び出しに失敗すると、以下のエラーコードが生成されます：

- EACCES：key で指定された共有メモリが既に存在し、呼び出しプロセスがそれにアクセスする権限がない。

- EEXIST：key で指定された共有メモリが既に存在し、shmflg に IPC_CREAT と IPC_EXCL フラグが同時に指定されている。

- EINVAL：共有メモリの作成時にパラメータ size が SHMMIN 未満または SHMMAX を超えている。

- ENFILE：システム全体のオープンファイルの総数の制限に達している。

- ENOENT：指定された key に対応する共有メモリが存在せず、IPC_CREAT が指定されていない。

- ENOMEM：メモリが不足しており、共有メモリにメモリを割り当てることができない。

39.3 shmat() マッピング関数

共有メモリの基本概念から容易に理解できるように、プロセスが共有メモリにアクセスしたい場合は、それをプロセスの仮想空間にマッピングし、その後アクセスする必要があります。そのためにシステムが提

株式会社日昇テクノロジー

供している shmat()関数は、共有メモリ領域オブジェクトを呼び出しプロセスのアドレス空間にマッピングします。関数のプロトタイプは以下の通りです：

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

引数の説明：

- **shmid**：共有メモリ ID で、通常は shmget()関数によって返されます。

- **shmaddr**：NULL でない場合、システムは shmaddr に基づいて適切なメモリ領域を選択します。NULL の場合、システムは自動的に適切な仮想メモリ空間アドレスを選択して共有メモリをマッピングします。

- **shmflg**：共有メモリを操作する方法：

- SHM_RDONLY：共有メモリを読み取り専用でマッピングします。

- SHM_REMAP：再マッピングします。この場合、shmaddr は NULL であってはなりません。

- NULLSHM：shmaddr より小さい最大のページアラインアドレスを自動的に選択します。

shmat()関数の呼び出しが成功すると、共有メモリの開始アドレスが返され、これによって共有メモリを操作できるようになります。共有メモリのマッピングには以下の点に注意が必要です：

- 共有メモリは読み取り専用または読み書き可能でマッピングできるだけで、書き込み専用でマッピングすることはできません。

- shmat()の第二引数 shmaddr は一般に NULL に設定され、システムに適切なアドレスの選択を任せます。しかし、実際に NULL でない場合、shmflg に SHM_RND が設定されている必要があります。この場合、システムは shmaddr よりも小さく、最大のページアラインアドレス（つまり SHMLBA の整数倍）を共有メモリ領域の開始アドレスとして選択します。SHM_RND が設定されていない場合、shmaddr は厳密なページアラインアドレスでなければなりません。

株式会社日昇テクノロジー

39.4 shmdt() 関数によるマッピング解除

shmdt()関数は shmat()関数の逆であり、プロセスと共有メモリ間のマッピングを解除するために使用されます。マッピングを解除した後、そのプロセスはその共有メモリにアクセスできなくなります。関数のプロトタイプは以下の通りです：

```
int shmdt(const void *shmaddr);
```

引数の説明：

- **shmaddr**：マッピングされた共有メモリの開始アドレス。

shmdt()関数の呼び出しが成功すると 0 を返し、エラーが発生した場合は-1 を返し、エラーの原因を errno に格納します。

shmdt()関数はシンプルですが、注目すべきポイントがあります。この関数は指定された共有メモリ領域を削除するのではなく、以前に shmat()関数でマッピングされた共有メモリを現在のプロセスから切り離すだけであり、共有メモリは物理メモリに存在し続けます。

39.5 shmctl() 関数による属性の取得または設定

カーネルは、共有メモリの関連属性を取得または設定するために shmctl()関数を提供しています。関数のプロトタイプは以下の通りです：

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

引数の説明：

- **shmid**：共有メモリの識別子。

- **cmd**：関数の機能制御コマンドで、次の値を取ることができます：

- IPC_STAT：属性情報を buf に格納します。

- IPC_SET：buf が指す内容に属性情報を設定します。

- IPC_RMID：その共有メモリを削除します。

株式会社日昇テクノロジー

- IPC_INFO：共有メモリのシステム制限値情報を取得します。

- SHM_INFO：共有メモリのリソース消費情報を取得します。

- SHM_STAT：IPC_STAT と同様の機能を持ちますが、shmid はカーネルがすべての SHM 情報を記録する配列のインデックスであるため、すべてのインデックスを反復処理することで、システム内のすべての SHM の関連情報を取得できます。

- SHM_LOCK：システムがその SHM をスワップ領域に交換することを禁止します。

- SHM_UNLOCK：システムがその SHM をスワップ領域に交換することを許可します。

- **buf**：共有メモリ属性情報構造体へのポインタで、設定または取得する情報はこの構造体を通じて行われます。shmid_ds 構造体は以下の通りです：

注意：SHM_LOCK オプションは、読み書き権限をロックするのではなく、SHM がスワップ領域との交換ができるかどうかをロックします。SHM がスワップ領域に交換された後、SHM_LOCK が設定されている場合、この SHM にアクセスするすべてのプロセスはページエラーに遭遇します。プロセスは IPC_STAT 後に取得された mode をチェックすることで、SHM_LOCKED 情報を検出できます。

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* 所有権と権限 */
    size_t shm_segsz; /* 共有メモリのサイズ（バイト） */
    time_t shm_atime; /* 最後にアタッチされた時間 */
    time_t shm_dtime; /* 最後にデタッチされた時間 */
    time_t shm_ctime; /* 最後に状態が変更された時間 */
    pid_t shm_cpid; /* 作成者の PID */
    pid_t shm_lpid; /* 最後にアタッチまたはデタッチしたプロセスの PID */
    shmatt_t shm_nattch; /* この共有メモリをアタッチしているプロセスの数 */
    ...
```

株式会社日昇テクノロジー

};

権限情報構造体は以下の通りです：

```
struct ipc_perm {
    key_t __key; /* その共有メモリのキー値 */
    uid_t uid; /* 所有者の有効 UID */
    gid_t gid; /* 所有者の有効 GID */
    uid_t cuid; /* 作成者の有効 UID */
    gid_t cgid; /* 作成者の有効 GID */
    unsigned short mode; /* 読み書き権限 + SHM_DEST + SHM_LOCKED フラグ */
    unsigned short __seq; /* シーケンス番号 */
};
```

39.6 使用例

共有メモリを使用する一般的な手順は以下の通りです：

1. 共有メモリ ID を作成または取得します。
2. 共有メモリをプロセスの仮想メモリ空間の特定の領域にマッピングします。
3. 使用しなくなったら、マッピング関係を解除します。
4. その共有メモリを必要とするプロセスがなくなったら、それを削除します。

共有メモリはその特性上、プロセス内の他のメモリセグメントと使用習慣が異なります。一般的にプロセスはスタック領域の割り当てを自動的に回収できますが、ヒープ領域は malloc で申請し、free で回収します。これらのメモリは回収後には存在しないとみなせますが、共有メモリは異なります。shmdt()関数でマッピングを解除した後も、実際にはその使用中のメモリは存在し続け、shmat で再びマッピングして使用することができます。shmctl()関数でこの共有メモリを削除しない限り、それはシステムがシャットダウ

株式会社日昇テクノロジー

ンされるまで永続的に保持されます。さらに、データの競合を避けるために、共有メモリを信号量と組み合わせて使用するべきです。

共有メモリはその特性により、プロセス内の他のメモリセグメントとは使用上の慣習が異なります。通常、プロセスはスタック領域を自動的に回収し、ヒープ領域は malloc で割り当てられ、free で回収されます。これらのメモリは回収後に存在しないとみなされます。しかし、共有メモリは異なり、shmdt()関数でマッピングを解除した後も、そのメモリは依然として存在し、shmat を使用して再びマッピングすることができます。shmctl()関数を使用してこの共有メモリを削除しない限り、システムがシャットダウンされるまでずっと保持されます。さらに、データの競合を避けるために、共有メモリを使用する際はセマフォと組み合わせるべきです。

実験の概念は次の通りです。まず、クリティカルセクションを制御するための System V セマフォを作成し、次に共有メモリ書き込みプロセスと共有メモリ読み取りプロセスの 2 つのプロセスを実装します。書き込みプロセスではデータを書き込み、読み取りプロセスではデータを読み取って表示します。

39.6.1 共有メモリ書き込みプロセス

リスト 1: 共有メモリ書き込みプロセス（コードリポジトリ /system_programing/shm_write/sources/shm_write.c ファイル）

```
1 #include <sys/types.h>

2 #include <sys/shm.h>

3 #include <sys/sem.h>

4 #include <sys/ipc.h>

5 #include <unistd.h>

6 #include <stdlib.h>

7 #include <stdio.h>
```


株式会社日昇テクノロジー

```
8  #include <string.h>

9  #include <sys/stat.h>

10 #include <fcntl.h>

11 #include <errno.h>

12

13 #include "sem.h"

14

15

16 int main()

17 {

18     int running = 1;

19     void *shm = NULL;

20     struct shared_use_st *shared = NULL;

21     char buffer[BUFSIZ + 1]; // 入力されたテキストを保存するためのバッファ

22     int shmid;

23     int semid; // セマフォの識別子

24

25     // 共有メモリの作成

26     shmid = shmget((key_t)1234, 4096, 0644 | IPC_CREAT);

27     if(shmid == -1)

28     {

29         fprintf(stderr, "shmget failed¥n");

30         exit(EXIT_FAILURE);
```

株式会社日昇テクノロジー

```
31	}
32	// 共有メモリを現在のプロセスのアドレス空間に接続
33	shm = shmat(shmid, (void*)0, 0);
34	if(shm == (void*)-1)
35	{
36	    fprintf(stderr, "shmat failed¥n");
37	    exit(EXIT_FAILURE);
38	}
39	printf("Memory attached at %p¥n", shm);
40	
41	/** セマフォを開く、存在しない場合は作成 */
```

```
42	semid = semget((key_t)6666, 1, 0666|IPC_CREAT);
43	
44	if(semid == -1)
45	{
46	    printf("sem open fail¥n");
47	    exit(EXIT_FAILURE);
48	}
49	
50	
51	
52	while(running) // 共有メモリにデータを書き込む
```

株式会社日昇テクノロジー

```
53  {
54      // 共有メモリにデータを書き込む
55      printf("Enter some text: ");
56      fgets(buffer, BUFSIZ, stdin);
57      strncpy(shm, buffer, 4096);
58
59      sem_v(semid); /* セマフォを解放 */
60
61      // "end"が入力されたらループを抜ける（プログラムを終了）
62      if(strncmp(buffer, "end", 3) == 0)
63          running = 0;
```

```
64  }
65
66  // 共有メモリを現在のプロセスから切り離す
67  if(shmdt(shm) == -1)
68  {
69      fprintf(stderr, "shmdt failed¥n");
70      exit(EXIT_FAILURE);
71  }
72  sleep(2);
73  exit(EXIT_SUCCESS);
74 }
```

株式会社日昇テクノロジー

コードの説明は以下の通りです：

- 26 行目で、shmget()を呼び出して、4096 バイトの共有メモリを作成または取得します。

- 33 行目で、shmat()関数を呼び出して共有メモリを現在のプロセスにマッピングし、そのアドレスを shm ポインタに保存します。

- 56 行目で、strncpy 関数を使用してユーザーからの入力を共有メモリ shm にコピーします。

共有メモリに書き込んだ後、セマフォを解放する操作で他のプロセスがリソースを取得できることを通知します。これは共有メモリのクリティカルセクションを保護する一般的な方法です。

39.6.2 共有メモリ読み取りプロセス

リスト 2: 共有メモリ読み取りプロセス（コードリポジトリ /system_programing/shm_read/sources/shm_read.c ファイル）

```
1 #include <sys/types.h>
2 #include <sys/shm.h>
3 #include <sys/sem.h>
4 #include <sys/ipc.h>
5 #include <unistd.h>
6 #include <stdlib.h>
7 #include <stdio.h>
8 #include <string.h>
9 #include <sys/stat.h>
```

株式会社日昇テクノロジー

```
10 #include <fcntl.h>

11 #include <errno.h>

12

13 #include "sem.h"

14

15 int main(void)

16 {

17     int running = 1; // プログラムが続行するかどうかのフラグ

18     char *shm = NULL; // 割り当てられた共有メモリの最初のアドレス

19     int shmid; // 共有メモリ識別子

20     int semid; // セマフォ識別子

21

22     // 共有メモリの作成

23     shmid = shmget((key_t)1234, 4096, 0666 | IPC_CREAT);

24     if(shmid == -1)

25     {

26         fprintf(stderr, "shmget failed¥n");

27         exit(EXIT_FAILURE);

28     }

29

30     // 共有メモリを現在のプロセスのアドレス空間に接続

31     shm = shmat(shmid, 0, 0);

32     if(shm == (void*)-1)
```

株式会社日昇テクノロジー

```
33  {
34      fprintf(stderr, "shmat failed¥n");
35      exit(EXIT_FAILURE);
36  }
37  printf("¥nMemory attached at %p¥n", shm);
38
39  /** セマフォを開く、存在しない場合は作成 */
40  semid = semget((key_t)6666, 1, 0666|IPC_CREAT); // セマフォの作成
41
42  if(semid == -1)
43  {
44      printf("sem open fail¥n");
45      exit(EXIT_FAILURE);
46  }
47
48  init_sem(semid, 0);
49
50  while(running) // 共有メモリからデータを読み込む
51  {
52      /** セマフォを待つ */
53      if(sem_p(semid) == 0)
54      {
```

株式会社日昇テクノロジー

```
55          printf("You wrote: %s", shm);
56          sleep(rand() % 3);
57
58          // 'end'が入力された場合、ループ（プログラム）を終了
59          if(strncmp(shm, "end", 3) == 0)
60              running = 0;
61      }
62  }
63
```

```
64  del_sem(semid); // セマフォの削除
65
66  // 共有メモリを現在のプロセスから切り離す
67  if(shmdt(shm) == -1)
68  {
69      fprintf(stderr, "shmdt failed¥n");
70      exit(EXIT_FAILURE);
71  }
72
73  // 共有メモリの削除
74  if(shmctl(shmid, IPC_RMID, 0) == -1)
75  {
76      fprintf(stderr, "shmctl(IPC_RMID) failed¥n");
```

株式会社日昇テクノロジー

```
77        exit(EXIT_FAILURE);
78     }
79     exit(EXIT_SUCCESS);
80 }
```

コードの説明は以下の通りです：

- 23 行目で、shmget()を呼び出して、4096 バイトの共有メモリを作成または取得します。

- 31 行目で、shmat()関数を呼び出して共有メモリを現在のプロセスにマッピングし、そのアドレスを shm ポインタに保存します。

- 53～61 行目で、sem_p でセマフォを待機し、セマフォを取得した後、printf 関数を使用して共有メモリ shm の内容を表示します。

39.6.3 実験操作

このサンプルコードは、system_programing/shm_write および shm_read ディレクトリにあり、それぞれをコンパイルして実行すると次のようになります：

書き込みプロセス：任意の情報を入力でき、"end"を入力すると終了し、その時点で共有メモリが削除されます。

```
# 以下の操作は system_programing/shm_write コードディレクトリで実行します
# X86 版プログラムのコンパイル
make
# X86 版プログラムの実行
./build_x86/shm_write_demo
# 以下は実行時の出力です。任意の内容を入力し、end で終了します
Memory attached at 0x7fde8c9a3000
```

株式会社日昇テクノロジー

Enter some text: embedfire

Enter some text: test

Enter some text: hello world

Enter some text: end

読み取りプロセス：

新しいターミナルを開き、shm_read ディレクトリに切り替えてコンパイルして実行します：

```
# 以下の操作は system_programing/shm_read コードディレクトリで実行します

# X86 版プログラムのコンパイル

make

# X86 版プログラムの実行
```

```
./build_x86/shm_read_demo

# 以下は実行時の出力です。任意の内容を入力し、end で終了します

Memory attached at 0x7fa254d44000

You wrote: embedfire

You wrote: test

You wrote: hello world

You wrote: end
```

小技：この例では、プロセスが"end"文字で終了せずに（Ctrl+C や Ctrl+D などで）終了した場合、読み取りプロセスが共有メモリをアクティブに削除することはありません。この場合、ipcs -m コマンドを使用して共有メモリがまだ存在していることを確認し、ipcrm -m [共有メモリ shmid]で削除できます。

株式会社日昇テクノロジー

第 40 章 スレッド

40.1 スレッドとプロセス

多くの Linux の書籍では、プロセス（process）とスレッド（thread）について、プロセスはリソース管理の最小単位であり、スレッドはプログラム実行の最小単位であると説明されています。

この説明は非常に簡潔で、オペレーティングシステムの設計上、プロセスからスレッドが進化しました。スレッドの主な目的は、プロセスのコンテキストスイッチのオーバーヘッドを減らすことにありますが、これはどのようなことでしょうか？前述の通り、プロセスはリソース管理の最小単位であり、各プロセスは独自のデータセグメント、コードセグメント、スタックセグメントを持っています。これは、プロセス間のスイッチングが比較的複雑なコンテキストスイッチングを必要とすることを意味します。頻繁にプロセスを切り替える場合、このようなオーバーヘッドは非常に大きくなります。プロセスのコンテキストを切り替える際には、仮想アドレス空間の再マッピング、OS カーネルの入出力、レジスタの切り替えが必要であり、プロセッサのキャッシュメカニズムにも影響を与えます。したがって、プロセス切り替え時の CPU の追加オーバーヘッドをさらに削減するために、Linux では別の概念であるスレッドが進化しました。

スレッドは、オペレーティングシステムがスケジュールして実行できる基本単位であり、Linux では軽量プロセスとも呼ばれます。Linux システムでは、プロセスは少なくとも 1 つのスレッドを指令実行体として必要とし、プロセスはリソース（CPU、メモリ、ファイルなど）を管理し、スレッドを特定の CPU 上で実行させます。1 つのプロセスは複数のスレッドを持つことができ、複数の CPU を使用して各スレッドを同時に実行させることで、最大限の並列性を実現し、作業の効率を高めることができます。また、シングル CPU のマシン上でも、マルチスレッドモデルを使用してプログラムを設計することで、デザインがよりシンプルになり、機能が完全になり、プログラムの実行効率が向上します。

b これらの概念から、スレッドの本質はプロセス内の制御シーケンスであり、プロセスの内部に存在するものであることが分かります。1 つのプロセスは 1 つまたは複数のスレッドを持つことができます。プロ

株式会社日昇テクノロジー

セスが fork()関数を実行して新しいプロセスを作成するとき、そのプロセスのコピーが作成されます。この新しいプロセスは独自の変数と PID を持ち、親プロセスとほぼ完全に独立して実行されます。新しいプロセスを得るコストは非常に大きいです。一方、プロセス内で新しいスレッドを作成すると、新しい実行スレッドは独自のスタックを持ちますが、グローバル変数、ファイルディスクリプタ、シグナル処理関数、現在のディレクトリ状態を作成者と共有します。つまり、現在のプロセスのリソースを使用するだけで、現在のプロセスのコピーを生成するわけではありません。

Linux システム内の各プロセスは独立したアドレス空間を持ち、一つのプロセスがクラッシュしても、システムの保護モード下で他のプロセスに影響を与えることはありません。しかし、スレッドはプロセス内部の制御シーケンスであるため、プロセスがクラッシュするとスレッドも同様にクラッシュします。そのため、マルチプロセスプログラムはマルチスレッドプログラムよりも堅牢ですが、プロセスの切り替えにはリソースが多く消費され、効率が低下します。これにより、同時に実行され、特定の変数を共有する必要がある並行操作には、プロセスではなくスレッドを使用する必要があります。

要約すると：

- プログラムには少なくとも 1 つのプロセスが必要で、プロセスには少なくとも 1 つのスレッドが必要です。
- スレッドが使用するリソースはプロセスのリソースであり、プロセスがクラッシュするとスレッドもクラッシュします。
- スレッドのコンテキストスイッチはプロセスよりも速く、本質的にスレッドは多くのリソースをプロセスと共有するため、切り替える際に保存および切り替える項目が比較的少なくなります。

40.2 スレッドの作成

スレッドプログラミングを説明する前に、POSIX（Portable Operating System Interface）という知識点を理解する必要があります。POSIX は、IEEE が UNIX オペレーティングシステム上でソフトウェアを実

株式会社日昇テクノロジー

行するために定義された API インターフェースなどの一連の標準の総称で、公式名称は IEEE Std 1003 であり、国際標準名は ISO/IEC 9945 です。この標準は、1985 年頃に始まったプロジェクトから発生しました。POSIX という名前は、IEEE の要請に応えてリチャード・ストールマン（RMS）が提案した記憶に残りやすい名前です。基本的には、Portable Operating System Interface（移植可能なオペレーティングシステムインターフェース）の略であり、X は Unix API の遺産を示しています。

簡単に言えば、アプリケーションが POSIX 標準のインターフェースを使用してシステム関数を呼び出す場合、そのアプリケーションは POSIX 標準に準拠しているシステム上で非常に容易に移植されるか、直接互換性を持つことになります。

Linux システム下のマルチスレッドは POSIX 標準に従っており、その中の一般的なスレッドライブラリが pthread です。これは POSIX によって提案された一般的なスレッドライブラリであり、優れた移植性を持っています。学ぶ Linux マルチスレッドプログラミングも正にそれを使用しています。使用時には以下のヘッダーファイルを含める必要があります：

```
#include <pthread.h>
```

さらに、リンク時にはライブラリ libpthread.a を使用する必要があります。pthread のライブラリは Linux システムのライブラリではないので、コンパイル時に`-lpthread`オプションを付けて明示的にリンクする必要があります。

40.2.1 pthread_create()によるスレッドの作成

pthread_create()関数はスレッドを作成するために使用され、スレッドを作成することは、実際にはそのスレッド関数のエントリポイントを決定することです。スレッドが作成されると、関連するスレッド関数の実行が開始されます。関数プロトタイプは以下の通りです：

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
            void *(*start_routine) (void *), void *arg);
```

株式会社日昇テクノロジー

引数説明：

- thread：スレッド識別子へのポインタ。

- attr：スレッド属性の設定。具体的な内容は次の節で説明します。

- start_routine：`start_routine`は関数ポインタで、スレッドのエントリポイント、つまりスレッドが実行する関数のコードを指します。

- arg：スレッド実行時に渡される引数。

- 戻り値：スレッドの作成に成功した場合は 0 を返します。失敗した場合は、対応するエラーコードを返します。

40.3 スレッド属性

上記の pthread_create では、スレッド属性を入力パラメータとして必要とします。Linux でのスレッド属性構造は以下のようになります：

```
typedef struct
{
   int detachstate; // スレッドの分離状態
   int schedpolicy; // スレッドスケジューリングポリシー
   struct sched_param schedparam; // スレッドのスケジューリングパラメータ
   int inheritsched; // スレッドの継承性
   int scope; // スレッドのスコープ
   size_t guardsize; // スレッドスタックの末尾のガードバッファサイズ
   int stackaddr_set; // スレッドのスタック設定
   void* stackaddr; // スレッドスタックの位置
```

株式会社日昇テクノロジー

size_t stacksize; // スレッドスタックのサイズ

} pthread_attr_t;

注意：pthread は Linux システムのデフォルトライブラリではなく、POSIX スレッドライブラリです。Linux でライブラリとして使用する場合、コンパイル時に-lpthread（または-pthread）を付けて、明示的にこのライブラリをリンクする必要があります。関数がエラーを実行した場合のエラーメッセージは戻り値として返され、システムのグローバル変数 errno は変更されないため、perror()を使用してエラーメッセージを出力することはできません。

スレッドの属性は多岐にわたり、その属性値は直接設定できないため、関連する関数を使用して操作する必要があります。主なスレッド属性には、スコープ（scope）、スタックサイズ（stacksize）、スタックアドレス（stackaddress）、優先度（priority）、分離状態（detachedstate）、スケジューリングポリシーとパラメータ（scheduling policy and parameters）などがあります。デフォルトの属性は非バインド、非分離、スタックサイズ 1M、親プロセスと同じ優先度です。スレッド属性に関連する API インターフェースについては、以下のように説明されます：

API	説明
pthread_attr_init()	スレッドオブジェクトの属性を初期化します
pthread_attr_destroy()	スレッド属性オブジェクトを破棄します
pthread_attr_getaffinity_np()	スレッド間の CPU 親和性を取得します
pthread_attr_setaffinity_np()	スレッドの CPU 親和性を設定します
pthread_attr_getdetachstate()	スレッドの分離状態属性を取得します
pthread_attr_setdetachstate()	スレッドの分離状態属性を変更します
pthread_attr_getguardsize()	スレッドのスタック保護領域のサイズを取得します
pthread_attr_setguardsize()	スレッドのスタック保護領域のサイズを設定します
pthread_attr_getscope()	スレッドのスコープを取得します
pthread_attr_setscope()	スレッドのスコープを設定します
pthread_attr_getstack()	スレッドのスタック情報（スタックアドレスとスタックサイズ）を取得します
pthread_attr_setstack()	スレッドスタック領域を設定します
pthread_attr_getstacksize()	スレッドのスタックサイズを取得します

株式会社日昇テクノロジー

pthread_attr_setstacksize()	スレッドのスタックサイズを設定します
pthread_attr_getschedpolicy()	スレッドのスケジューリングポリシーを取得します
pthread_attr_setschedpolicy()	スレッドのスケジューリングポリシーを設定します
pthread_attr_getschedparam()	スレッドのスケジューリング優先度を取得します
pthread_attr_setschedparam()	スレッドのスケジューリング優先度を設定します
pthread_attr_getinheritsched()	スレッドがスケジューリング属性を継承するかどうかを取得します
pthread_attr_setinheritsched()	スレッドがスケジューリング属性を継承するかどうかを設定します

特に他の要件がない場合は、スレッド関連の属性を考慮する必要はなく、デフォルトの属性を使用できます。

40.3.1 スレッドオブジェクト属性の初期化

スレッドオブジェクトの属性を初期化するには、pthread_attr_init()関数を使用します。関数プロトタイプは以下の通りです：

```
int pthread_attr_init(pthread_attr_t *attr);
```

- attr：スレッド属性へのポインタ
- 戻り値：関数呼び出しが成功すると 0 が返され、それ以外の場合は対応するエラーコードが返されます。

40.3.2 スレッド属性オブジェクトの破棄

pthread_attr_destroy() 関数はスレッド属性オブジェクトを破棄するために使用されます。もし pthread_create() 関数が破棄されたスレッド属性オブジェクトを使用してスレッドを作成しようとすると、エラーが返されます。

pthread_attr_destroy() 関数のプロトタイプ：

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

- attr：スレッド属性へのポインタ

株式会社日昇テクノロジー

- 戻り値：関数呼び出しが成功すると 0 が返され、それ以外の場合は対応するエラーコードが返されます。

40.3.3 スレッドの分離状態

スレッド属性の中には分離状態がありますが、スレッドの分離状態とは何でしょうか？ある時点で、スレッドは結合可能（joinable）または分離（detached）のいずれかの状態にあります。結合可能なスレッドは他のスレッドによってそのリソースを回収され、終了させることができますが、他のスレッドによって回収されるまで、そのメモリリソース（例えばスタック）は解放されません。反対に、分離されたスレッドは他のスレッドによって回収または終了させることができず、そのメモリリソースは終了時にシステムによって自動的に解放されます。

要するに、スレッドの分離状態はスレッドがどのように自身を終了させるかを決定します。

プロセス内のスレッドは pthread_join() 関数を呼び出して、特定のスレッドの終了を待ち、そのスレッドの終了状態を取得し、スレッドが占有していたリソースを回収することができます。スレッドの戻り値に興味がない場合は、rval_ptr を NULL に設定できます。

```
int pthread_join(pthread_t tid, void **rval_ptr);
```

さらに、スレッドは pthread_detach()関数を呼び出して自身を分離状態に設定することができます。分離状態に設定されたスレッドは、スレッドが終了した時にオペレーティングシステムが自動的に占有していたリソースを回収します。分離状態に設定されたスレッドは pthread_join() を呼び出して終了を待つことはできません。

```
int pthread_detach(pthread_t tid);
```

スレッドが結合可能である場合、つまりスレッドが終了しても自動的にリソースが解放されず、ゾンビスレッドとなり、そのスレッドの終了値を他のスレッドが取得できることを意味します。したがって、特定のスレッドの終了値に興味がない場合は、スレッドを分離状態に設定してゾンビスレッドになるのを防ぐ

株式会社日昇テクノロジー

べきです。

スレッドを作成する際にスレッドの終了状態に興味がないことが分かっている場合は、pthread_attr_t 構造体の detachstate 属性を変更して、スレッドを分離状態で開始することができます。

pthread_attr_setdetachstate() 関数のプロトタイプは以下の通りです：

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

特定のスレッドの分離状態を取得するには、`pthread_attr_getdetachstate()` 関数を使用します：

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
```

関数呼び出しが成功すると 0 が返され、それ以外の場合は対応するエラーコードが返されます。

- 引数説明：

  - attr：スレッド属性へのポインタ。

  - detachstate：値が `PTHREAD_CREATE_DETACHED` であればスレッドは分離状態、PTHREAD_CREATE_JOINABLE であれば結合状態を意味します。

40.3.4 スレッドのスケジューリングポリシー

スレッド属性にはスケジューリングポリシーの設定が含まれています。POSIX 標準では、以下の 3 つのスケジューリングポリシーが指定されています：

- タイムシェアリングスケジューリングポリシー：`SCHED_OTHER`。これはスレッド属性のデフォルト値です。他の 2 つのスケジューリング方法は、リアルタイムスケジューリング機能を持っているため、スーパーユーザー権限で実行されるプロセスでのみ使用できますが、動作上わずかに異なります。

- リアルタイムスケジューリングポリシー：先入れ先出し（FIFO）方式 `SCHED_FIFO`。キューベースのスケジューラで、各優先順位に異なるキューが使用され、キューに最初に入ったスレッドが優先

株式会社日昇テクノロジー

的に実行されます。スレッドは、より高い優先順位のタスクが到着するか、自ら CPU の使用権を放棄するまで、CPU を占有し続けます。

- リアルタイムスケジューリングポリシー：ラウンドロビン（RR）方式`SCHED_RR`。FIFO に似ていますが、各スレッドに実行時間のクォータが与えられ、`SCHED_RR`ポリシーを使用するスレッドの時間切れが発生すると、システムは再度時間を割り当て、そのスレッドを準備キューの最後に置き、スレッドを切り替えます。キューの最後に置くことで、同じ優先度を持つすべての RR スレッドのスケジューリングが公平になることが保証されます。

スケジューリングに関連する API インターフェースは以下の通りです：

```
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inheritsched);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
```

関数呼び出しが成功すると 0 が返され、それ以外の場合は対応するエラーコードが返されます。

- 引数説明：

- attr：スレッド属性へのポインタ。

- inheritsched：スレッドがスケジューリング属性を継承するかどうか。

PTHREAD_INHERIT_SCHED はスケジューリング属性が作成されたスレッドから継承され、attr で設定されたスケジューリング属性は無視されます。PTHREAD_EXPLICIT_SCHED はスケジューリング属性が attr で指定された属性値に設定されます。

- policy：スレッドの 3 つのスケジューリングポリシー、SCHED_OTHER、SCHED_FIFO、SCHED_RR のいずれか。

株式会社日昇テクノロジー

40.3.5 スレッドの優先順位

言うまでもなく、スレッドの優先順位はそのスレッドが実行される優先度を意味します。Linux システムでは、優先順位の数値が小さいほど、スレッドの優先度は高くなります。Linux はスレッドの優先順位に従ってスレッドをスケジュールし、スレッド属性で指定されたスケジューリング戦略に従います。

スレッドの静的優先順位（staticpriority）を取得・設定するには以下の関数を使用しますが、これは静的優先順位であり、スレッドのスケジューリングポリシーが SCHED_OTHER の場合、その静的優先順位は 0 に設定されなければなりません。このスケジューリングポリシーは Linux システムのデフォルトの戦略であり、0 優先度のスレッドは動的優先順位によってスケジュールされます。この「動的」という言葉は、スレッドの実行に伴い、スレッドのパフォーマンスに基づいて変化することを意味し、動的優先順位はスレッドの nice 値から始まり、スレッドが準備状態でありながらスケジューラーによって無視された場合、その動的優先順位は自動的に一つ増加します。これにより、これらのスレッドが CPU を公平に競合することを保証します。

```
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);
```

- 引数説明：
  - attr：スレッド属性へのポインタ。
  - param：静的優先順位の値。

スレッド優先順位の特徴：

- 新しいスレッドのデフォルト優先順位は 0 です。
- 新しいスレッドは親スレッドのスケジューリング優先順位を継承しません（PTHREAD_EXPLICIT_SCHED）。
- スレッドのスケジューリングポリシーが SCHED_OTHER の場合、スレッド優先順位の変更は許可

株式会社日昇テクノロジー

されません。リアルタイムスケジューリングポリシー（SCHED_RR または SCHED_FIFO）を使用する場合のみ有効であり、実行時に pthread_setschedparam()関数を通じて変更できます。デフォルトは 0 です。

40.3.6 スレッドスタック

スレッドスタックは非常に重要なリソースであり、関数の形式パラメータ、ローカル変数、スレッド切り替え時のレジスタなどのデータを格納することができます。前の文で述べたように、スレッドはプロセスのメモリ空間を使用します。したがって、1 つのプロセスに n 個のスレッドがあり、デフォルトのスレッドスタックサイズが 1M の場合、プロセスのメモリ空間が不足する可能性があります。そのため、複数のスレッドがある場合、プロセスのメモリ空間が不足しないように、特定のスレッドスタックのサイズを適宜減らすことができます。一方、一部のスレッドは大量の作業を完了する必要があるかもしれません。または、スレッドが呼び出す関数が大きなローカル変数を割り当てたり、関数呼び出しのレベルが深い場合、必要なスタック空間が非常に大きくなる可能性があります。そのため、スレッドスタックのサイズを増やすこともできます。

スレッドスタックサイズの設定と取得には以下の関数を使用できます：

パラメータ説明：

- attr：スレッド属性へのポインタ。

- stacksize：スレッドスタックのサイズ。

40.4 スレッドの終了

スレッドの終了について、スレッドが作成された後、システムは関連するスレッド関数の実行を開始します。その関数が実行完了した後、スレッドは終了します。これはスレッドの暗黙的な終了方法であり、プロセスの終了と似ています。プロセスが作業を完了した後に終了します。もう一つのスレッド終了方法は、pthread_exit()関数を使用してスレッドを明示的に終了させることです。これはスレッドの能動的な行動で

株式会社日昇テクノロジー

す。スレッド関数を使用する際には、exit()関数を誤って使用してエラー処理をすることはできません。なぜなら、exit()関数は呼び出しプロセスを終了させるためのものであり、一つのプロセスには多数のスレッドが含まれているため、exit()を使用した後、そのプロセス内の全てのスレッドが終了するからです。したがって、スレッド内では pthread_exit()関数のみを呼び出すことができ、プロセス終了関数 exit()は呼び出すことができません。

関数原型：

```
void pthread_exit(void *retval);
```

パラメータ説明：

- retval：retval が NULL でない場合、スレッドの終了値を retval に保存します。スレッドの終了値に関心がなければ、引数に NULL を指定できます。

一般的に、プロセス内の各スレッドの実行は相互に独立しています。スレッドの終了は互いに通知されず、他のスレッドに影響を与えません。終了したスレッドが使用していたリソースは、スレッドの終了とともにシステムに返却されるわけではなく、引き続きスレッドが属していたプロセスが保持します。これは、プロセス内の複数のスレッドがデータセグメントを共有しているからです。以前の記事からわかるように、プロセス間で wait()システムコールを使用して他のプロセスの終了を待つように、スレッドにも同様の関数があります：

```
int pthread_join(pthread_t tid, void **rval_ptr)；
```

あるスレッドが別のスレッドの終了を待ち、その終了値を取得したい場合、pthread_join()関数を使用して、指定されたスレッドの終了をブロックする形で待つことができます。関数が戻るときに、待っていたスレッドのリソースは回収されます。プロセスが既に終了している場合、この関数は直ちに戻ります。そして、thread に指定されたスレッドは結合可能な状態でなければなりません。この関数は成功すると 0 を返し、そうでなければ対応するエラーコードを返します。

パラメータ説明：

株式会社日昇テクノロジー

- thread: スレッド識別子、つまりスレッド ID で、一意のスレッドを識別します。

- retval: 待機されたスレッドの戻り値を格納するためのユーザー定義のポインタ。

結合可能な状態のスレッドが使用していたメモリは、他のスレッドが pthread_join()を実行した後にのみ解放されるため、メモリリークを避けるために、全てのスレッドが終了する際には、DETACHED に設定されているか、pthread_join()を使用してリソースを回収する必要があります。

40.5 スレッドの実験

スレッドの実験について、日常使用する場合、特に必要がない限り、スレッドの属性を変更する必要はほとんどありません。ここでは、スレッドの実験を行います。実験では、プロセスを作成し、スレッドの属性はデフォルトの属性で、スレッドが完了した後に退出します。コードは以下のとおりです：

リスト 1: スレッド実験テスト

(base_code/system_programing/thread/sources/thread.c ファイル)

```
1 #include <unistd.h>

2 #include <fcntl.h>

3 #include <stdio.h>
```

```
4 #include <stdlib.h>

5 #include <pthread.h>

6

7 /* 実行するスレッド */

8 void *test_thread(void *arg)

9 {

10    int num = (unsigned long long)arg; /* sizeof(void*) == 8、sizeof(int) == 4 (64 ビットシステム) */

11
```

株式会社日昇テクノロジー

```
12    printf("これはテストスレッドです、引数は %d¥n", num);

13    sleep(5);

14    /* スレッドを終了する */

15    pthread_exit(NULL);

16 }

17

18

19 int main(void)

20 {

21    pthread_t thread;

22    void *thread_return;

23    int arg = 520;

24    int res;

25
```

```
26    printf("スレッドの作成を開始¥n");

27

28    /* スレッドを作成する、スレッドの機能は test_thread 関数 */

29    res = pthread_create(&thread, NULL, test_thread, (void*)(unsigned long long)(arg));

30    if(res != 0)

31    {

32       printf("スレッド作成失敗¥n");

33       exit(res);
```

株式会社日昇テクノロジー

```
34    }

35

36    printf("スレッドの作成に成功¥n");

37    printf("スレッドの終了を待っています...¥n");

38

39    /* スレッドの終了を待つ */

40    res = pthread_join(thread, &thread_return);

41    if(res != 0)

42    {

43        printf("スレッドの終了に失敗¥n");

44        exit(res);

45    }

46

47    printf("スレッドの終了に成功¥n");

48

49    return 0;

50 }
```

コードの分析は以下の通りです：

- 8 行目から 16 行目にかけて、スレッドが実行する関数として test_thread 関数を定義します。関数内の操作は、渡された arg パラメータを出力し、一定時間スリープした後、thread_exit を呼び出してスレッドを終了します。

- 29 行目で、pthread_create 関数を呼び出してスレッドを作成します。スレッド関数のポインタには

株式会社日昇テクノロジー

test_thread を指定し、関数パラメータ arg（520）を渡します。作成後、スレッドは test_thread のコードを実行し始めます。

- 40 行目で、スレッドが終了するのを待つために pthread_join を呼び出します。

この例では、Makefile に lpthread リンクライブラリを追加する必要があります：

リスト 2: lpthread リンクの追加

（base_code/system_programing/thread/sources/Makefile ファイル）

```
LINK = -lpthread
```

40.5.1 実験操作

system_programing/thread ディレクトリに移動して、make を実行してソースコードをコンパイルし、その後実行します。実験の現象は以下の通りです：

```
# 以下の操作は system_programing/thread コードディレクトリで行います

# X86 バージョンのプログラムをコンパイル

make

# X86 バージョンのプログラムを実行
```

```
./build_x86/thread_demo


# 実行の出力は以下の通り

start create thread

create threads success

waiting for threads to finish...

This is test thread, arg is 520

# しばらく待ってスレッドが終了
```

株式会社日昇テクノロジー

thread exit ok

実験の現象から、プロセスは pthread_join の場所で対応するスレッドが終了するのを待ってから、後続のコードを実行することがわかります。

この例には、スレッド属性の実験も含まれています。興味があれば、base_code/system_programing/thread_attr ディレクトリの内容を見て、学習することができます。

第 41 章 POSIX セマフォ

この章では、プロセス/スレッド間通信の別のメカニズムである POSIX セマフォについて説明します。System V セマフォとの違いを明確にするため、特に指定がない限り、この章でのセマフォは POSIX セマフォを指します。また、以前に説明したシグナルと混同しないでください。シグナルとセマフォは異なる二つのメカニズムです。

41.1 POSIX セマフォの基本概念

セマフォ（Semaphore）は、プロセス/スレッド間の通信を実現するメカニズムであり、プロセス/スレッド間の同期またはクリティカルリソースへの排他的アクセスを実現するためによく使用されます。多プロセス/スレッドシステムでは、各プロセス/スレッド間で同期または排他を実現し、クリティカルリソースの保護が必要です。セマフォ機能は、このようなサポートを提供します。

POSIX 標準では、セマフォは二種類に分かれます。一つは名前なしセマフォ、もう一つは名前付きセマフォです。名前なしセマフォは主にプロセス/スレッド間の同期または排他に使用され、名前付きセマフォは主にプロセス間の同期または排他に使用されます。名前なしセマフォと名前付きセマフォの違いは、作成と破棄の形式にありますが、他の動作は同じで、名前なしセマフォは直接メモリに保存され、名前付きセマフォはファイルを作成する必要があります。

その名の通り、名前なしセマフォには名前がなく、メモリ内にのみ存在します。これは、セマフォを使用するプロセス/スレッドが名前なしセマフォが存在するメモリ領域にアクセスできる必要があることを意味

株式会社日昇テクノロジー

し、したがって名前なしセマフォは同一プロセス内のスレッド間の同期または排他にのみ使用できます。対照的に、名前付きセマフォは名前でアクセスできるため、それらの名前を知っている任意のプロセスやプロセス/スレッドによって使用できます。単一プロセス内で POSIX セマフォを使用する場合、名前なしセマフォの方がシンプルです。複数のプロセス間で POSIX セマフォを使用する場合、名前付きセマフォの方がシンプルです。

マルチプロセス/スレッドオペレーティングシステム環境では、複数のプロセス/スレッドが同時に実行され、一部のプロセス/スレッド間にはある程度の関連が存在する場合があります。複数のプロセス/スレッドが同じタスクを完了するために協力し合うことがあり、これによりプロセス/スレッド間の同期関係が形成され、セマフォを使用して同期を実現できます。

また、異なるプロセス/スレッド間で限られたシステムリソース（ハードウェアまたはソフトウェアリソース）をめぐって競争状態になることがあり、これがプロセス/スレッド間の排他関係です。複数のプログラムが同時に共有リソースにアクセスすることで発生する一連の問題を防ぐために、トークンを生成して使用することによって、ある時点で一つの実行プロセス/スレッドのみがコードのクリティカルセクションにアクセスできるようにする方法が必要です。

クリティカルセクションとは、データ更新のコードが排他的に実行される必要があることを指します。そして、セマフォはこのようなアクセスメカニズムを提供でき、一つのクリティカルセクションに同時にアクセスするプロセス/スレッドを一つに制限することができるため、共有リソースへのプロセス/スレッドのアクセスを調整するために使用できます。プロセス/スレッド間の排他と同期の関係は、クリティカルリソースの存在に起因します。クリティカルリソースとは、同時に限られた数（通常は一つ）のプロセス/スレッドのみがアクセス（読み取り）または変更（書き込み）できるリソースであり、通常はハードウェアリソース（プロセッサ、メモリ、ストレージ、その他の周辺機器など）およびソフトウェアリソース（共有コードセグメント、共有構造体や変数など）を含みます。

抽象的に言えば、セマフォには非負の整数が存在し、それを取得するすべてのプロセス/スレッドはその

株式会社日昇テクノロジー

整数を一つ減らします（もちろんリソースを使用するためです）。その整数値がゼロになった場合、それを取得しようとするすべてのプロセス/スレッドはブロックされます。通常、セマフォのカウント値は有効なリソース数に対応し、残りの使用可能な排他リソース数を示します。その値の意味は二つの場合に分かれます：

- 0：使用可能なセマフォがなく、プロセス/スレッドはセマフォ値が 0 より大きくなるまでスリープ状態に入ります。
- 正の値：一つまたは複数の使用可能なセマフォがあり、プロセス/スレッドはそのリソースを使用できます。プロセス/スレッドはセマフォ値を 1 減らし、一つのリソース単位を使用したことを示します。

セマフォの操作は二つに分けられます：

- P 操作：使用可能なリソース（セマフォ値が 0 より大きい）がある場合は、一つのリソースを占有します（セマフォ値を一つ減らし、クリティカルセクションコードに入ります）。使用可能なリソース（セマフォ値が 0 に等しい）がない場合は、システムがそのプロセス/スレッドにリソースを割り当てるまでブロックされます（待機キューに入り、リソースがそのプロセス/スレッドに回ってくるまで待ちます）。これは、車を駐車場に入れる前に、セキュリティに駐車カードを申請するようなものです。P 操作はリソースの申請であり、申請が成功すると、リソース数（空き駐車スペース）が一つ減ります。申請が失敗すると、入口で待つか、または去るかのどちらかです。
- V 操作：そのセマフォの待機キューにプロセス/スレッドが待っている場合は、一つのブロックされたプロセス/スレッドを起こします。プロセス/スレッドが待っていない場合は、一つのリソースを解放します（セマフォ値を一つ増やします）。これは、駐車場から出るときと同じで、空き駐車スペースが一つ増えます。

例えば、二つのプロセス/スレッドがセマフォ sem を共有し、sem の使用可能なセマフォの数値が 1 である場合、一方のプロセス/スレッドが P（sem）操作を実行すると、セマフォを取得し、クリティカルセ

株式会社日昇テクノロジー

クションに入り、sem を 1 減らします。もう一方のプロセス/スレッドは、クリティカルセクションに入ることができず、P（sem）操作を試みたときに sem が 0 であるため、最初のプロセス/スレッドがクリティカルセクションを離れ、V（sem）操作を実行してセマフォを解放した後、二番目のプロセス/スレッドが実行を再開できます。

41.2 POSIX 名前付きセマフォ

Linux でセマフォを使用して同期を行う場合、ヘッダーファイル semaphore.h を含める必要があります。

名前付きセマフォは実際にはファイルであり、その名前は"sem.[セマフォ名]"のような文字列で構成されます。ファイル名の前に"sem."があり、これは特別なセマフォファイルを指します。作成成功後、システムはそれを`/dev/shm`パス下に配置します。異なるプロセス間で同じセマフォファイル名を約束すれば、対応する名前付きセマフォにアクセスし、セマフォを利用して同期または排他操作を行うことができます。名前付きセマフォはファイルであるため、プロセスが終了した後も自動的には消えず、手動で削除してリソースを解放する必要があります。

主要な関数：

```
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);

int sem_post(sem_t *sem);

int sem_close(sem_t *sem);

int sem_unlink(const char *name);
```

株式会社日昇テクノロジー

- sem_open()関数は、名前付きセマフォを開く/作成するために使用されます。引数の説明は以下の通りです：

- name：開くまたは作成するセマフォの名前。

- oflag：指定されたファイルが存在しない場合、O_CREATE または O_EXCL を指定して作成操作を行うことができます。0 を指定すると、後続の 2 つの引数は省略できます。それ以外の場合は、後続の 2 つの引数が必要です。

- mode：ファイルの読み書き権限を数字で指定します。セマフォが既に存在する場合、このパラメータは無視されます。

- value：セマフォの初期値。このパラメータは新しく作成された場合にのみ設定が必要で、セマフォが既に存在する場合は無視されます。

- 戻り値：sem_t 型のポインターが返され、これは作成/開かれたセマフォを指します。後続の関数は、このセマフォポインターを通じて対応するセマフォにアクセスします。

- sem_wait()関数はセマフォの待機（取得）を行います。セマフォの値が 0 より大きい場合、セマフォの値を 1 減らして直ちに返ります。セマフォの値が 0 の場合、プロセス/スレッドはブロックされます。これは P 操作に相当します。成功すると 0 が返され、失敗すると-1 が返されます。

- sem_trywait()関数もセマフォの待機を行いますが、指定されたセマフォのカウンタが 0 の場合、EAGAIN エラーを直ちに返してブロックせずに待機します。

- sem_post()関数はセマフォを解放し、セマフォの値を 1 増やします。これは V 操作に相当します。成功すると 0 が返され、失敗すると-1 が返されます。

- sem_close()関数はセマフォを閉じるために使用されます。これは現在のプロセス/スレッドがセマフォの使用を終了することを意味し、その作用は現在のプロセス/スレッドに限定されます。他のプロセス/スレッドは引き続きそのセマフォを使用できます。また、プロセスが終了する際（正常終了またはシグナルによる中断終了の場合）、カーネルは自動的にこの関数を呼び出してプロセスが使用してい

株式会社日昇テクノロジー

たセマフォを閉じます。この後、他のプロセス/スレッドがこのセマフォを使用しなくても、カーネルはセマフォを維持します。

- sem_unlink()関数は、指定された名前のセマフォファイルを直接削除してセマフォを削除するために使用されます。

41.3 POSIX 名前なしセマフォ

名前なしセマフォの操作は名前付きセマフォとほぼ同じですが、ファイルシステムの識別を使用せず、プログラム実行中のメモリ内に直接存在します。異なるプロセス間でアクセスすることはできず、異なるプロセス間での相互アクセスには使用できません。同様に、親プロセスがセマフォを初期化し、そのコピーを fork すると、得られるのはセマフォのコピーであり、これら 2 つのセマフォ間には関係がありません。

主要な関数：

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
```

- sem_init()：セマフォを初期化します。

- sem は初期化するセマフォです。既に初期化されているセマフォに対して`sem_init`操作を行うと、予測不可能な問題が発生する可能性があります。

- pshared は、このセマフォがプロセス間で共有されるか、スレッド間で共有されるかを示します。現在のところ Linux ではプロセス間で共有される名前なしセマフォは実装されていないため、この値は 0 にする必要があり、これはセマフォが現在のプロセスのローカルセマフォであることを意味します。

- value はセマフォの初期値です。

株式会社日昇テクノロジー

- 戻り値：成功すると 0 が返され、失敗すると-1 が返されます。

- sem_destroy()：セマフォを破棄します。`sem`は破棄するセマフォです。`sem_init`で初期化されたセマフォのみが`sem_destroy()`関数で破棄できます。成功すると 0 が返され、失敗すると-1 が返されます。

- sem_wait()、sem_trywait()、sem_post()などの関数は、名前付きセマフォと同様に使用されます。

41.4 POSIX セマフォの使用例

41.4.1 名前付きセマフォ

まず、名前付きセマフォのサンプルコードを分析しましょう：

リスト 1: POSIX 名前付きセマフォ

（base_code/system_programing/posix_sem1/sources/posix_sem.c ファイル）

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #include <stdlib.h>
6 #include <stdio.h>
7 #include <sys/types.h>
8 #include <fcntl.h>
9 #include <sys/wait.h>
10
11 int main(int argc, char **argv)
12 {
```

株式会社日昇テクノロジー

```
13    int pid;

14    sem_t *sem;

15    const char sem_name[] = "my_sem_test";

16

17    pid = fork();

18

19    if (pid < 0) {

20        printf("フォークエラー！¥n");

21    }

22    /* 子プロセス */

23    else if (pid == 0) {

24        /* 初期値が 1 のセマフォを作成/オープン */

25        sem = sem_open(sem_name, O_CREAT, 0644, 1);

26

27        if (sem == SEM_FAILED) {

28            printf("セマフォの作成に失敗しました...¥n");

29

30            sem_unlink(sem_name);

31

32            exit(-1);

33        }

34        /* セマフォを取得 */
```

株式会社日昇テクノロジー

```
35        sem_wait(sem);

36

37        for (int i = 0; i < 3; ++i) {

38

39            printf("子プロセス実行: %d¥n", i);

40            /* CPU 使用を解放するためにスリープ */

41            sleep(1);

42        }

43

44        /* セマフォを解放 */

45        sem_post(sem);

46

47    }

48    /* 親プロセス */

49    else {
```

```
50

51        /* 初期値が 1 のセマフォを作成/オープン */

52        sem = sem_open(sem_name, O_CREAT, 0644, 1);

53

54        if (sem == SEM_FAILED) {

55            printf("セマフォの作成に失敗しました...¥n");

56
```

株式会社日昇テクノロジー

```
57          sem_unlink(sem_name);

58

59          exit(-1);

60      }

61      /* セマフォを申請 */

62      sem_wait(sem);

63

64      for (int i = 0; i < 3; ++i) {

65

66          printf("親プロセス実行: %d¥n", i);

67          /* CPU 使用を解放するためにスリープ */

68          sleep(1);

69      }

70

71      /* セマフォを解放 */
```

```
72      sem_post(sem);

73      /* 子プロセスの終了を待つ */

74      wait(NULL);

75

76      /* セマフォを閉じる */

77      sem_close(sem);

78      /* セマフォを削除 */
```

株式会社日昇テクノロジー

```
79      sem_unlink(sem_name);
80    }
81
82    return 0;
83 }
```

このコード例の分析は以下の通りです。

- 23 行目から 47 行目にかけて、fork を使って子プロセスが作成された部分です。この部分は子プロセスのコードです。

- 25 行目で、sem_open()関数を使って信号量を開いたり、新しく作成したりしています。信号量の初期値は 1 です。

- 35 行目で、sem_wait()を呼び出して信号量を取得しようとします。もし信号量の値が 0 なら、コードはここでブロックされて待機します。

- 41 行目では、for ループ内で少し休むことによって、ループの中で一つずつ出力した後に CPU を一時的に解放します。これにより他のプロセスが実行できるようになります。

- 45 行目で、ループが完了した後、sem_post()を呼び出して信号量を解放します。

- 49 行目から 80 行目にかけて、この部分は親プロセスのコードです。

- 52 行目から 72 行目まで、それは子プロセスの内容と全く同じです。信号量を開いて取得した後、ループで出力し、その後信号量を解放します。

- 74 行目から 79 行目まで、親プロセスは子プロセスの終了を待ってから、sem_close()と sem_unlink()を呼び出して信号量を閉じて解放します。

このコードでは、2 つのプロセスの for ループ内に sleep()を意図的に挿入して CPU の解放を模倣しています。プロセス A が CPU を解放した後、通常の状況では他のプロセス B が CPU を得てコードを実行します。しかし、この例の親子プロセスの出力操作は同じセマフォを待つ必要があるため、プロセス A

株式会社日昇テクノロジー

がスリープしていてもまだ CPU を解放していないため、プロセス B はセマフォを得られずに実行されません。

したがって、以下のような結果を推測できます：

- セマフォの制御により、実行後の結果はプロセス A が連続して 0、1、2 の 3 つのステートメントを出力し、プロセス A がセマフォを解放した後、プロセス B が連続して 0、1、2 のステートメントを出力します。

- もし例のコードからすべてのセマフォに関連する操作をコメントアウトし（for ループ内の sleep はそのままに）、実行すると、sleep の存在により、実行後の結果はプロセス A が 0 を出力した後にスリープして CPU を解放し、プロセス B が 0 を出力した後にスリープして CPU を解放します。そして、プロセス A が 1 を出力し、プロセス B が 1 を出力するなど、これら 2 つのプロセスが交互に実行し、交互に出力します。

41.4.1.1 実験操作

この実験のコードは base_code/system_programing/posix_sem1 ディレクトリに保存されており、コンパイルと実行のプロセスは以下の通りです：

```
#base_code/system_programing/posix_sem1 コードディレクトリで以下の操作を行います。
#X86 バージョンのプログラムをコンパイルします。
make
#X86 バージョンのプログラムを実行します。
./build_x86/posix_sem1_demo
```


株式会社日昇テクノロジー

- 実行の出力は以下の通りです。

parent process run: 0

parent process run: 1

parent process run: 2

child process run: 0

child process run: 1

child process run: 2

見ての通り、2 つのプロセスはそれぞれ連続して出力しています。興味があれば、親プロセスと子プロセスのセマフォ待機操作 sem_wait をコメントアウトして、実験現象を観察することができます。

注意：fork()後に先に親プロセスが実行されるか子プロセスが実行されるかは不定ですが、連続出力か交互出力かを区別できれば、この例でのセマフォの役割がわかります。

コードの実行中に新しいターミナルを開き、以下のコマンドを入力すると：

```
ls -l /dev/shm
```

-rw-r--r-- 1 root root 32 2 月 14 13:31 sem.my_sem_test

-dev/shm ディレクトリ下に sem.my_sem_test ファイルが存在することがわかります。これは実験中に作成されたセマフォで、プロセスが完了するとこのセマフォは削除されます。sudo 権限で rm コマンドを呼び出してこのセマフォファイルを手動で削除することもできます。

41.4.2 無名セマフォの例：

以下は、3 つのスレッド間での順序実行を無名セマフォ同期メカニズムを使用して実現する例です：

リスト 2: POSIX 無名セマフォ

（base_code/system_programing/posix_sem/sources/posix_sem.c ファイル）

```
1 #include <unistd.h>
```


株式会社日昇テクノロジー

```
2  #include <stdio.h>

3  #include <stdlib.h>

4  #include <pthread.h>

5  #include <semaphore.h>

6

7  #define THREAD_NUMBER 3 /* スレッド数 */

8  #define REPEAT_NUMBER 4 /* 各スレッドの小タスク数 */

9

10 sem_t sem[THREAD_NUMBER];

11

12 /* スレッド関数 */

13 void *thread_func(void *arg)

14 {

15     int num = (unsigned long long)arg;

16     int delay_time = 0;

17     int count = 0;

18

19     /* セマフォを待つ、P 操作 */

20     sem_wait(&sem[num]);

21

22     printf("スレッド %d が開始¥n", num);

23     for (count = 0; count < REPEAT_NUMBER; count++)

24     {
```

株式会社日昇テクノロジー

```
25        printf("¥t  スレッド %d: ジョブ %d ¥n", num, count);

26        sleep(1);

27    }

28

29    printf("スレッド %d 終了¥n", num);

30    /* スレッドを終了 */

31    pthread_exit(NULL);

32 }

33

34

35 int main(void)

36 {

37    pthread_t thread[THREAD_NUMBER];

38    int i = 0, res;

39    void *thread_ret;

40


41    /* 3  つのスレッドと  3  つのセマフォを作成 */

42    for (i = 0; i < THREAD_NUMBER; i++)

43    {

44        /* セマフォを作成、初期セマフォ値は  0 */

45        sem_init(&sem[i], 0, 0);

46        /* スレッドを作成 */
```

株式会社日昇テクノロジー

```
47        res = pthread_create(&thread[i], NULL, thread_func, (void*)(unsigned long long)i);

48

49        if (res != 0)

50        {

51            printf("スレッド %d の作成失敗¥n", i);

52            exit(res);

53        }

54    }

55

56    printf("スレッド作成成功¥n スレッドの完了を待っています...¥n");

57

58    /* 順番にセマフォを解放、V 操作 */

59    for (i = 0; i < THREAD_NUMBER; i++)

60    {

61        /* V 操作を実施 */

62        sem_post(&sem[i]);

63        /* スレッドの実行完了を待つ */

64        res = pthread_join(thread[i], &thread_ret);

65        if (!res)

66        {

67            printf("スレッド %d 結合成功¥n", i);

68        }
```

株式会社日昇テクノロジー

```
69         else
70         {
71             printf("スレッド %d の結合失敗¥n", i);
72         }
73
74     }
75
76     for (i = 0; i < THREAD_NUMBER; i++)
77     {
78         /* セマフォを削除 */
79         sem_destroy(&sem[i]);
80     }
81
82     return 0;
83 }
```

株式会社日昇テクノロジー

このコードの説明は以下の通りで、main 関数から順を追って分析します：

- 44 行目と 45 行目では、for ループ内で 3 つのセマフォを sem 配列に格納して作成し、3 つのスレッドを作成します。スレッドが呼び出す関数はすべて thread_func で、変数 i を通じてスレッド番号が渡されます。
- 13 行目では、3 つのスレッドが同じ関数を実行します。このコードの考え方は前のセクションの名前付きセマフォの例と似ています。

- 20 行目では、スレッドは最初に実行せず、sem_wait()を直接呼び出してセマフォ sem[num]を待ちます。ここでの num はスレッド作成時に渡された順番の引数 i です。つまり、各スレッドは自分の番号と同じセマフォを待ちます。

- 23 行目から 27 行目では、セマフォを取得した後、for ループ内で情報を出力し、スリープして CPU を解放します。

- 31 行目で、このスレッドを終了します。

- 58 行目から 73 行目では、この時点で各サブスレッドがすでに作成され、セマフォを待っています。この時、元のスレッドの for ループ内で sem_post()を呼び出して順番にセマフォを解放し、pthread_join()を呼び出してそのスレッドが完了するのを待ってから次のセマフォを解放します。

- 78 行目では、sem_destroy を呼び出して各セマフォを解放します。

以下のような現象が推測できます：元のスレッドの制御下で、作成されたスレッド ABC はセマフォの解放順に従って実行され、たとえ前のスレッドが CPU を解放した操作があっても、次のスレッドは自分のセマフォを待っていない限り、CPU を得ることはありません。これにより、ACBBAC のような無秩序な操作が発生することはありません。

41.4.2.1 実験操作：

この実験のコードは base_code/system_programing/posix_sem ディレクトリに保存されており、コンパイルと実行のプロセスは以下の通りです：

```
#以下の操作は、base_code/system_programing/posix_sem コードディレクトリで実行されます。
#X86 バージョンのプログラムをコンパイルします。
make
#X86 バージョンのプログラムを実行します。
```

株式会社日昇テクノロジー

./build_x86/posix_sem_demo

#実行の出力は以下の通りです。

Create treads success

Waiting for threads to finish...

Thread 0 is starting

    Thread 0: job 0

    Thread 0: job 1

    Thread 0: job 2

    Thread 0: job 3

Thread 0 finished

Thread 0 joined

Thread 1 is starting

    Thread 1: job 0

    Thread 1: job 1

    Thread 1: job 2

    Thread 1: job 3

Thread 1 finished

Thread 1 joined

Thread 2 is starting

    Thread 2: job 0

    Thread 2: job 1

    Thread 2: job 2

株式会社日昇テクノロジー

Thread 2: job 3

Thread 2 finished

Thread 2 joined

3 つのプロセスがそれぞれ連続して出力され、セマフォの解放順に従って実行されていることがわかります。興味があれば、スレッド関数のセマフォ待機操作 sem_wait をコメントアウトして、実験現象を観察することができます。

注意：無名セマフォはシステム内でファイルを作成しないので、名前付きセマフォのように ls -l /dev/shm コマンドで確認することはできません。

第 42 章 POSIX ミューテックス

42.1 ミューテックスの基本概念

前のセマフォの章で、異なるプロセス/スレッドが何らかのクリティカルセクションにアクセスするときには、互いに排他的な保護を行う必要があることを学びました。このような排他的保護は、ある種のロックメカニズムとみなすことができます。たとえば、トイレに行くときにドアに鍵をかけて他人の入室を防ぐようなものです。Linux システムにおけるロックメカニズムは、ミューテックス、ファイルロック、読み書きロックなど、多岐にわたる概念です。

実際、セマフォも一種のロックであり、共有リソースが同時に 1 つのプロセスまたはスレッドによってのみ操作されるようにするために使用されます。

システムには多くのクリティカルセクションが存在しますが、ロックメカニズムに依存して Linux システムは完璧に動作します。ロックメカニズムは Linux システムの核心であり、プロセス/スレッドがいつクリティカルセクションにアクセスできるかを決定します。多プロセスおよび多スレッドプログラミングにおいて、ロックは非常に重要な役割を果たします。セマフォもロックとして使用できますが、今日説明するのは POSIX 標準のメカニズムであるミューテックス（mutex）です。一部の資料ではミューテックス

株式会社日昇テクノロジー

を排他制御変数と呼ぶこともありますが、これらは同じものを指しています。

ミューテックスとセマフォの違いは、ミューテックスには所有権、再帰的アクセスなどの特性があり、クリティカルセクションの排他的処理の実現によく使用されます。任意の時点でミューテックスの状態は 2 つしかなく、ロックされているかロックされていないかのどちらかです。

ミューテックスがスレッドによって保持されているとき、それはロック状態にあり、スレッドはミューテックスの所有権を持っています。そのスレッドがミューテックスを解放すると、ミューテックスはアンロック状態になり、スレッドはそのミューテックスの所有権を失います。

つまり、同時に 1 つのスレッドのみがミューテックスを取得でき、特に、そのミューテックスを保持しているスレッドは、ブロックされることなく再度そのロックを取得できるというのが、ミューテックスの再帰的アクセスの特性です。これは、一般的なセマフォと大きく異なります。セマフォでは、利用可能なセマフォが存在しないために、スレッドがセマフォを再帰的に取得しようとするとブロックされ、最終的にデッドロックが発生します。

デッドロックとは、自分自身をブロックすること、つまり自分自身をドアの外にロックし、鍵が室内にある状態です。また、デッドロックの状況には、2 つのスレッドが互いにブロックするケースもあります。これは、あなたの家の鍵が友人の家にあり、友人の家の鍵があなたの家にあるような状況で、お互いに家に入れないようなものです。

デッドロックを避けるためには、以下のルールに従うことが最善です：

- 共有リソースを操作する前に必ずロックを取得する。
- 操作を完了したら必ずロックを解放する。
- ロックを保持する時間をできるだけ短くする。
- 複数のロックを取得する場合は、取得順序と解放順序を一致させる。

スレッド間の同期機能を実現したい場合は、セマフォがより良い選択かもしれません。ミューテックスもスレッド間の同期に使用できますが、主にリソースの排他保護に使用されます。ミューテックスはリソー

株式会社日昇テクノロジー

ス保護のトークンとして機能し、あるスレッドがリソースにアクセスしたい場合、まずトークンを取得する必要があります。そして、リソースを使用した後、トークンを返却して他のスレッドがそのリソースにアクセスできるようにします。

セマフォもクリティカルセクションの保護に使用できます。スレッドがセマフォを取得した後にリソースを使用し始め、使用が終わったらセマフォを解放します。これにより、他のスレッドもセマフォを取得してリソースを使用できるようになります。しかし、セマフォには他のスレッドがセマフォを解放したときに、互いに排他操作を保証できないという潜在的な問題があります。

ミューテックスの使用は比較的単純で、ロックの形式で存在します。初期化時にミューテックスはアンロック状態にあり、スレッドによって保持されるとすぐにロック状態になります。ミューテックスは、特に以下のシナリオに適しています：

1. クリティカルセクションの保護。
2. スレッドがミューテックスを複数回取得する可能性がある場合。これにより、同一スレッドが複数回再帰的に保持することによって発生するデッドロックの問題を回避できます。

マルチスレッド環境では、複数のスレッドが同一のクリティカルセクションを競合するシナリオがよくあります。ミューテックスは、クリティカルセクションの保護に使用され、排他的なアクセスを実現します。さらに、ミューテックスはセマフォに存在する優先度反転の問題の影響を軽減することができます。

たとえば、2 つのスレッドがシリアルポートを介してデータを送信する必要がある場合、ハードウェアリソースが 1 つしかないため、2 つのスレッドが同時にデータを送信することはできません。これによりデータエラーが発生します。そこで、シリアルポートリソースを保護するためにミューテックスを使用できます。一方のスレッドがシリアルポートを使用している間、他のスレッドはシリアルポートを使用することができず、一方のスレッドがシリアルポートの使用を終了した後、他のスレッドがシリアルポートの使用権を得ることができます。

マルチスレッド環境では、複数のスレッドが同一のクリティカルセクションにアクセスするシナリオが存

株式会社日昇テクノロジー

在しますが、そのリソースはスレッドによって独占的に処理されます。他のスレッドはリソースが使用中の場合、そのクリティカルセクションにアクセスすることは許可されません。この時、リソース保護のためにミューテックスが使用されますが、ミューテックスはどのようにしてこのような競合を避けるのでしょうか？

異なるスレッドがクリティカルセクションに同期的にアクセスする際にミューテックスを使用すると、スレッドがミューテックスを取得しなければリソースにアクセスできません。一度に 1 つのスレッドが成功してミューテックスを取得すると、ミューテックスは即座にロック状態になり、他のスレッドはミューテックスを取得できずにリソースにアクセスできません。この時、スレッドには 2 つの選択肢があります：引き返すか、ミューテックスが解放されるまでブロックされ続けるかです。ミューテックスが保持されているスレッドによって解放された後、スレッドはミューテックスを取得してクリティカルセクションにアクセスでき、この時ミューテックスは再びロックされます。これにより、同時に 1 つのスレッドのみがこのクリティカルセクションにアクセスしていることが保証され、クリティカルセクションの操作の安全性が保証されます。

42.2 ミューテックスの初期化

42.2.1 静的初期化

ミューテックスを使用する前に、ミューテックスを初期化する必要があります。POSIX 標準では、静的初期化と動的初期化の 2 つの方法をサポートしています。静的初期化を行う場合、以下のコードのいずれかを使用できます（一つを選択してください）：

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;

pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

株式会社日昇テクノロジー

pthread_mutex_t はミューテックスの構造体であり、ミューテックス構造を定義し、それに値を割り当てることで、異なるタイプのミューテックスをリストします。これら 3 種類のミューテックスの違いは、ミューテックスを所有していない他のスレッドがミューテックスを取得しようとしたときに、ブロックして待つ必要があるかどうかに主にあります：

- PTHREAD_MUTEX_INITIALIZER：デフォルトのミューテックス、つまり高速ミューテックスをリストします。スレッド 1 がミューテックスを所有している場合、その時点でミューテックスはロック状態にあり、スレッド 2 がミューテックスを取得しようとすると、ミューテックスを所有するスレッド 1 がアンロックするまでスレッド 2 はブロックされます。
- PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP：再帰的ミューテックスをリストします。スレッド 1 がミューテックスを所有している場合、スレッド 2 がミューテックスを取得しようとすると成功せず、ブロックされます。しかし、スレッド 1 が再度ミューテックスを取得しようとすると、成功し、ミューテックスの所有回数が 1 増加します。
- PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP：エラーチェックミューテックスをリストします。これは高速ミューテックスのノンブロッキングバージョンで、エラーコードを即座に返します（スレッドはブロックされません）。

42.2.2 動的初期化

ミューテックスの動的初期化は、pthread_mutex_init() 関数を呼び出して行います。この関数のプロトタイプは以下の通りです：

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

pthread_mutex_init() 関数は動的にミューテックスを初期化するためのもので、パラメータの説明は以下の通りです：

- mutex は初期化するミューテックス構造のポインターです。

株式会社日昇テクノロジー

• mutexattr は属性パラメータで、ミューテックスの属性を設定することができます。これにより、属性がミューテックスの振る舞いを制御します。mutexattr が NULL の場合、デフォルトのミューテックス属性が使用されます。デフォルト属性は高速ミューテックスです。

42.3 ミューテックスの取得と解放

前述のように、クリティカルセクションにアクセスするにはミューテックスを取得する必要があります。ミューテックスを取得することは、リソースへのアクセス

権を得ることに相当します。まるで家の鍵を持っているときにだけドアを開けられるように。

ミューテックスがアンロック状態のときにのみ、スレッドはミューテックスを取得することができます。あるスレッドがミューテックスを所有している場合、他のスレッドはそのミューテックスを取得することができず、所有しているスレッドが解放するまで待つ必要があります。スレッドはミューテックス取得関数を使用してミューテックスの所有権を取得します。

スレッドによるミューテックスの所有権は独占的です。任意の時点でミューテックスは 1 つのスレッドにのみ所有されることができます。ミューテックスがアンロック状態であれば、そのミューテックスを取得するスレッドは成功し、ミューテックスの所有権を得ます。しかし、ミューテックスがロック状態にある場合、ミューテックスのタイプに応じた処理が行われます。デフォルトでは高速ミューテックスが使用され、そのミューテックスを取得しようとするスレッドは成功せず、ミューテックスが解放されるまでブロックされます。もちろん、同じスレッドがミューテックスを繰り返し取得しようとすると、デッドロックの結果になることもあります。

ミューテックスを取得するためには 2 つの関数があります。mutex パラメータは操作するミューテックスを指定します：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);


int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

株式会社日昇テクノロジー

int pthread_mutex_unlock(pthread_mutex_t *mutex);

- pthread_mutex_lock() 関数は、クリティカルセクションへのアクセス権を獲得するために使用されます。他のスレッドが既にミューテックスをロックしている場合、この関数は指定されたミューテックスが解除されるまでスレッドをブロックします。
- pthread_mutex_trylock() は、pthread_mutex_lock() 関数のノンブロッキングバージョンです。これを使用すると、現在のスレッドはブロックされず、ミューテックスが既に使用中の場合は EBUSY エラーを即座に返します。
- 共有リソースにアクセスした後、pthread_mutex_unlock() 関数を通じて使用中のミューテックスを解放する必要があります。これにより、システムの他のスレッドがミューテックスを獲得し、そのリソースにアクセスする機会が得られます。

要するに、ミューテックスの使用プロセスは次の通りです：

1. スレッドがミューテックスを取得します。
2. 次に共有リソースにアクセスします。
3. 最後にミューテックスを解放します。

42.4 ミューテックスの破棄

pthread_mutex_destroy() 関数は、ミューテックスを破棄するために使用されます。ミューテックスが不要になった場合、これを使用して破棄できます。mutex パラメータは、破棄するミューテックスを指定します：

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

42.5 ミューテックスの実験

この実験の主な目的は、ミューテックスの排他状況を検証することです。システムは 3 つのスレッドを作成し、これら 3 つのスレッドにクリティカルセクションがアクセスされると仮定します。その場合、こ

株式会社日昇テクノロジー

れら 3 つのスレッドが順番に、かつ同時にはクリティカルセクションにアクセスできないようにしたいと考えます（クリティカルセクションが sleep()関数を呼び出すと仮定します）。したがって、アクセスできるスレッドを制限するためにミューテックスを使用することができます。ミューテックスを取得したスレッドは、クリティカルセクションにアクセスできます。

コードは以下の通りです：

リスト 1: POSIX 無名セマフォ

（base_code/system_programing/mutex/sources/mutex.c ファイル）

```
1 #include <unistd.h>
2 #include <fcntl.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <pthread.h>
6
```

```
7 #define THREAD_NUMBER 3 /* スレッド数 */
8
9 pthread_mutex_t mutex;
10
11 void *thread_func(void *arg)
12 {
13     int num = (unsigned long long)arg; /* sizeof(void*) == 8、sizeof(int) == 4 (64 ビット) */
14     int sleep_time = 0;
15     int res;
```

株式会社日昇テクノロジー

```
16
17    /* ミューテックスをロック */
18    res = pthread_mutex_lock(&mutex);
19    if (res)
20    { /* ロック失敗 */
21        printf("スレッド %d のロック失敗¥n", num);
22
23        /* ミューテックスをアンロック */
24        pthread_mutex_unlock(&mutex);
25
26        pthread_exit(NULL);
27    }
28
```

```
29    printf("スレッド %d がミューテックスを保持¥n", num);
30
31    /* 一定時間スリープ */
32    sleep(2);
33
34    printf("スレッド %d がミューテックスを解放¥n¥n", num);
35
36    /* ミューテックスをアンロック */
37    pthread_mutex_unlock(&mutex);
```

株式会社日昇テクノロジー

```
38
39     pthread_exit(NULL);
40 }
41
42
43 int main(void)
44 {
45     pthread_t thread[THREAD_NUMBER];
46     int num = 0, res;
47
48     srand(time(NULL));
49
50     /* ミューテックスの初期化 */
```

```
51     pthread_mutex_init(&mutex, NULL);
52     for (num = 0; num < THREAD_NUMBER; num++)
53     {
54         /* スレッドの作成 */
55         res = pthread_create(&thread[num], NULL, thread_func, (void*)(unsigned long long)num);
56         if (res != 0)
57         {
58             printf("スレッド %d の作成失敗¥n", num);
59             exit(res);
```

株式会社日昇テクノロジー

```
60        }
61    }
62
63    for (num = 0; num < THREAD_NUMBER; num++)
64    {
65        /* スレッドの終了を待つ */
66        pthread_join(thread[num], NULL);
67    }
68
69    /* ミューテックスの破棄 */
70    pthread_mutex_destroy(&mutex);
71
72    return 0;
73 }
```

このコードは、main 関数から直接プロセスを分析すると以下の通りです：

- 45 行目で、pthread_mutex_init() を呼び出し、ミューテックスを動的に初期化しました。
- 54 行目で、for ループ内で 3 つのスレッドを作成し、それぞれのスレッドが thread_func 関数を呼び出し、変数 num を介してスレッド番号を渡します。
- 11 行目で、3 つのスレッドはすべて同じ関数を実行します。

  - 18 行目で、pthread_mutex_lock() を呼び出し、ミューテックスを待ちます。他のスレッドがミューテックスを占有している場合、スレッドはここでブロックされ、待ちます。

  - 29 行目と 34 行目で、これらの出力情報のコードはミューテックスで保護された範囲内にあるため、ペアで出現し、同じミューテックスを要求する他のスレッドによって中断されることはありま

株式会社日昇テクノロジー

せん。

- 32 行目で、ミューテックスを取得した後、リソースに対して操作を行います。ここでは、CPU の使用を解放するために直接 sleep を模擬します。

- 37 行目で、pthread_mutex_unlock() を呼び出し、他のスレッドがクリティカルリソースを使用できるように解除します。

• 65 行目で、この時点で各子スレッドは既に作成され、実行されており、元のスレッドの for ループ内で pthread_join() を呼び出し、他のスレッドの実行が完了するのを待ちます。

• 69 行目で、各スレッドの実行が完了した後、pthread_mutex_destroy() を呼び出し、ミューテックスを破棄します。

このように、元のスレッドの制御下で、それが作成したスレッド ABC は、ミューテックスの保護下で出力コードのペアである 29 行目、34 行目の情報を出力し、保護範囲内で CPU の使用を解放する sleep 操作があっても、他の 2 つのスレッドは CPU の利用機会を得られません。なぜなら、彼らはロックを取得していないからです。したがって、制御下で出現する出力情報は、A1A2、B1B2、C1C2 であり、A1B1C1、A2B2C2 のような情報の混乱は発生しません。

42.5.1 実験操作

この実験のコードは base_code/system_programing/mutex ディレクトリに保存されており、コンパイルおよび実行プロセスは以下の通りです：

```
# 以下の操作は base_code/system_programing/mutex コードディレクトリで行います
# X86 版プログラムをコンパイル
make
# X86 版プログラムを実行
./build_x86/mutex_demo
```

株式会社日昇テクノロジー

実行結果

Thread 0 is hold mutex

Thread 0 freed mutex

Thread 1 is hold mutex

Thread 1 freed mutex

Thread 2 is hold mutex

Thread 2 freed mutex

明確でない場合は、ミューテックス関連のコードをコメントアウトしてみてください。つまり、18～27行目のコードをコメントアウトして、ミューテックス操作がない場合にスレッドがクリティカルリソースにどのようにアクセスするかを確認し、再コンパイルして実行します。実行結果は以下の通りです：

```
# 18～27 行目のコードをコメントアウトして再コンパイル・実行

# 以下の操作は base_code/system_programing/mutex コードディレクトリで行います

# X86 バージョンのプログラムをコンパイル

make

# X86 バージョンのプログラムを実行

./build_x86/mutex_demo



# 18～27 行目のコードをコメントアウトして再コンパイル・実行した後の出力
```

株式会社日昇テクノロジー

Thread 0 is hold mutex

Thread 1 is hold mutex

Thread 2 is hold mutex

Thread 0 freed mutex

Thread 2 freed mutex

Thread 1 freed mutex

ロックの保護がないため、sleep 期間中に他のスレッドが CPU を得てリソースを操作しました。ここで、A1C1B1、A2C2B2 の情報出力順序が生じました。これは、スレッド A が重要なリソースを使用している間に、スレッド B と C も同時に使用したことを意味します。リソースが 1 つしかない場合、予測不能な問題が発生する可能性があります。

第 43 章 ネットワークプログラミング

インターネットが人類社会にもたらした巨大な変革は、誰もが目の当たりにしています。それは人間の生活のあらゆる面を変えました。インターネット通信の本質はデジタル通信であり、あらゆるデジタル通信は通信プロトコルの制定なしには成り立ちません。通信機器は、合意された統一された方法で情報をパッケージ化および解析することによってのみ、通信を実現することができます。インターネット通信で遵守すべき多くのプロトコルは、TCP/IP と総称されます。

43.1 ネットワーク関連知識の簡単な紹介

ネットワークに関する知識はあまりにも広大で、詳しく説明するには数冊の本を書く必要があるため、ここでは簡単に触れるだけにします。

TCP/IP はプロトコル群であり、多くのプロトコルを含んでいます。しかし、ネットワークアプリケーション開発者にとっては、その中のアプリケーション層のプロトコル、例えば HTTP、FTP、MQTT などがより多く耳にするものかもしれません。

株式会社日昇テクノロジー

例えば：

HTTP プロトコルは Hyper Text Transfer Protocol（ハイパーテキスト転送プロトコル）の略で、HTTP の使用は最も広範囲に及びます。例えば、日常的にパソコンを使用する際の一般的な操作：パソコンを起動し、ブラウザを開き、ウェブサイトのアドレスを入力し、エンターキーを押す、この瞬間に HTTP 通信が開始されます。HTTP プロトコルはアーキテクチャの上で動作し、ブラウザは HTTP クライアントとして URL を使用して HTTP サーバー、すなわち WEB サーバーに対してリクエストを送信します。WEB サーバーは受信したリクエストに基づいてクライアントにレスポンス情報を送信します。このようなブラウザとサーバー間の HTTP 通信により、我々は外出することなく世界各地からの情報を得ることができます。また、ウェブページは大型サーバーの専売特許ではなく、今日の IoT の波に乗って、エアコン、冷蔵庫、プラグ、ルーターなど、あらゆる場所にある小型デバイスにも組み込まれています。物理的なリンクが問題なく機能している場合、ユーザーは携帯電話やタブレットのブラウザを使用して、これらのデバイスをいつでもどこでも監視することができます。

FTP（File Transfer Protocol）は、ファイル転送プロトコルの略称です。FTP はアプリケーション層で動作するネットワークプロトコルであり、主にホスト間でファイルを共有し、2 台のデバイス間でファイルを転送（双方向転送）するために使用されます。これはクライアント-サーバーフレームワークシステムでもあります。ユーザーは FTP プロトコルをサポートするクライアントプログラムを通じて、リモートホスト上の FTP サーバープログラムに接続できます。クライアントプログラムからサーバープログラムにコマンドを送信し、サーバープログラムはユーザーの送信したコマンドを実行し、その結果をクライアントに返します。FTP は基本的なファイルのアップロード/ダウンロード機能だけでなく、ディレクトリ操作、権限設定、認証メカニズムも提供し、多くのクラウドストレージのファイル転送機能は FTP を基に実装されています。

IOT の発展の初期段階では、物联网シナリオでのデバイスがどのアプリケーション層プロトコルを使用して通信するかが、常に議論の的でした。多くの開発者がウェブ開発のモデルに慣れているため、しばしば

株式会社日昇テクノロジー

HTTP を通信手段として選択します。HTTP には以下のような不利な点があります：HTTP は同期プロトコルで、デバイスはサーバーの応答を待たなければならず、デバイスの数が多く、ネットワークが不安定なシナリオでは、同期通信を実現することが難しい；HTTP は単方向で、デバイスはサーバーにデータを送信することしかできず、ネットワークからのデータを受動的に受け取ることができないため、リアルタイム制御の場合には適していません；HTTP は多くのヘッダーとルールを持つ重量級のプロトコルで、デバイスに実装するには大量のシステムリソースを消費します。これらの状況に基づき、MQTT や COAP などの軽量で非同期の通信プロトコルが、IOT デバイスの開発者から特に好まれるようになりました。特に MQTT は、IBM が 1990 年に設計し、2014 年に OASIS のオープンスタンダードとなった通信プロトコルです。近年、MQTT の使用は爆発的に増加しており、IOT を統一する傾向があります。また、MQTT は IOTR 以外の領域でも広く利用されており、多くの企業がスマートフォンアプリのメッセージプッシュやインスタントメッセージングなどの機能を実装するために MQTT を使用しています。

組み込みデバイスのインターネット接続需要が高まっています。その理由は以下の通りです：

1. 近年、ネットワーク接続機能を備えた MCU や SoC が次々と登場し、オープンソースで軽量な TCP/IP プロトコルスタックが成熟し、完善してきました。また、クラウドプラットフォームの市場が益々繁栄しており、これらの要因が組み込みデバイスのネットワーク接続コストを大幅に削減し、リソースが限られた低性能デバイスのインターネット接続を可能にしました。
2. 「IoT+」の波が日に日に高まり、デバイスが遠隔監視可能であることが多くの製品の技術要件となっています。
3. 人々はデバイスの「スマート化」に対する熱意がますます高まっており、大データ、画像処理、音声認識、機械学習などの今日のホットな機能はすべてクラウドで統合され、クラウドプラットフォームが提供できるサービスとなりました。エンドデバイスのほとんどは、計算能力や記憶容量が限られています。これらのデバイスが「スマート」機能を手に入れたい場合、最も便利な方法はクラウドプラットフォームに接続し、各種クラウドサービスを利用することです。

株式会社日昇テクノロジー

インターネットの基盤は TCP/IP です。TCP/IP は非常に複雑なプロトコルファミリーであり、その設計思想や実装原理を全て明確に説明できたとしても、それを学ぶ時間やエネルギーをすべての人が持っているわけではありません。そのため、この本の重点は TCP/IP の解釈ではなく、その応用にあります。さらに、TCP/IP の複雑さは、単純にうまく使えるものではないということも意味します。アプリケーション開発にのみ焦点を当てていても、その多くの概念や設計思想を理解していなければ、正確で効率的で堅牢なアプリケーションを開発することはできません。

インターネットが人類社会にもたらした大きな変革は、誰の目にも明らかで、人類の生活のあらゆる面をほぼ変えてしまいました。インターネット通信の本質はデジタル通信であり、どんなデジタル通信も通信プロトコルの策定なしには成り立ちません。通信装置は、合意された統一された方法で情報をパッケージングおよび解析することによってのみ、通信を実現することができます。インターネット通信で守られるべき多くのプロトコルは、TCP/IP と総称されます。

43.1.1 ネットワークプロトコルの階層モデル

TCP/IP は複雑なプロトコルの集まりであり、ARP、IP、ICMP、UDP、TCP、DNS、DHCP、HTTP、FTP、MQTT など多くのネットワークプロトコルを含んでいます。これらのプロトコルは機能に基づいていくつかの異なる層に分類されます。たとえば、HTTP、FTP、MQTT はアプリケーション層に属しています。では、なぜ TCP/IP は層に分けられるのでしょうか？そして、その分層の基準は何でしょうか？

株式会社日昇テクノロジー

層	内容
アプリケーション層	DNS、HTTP、FTP、SMTP メールプロトコル
トランスポート層	主に TCP UDP プロトコルを使用
ネットワーク層	主に IP ICMP ARP プロトコル
リンク層	MAC 層
物理層	主に物理的な伝送媒体を定義

図 1: tcpip001

TCP/IP プロトコルスタック内の異なるプロトコルは異なる機能を果たし、あるプロトコルの実装は他のプロトコルに依存しています。このような依存関係に基づき、プロトコルスタックを層に分けることができます。図に示されるように、下層のプロトコルは隣接する上層のプロトコルにサービスを提供し、それが上層プロトコルの実現の基礎となります。

物理層（PHY）は、伝送信号に必要な物理的レベルや媒体特性を規定します。

リンク層（MAC）は、データフレームがネットワークカードによって受信できる条件を規定します。最も一般的な方法は、ネットワークカードの MAC アドレスを使用することで、送信側は送りたいデータフレームの先頭に受信側のネットワークカードの MAC アドレス情報を加え、受信側は自分の MAC アドレス情報をリッスンして初めて、そのデータを受信し処理します。

株式会社日昇テクノロジー

各ネットワークデバイスは自身のネットワークアドレスを持つべきで、ネットワーク層はホストのネットワークアドレスの定義方法と、ネットワークアドレスと MAC アドレス間のマッピング方法、つまり ARP プロトコルを規定します。ネットワーク層は、ホスト間のデータパケットの転送を実現しますが、一台のホスト内部では複数のネットワークプログラムが動作している可能性があります。

トランスポート層は、データパケットがどのアプリケーションプログラムに属するかを区別することができます。つまり、トランスポート層はデータパケットの端から端への転送を実現します。また、データパケットの転送プロセス中にパケットの損失、乱序、重複などの現象が発生する可能性がありますが、ネットワーク層はこれらのエラーに対処するメカニズムを提供していません。一方、トランスポート層はこれらの問題を解決することができます。たとえば TCP プロトコルの場合です。

アプリケーション層以下の作業はデータの伝達作業を完成し、アプリケーション層はどのようにこれらのデータを適用および処理するかを決定します。多くのアプリケーション層プロトコルが存在する理由は、インターネットで伝達されるデータの種類が多く、差異が大きく、適用シナリオが非常に多様であるためです。

43.1.2 プロトコル層間のデータのカプセル化とデカプセル化

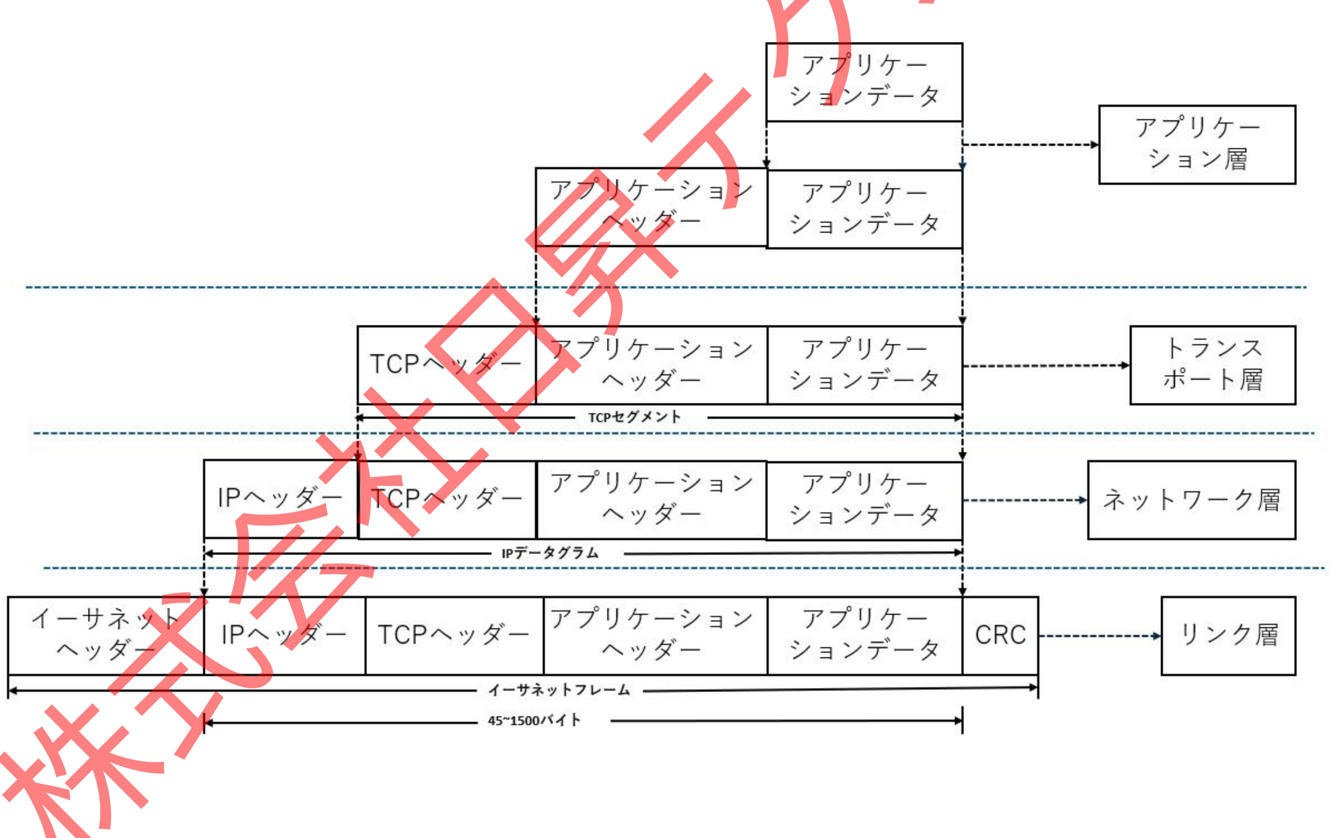
ここでは、データの送信と受信プロセスで、TCP/IP が何を行っているかを簡単に説明します。

ユーザーがデータを送信するとき、アプリケーション層からデータを転送層に渡します。これはアプリケーション層の操作であり、アプリケーション層には関連するプロトコルがあり、ユーザーのデータを MQTT、HTTP などのプロトコルでカプセル化し、最終的にアプリケーション層が転送層のインターフェースを呼び出してデータを転送層に渡します。転送層はデータの前に転送層のヘッダー（この場合は TCP プロトコルを例に、TCP ヘッダー、または UDP ヘッダーも可能）を追加し、次にデータをネットワーク層に渡します。同様に、ネットワーク層はデータの前にネットワーク層のヘッダー（IP ヘッダー）を追加し、次にデータをリンク層に渡します。リンク層はデータに最後のカプセル化、つまりリンク層のヘッダ

株式会社日昇テクノロジー

ー（この場合はイーサネットインターフェースを例に）を追加し、次にデータをネットワークカードに渡します。最後に、ネットワークカードはデータを物理リンク上の電気信号に変換し、データはこのようにしてネットワークに送信されます。データの送信プロセスは、TCP/IP の各層がデータにカプセル化を行うプロセスとして要約できます。

デバイスのネットワークカードがあるデータパケットを受信すると、それを受信バッファに配置し、TCP/IP カーネルに通知します。次に、TCP/IP カーネルが作業を開始し、受信バッファからデータパケットを取り出し、データパケット内のプロトコルヘッダー情報を層ごとに解析し、最終的にデータをあるアプリケーションプログラムに渡します。データの受信プロセスは、送信プロセスとは正反対で、TCP/IP の各層がデータを解析するプロセスとして要約できます。



43.2 IP プロトコル

IP プロトコル（Internet Protocol）、すなわちインターネットプロトコルは、ネットワーク層で動作するもので、TCP/IP プロトコルスタックの中核プロトコルです。上位プロトコルはすべて、IP プロトコルが提供するサービスに依存しています。IP プロトコルは、データグラムを送信元ホストから目的ホストに送

株式会社日昇テクノロジー

信する責任を持ち、IP アドレスを一意の識別コードとして使用します。簡単に言うと、異なるホスト間の IP アドレスは異なり、データグラムの送信過程で、IP プロトコルはデータグラムを分割することがあり、また、受信時には分割されたデータグラムを再構築する必要がある場合があります。

IP プロトコルは、接続のない信頼できないデータグラム配信プロトコルであり、プロトコル自体はエラーチェックや回復メカニズムを提供しません。

43.2.1 IP アドレスの概要

TCP/IP が設計された当初、インターネット上の各ホストを識別するために、ネットワークに接続された各ホストに IP アドレス（Internet Protocol Address）が割り当てられました。これは 32 ビットの整数アドレスで、有効な IP アドレスのみがインターネットに接続して他のホストと通信できます。IP アドレスはソフトウェアアドレスであり、ハードウェアアドレスではありません。ハードウェア MAC アドレスはネットワークカードに格納され、ローカルネットワーク内で目的のホストを見つけるために使用されます。一方、IP アドレスにより、あるネットワークのホストが別のネットワークのホストと通信できるようになり、これらのホスト間の MAC アドレスを気にする必要がありません。

IP アドレスについて学ぶ前に、ホストとルーターがネットワークに接続する通信方法について簡単に説明します。一般に、ホストはネットワークに接続するために 1 つのリンクのみを持ち、通常は 1 つのネットワークカードを持っています。ホストがデータグラムを送信したい場合、それをそのリンク上で送信します。1 つのネットワークカードは 1 つの MAC アドレスと IP アドレスに対応しますが、複数のネットワークカードを持つホストもあり、それにより複数の MAC アドレスと IP アドレスを持つことがあります。

各 IP アドレスは 32 ビット（4 バイト）の長さで、合計で 2^32 の可能な IP アドレスがあり、約 40 億の IP アドレスが使用可能です。これらのアドレスは通常、ドットで区切られた 10 進法（dotted-decimal notation）で記述されます。つまり、アドレスの各バイトはその 10 進数形式で記述され、バイト

株式会社日昇テクノロジー

間はドットで区切られます。たとえば、IP アドレスが 192.168.0.122 の場合、192 はそのアドレスの最初の 8 ビットの 10 進数等価数、168 はそのアドレスの 2 番目の 8 ビットの 10 進数等価数であり、以降同様です。

43.2.2 IP アドレスのアドレス割り当て

全世界のインターネット上では、各ホストが唯一の IP アドレスを身分証明として持つ必要があります。そこで、これら多数の IP アドレスはどのように割り当てられているのでしょうか？これは、単に IP アドレスを自由に選ぶことはできません。実際には、各ホストの IP アドレスの一部は、その所在するサブネットによって決定されます。これにより、IP アドレスの分類アドレス割り当ての概念が登場しました。ネットワークのタイプによって、IP アドレスがネットワーク部分とノード部分にどのように分割されるかが決定されます。分類アドレス割り当てでは、設計者はすべての IP アドレスを A、B、C、D、E の 5 つのクラスに分けました。それぞれのクラスのアドレスは、IP アドレスの一部を構成します。詳細は以下のとおりです：

ネットワークタイプ	第1バイト	第2バイト	第3バイト	第4バイト
クラスA	0 ネットワーク番号	ホスト番号		
クラスB	1 0 ネットワーク番号		ホスト番号	
クラスC	1 1 0 ネットワーク番号			ホスト番号
クラスD	1 1 1 0 マルチキャストアドレス			
クラスE	1 1 1 1 保留未使用			

3: tcpip003

- A クラスのネットワークアドレスでは、最初のバイトの最初のビットは必ず 0 でなければなりません。したがって、A クラスのネットワークアドレスの最初のバイトの範囲は 0～127 です（ただし、0 と 127 は有効な A クラスのネットワークアドレス番号ではありません）。A クラスアドレスには合計で 128 のネットワーク番号がありますが、特別な用途のために 3 つのネットワーク番号が使用されているため、インターネット上で使用可能なのは 125 個です。それぞれのネットワーク番号の後の

株式会社日昇テクノロジー

3 バイトは、A クラスのネットワークアドレスの異なるネットワーク番号のホスト数をリストします（最大 2^24-2=16777214 個）。したがって、A クラスのネットワークは約 2097151750 個のホストをサポートできます。これは約 40 億の IP アドレスの半分を占めます。これらの IP アドレスは通常、世界の超大型機関に割り当てられますが、通常、どの機関もこれほど多くの IP アドレスを使用することはありません。したがって、A クラスアドレスの多くの IP アドレスが無駄になることがあります。

- B クラスのネットワークアドレスでは、最初のバイトの最初のビットは 1 でなければならず、2 番目のビットは 0 でなければなりません。したがって、B クラスのネットワークアドレスの最初のバイトの範囲は 128～191 です。前 2 バイトの残りの 14 ビットはネットワーク番号をリストし、最大 16384 のネットワーク番号があります。そのうち 16 のネットワーク番号は予約されています。したがって、企業に割り当てられるネットワーク番号は 16368 個あり、各ネットワーク番号には 2^16-2=65534 個のホストが含まれます。全体として、B クラスのネットワークは約 1072660512 個のホストをサポートできます。これはすべての IP アドレスの約四分の一を占めます。

- C クラスのネットワークアドレスでは、最初のバイトの最初のビットは 1 でなければならず、2 番目と 3 番目のビットも 1 で、4 番目のビットは 0 でなければなりません。したがって、C クラスのネットワークアドレスの最初のバイトの範囲は 192～223 です。C クラスアドレスの最初の 3 バイトはネットワーク番号をリストし（残りの 21 ビット）、2097152 のネットワーク番号がありますが、そのうちの 256 個が予約されています。したがって、C クラスアドレスの使用可能なネットワーク番号は 2096896 個で、各ネットワーク番号には 254 個のホストが含まれます。全体として、C クラスのネットワークは約 532611584 個のホストをサポートできます。これはすべての IP アドレスの約八分の一を占めます。

- D クラスの IP アドレスでは、最初のバイトの最初の 4 ビットは 1110 でなければなりません。したがって、D クラスの IP アドレスの最初のバイトの範囲は 224～239 です。これはすべての IP ア

株式会社日昇テクノロジー

ドレスの約十六分の一を占めます。D クラスアドレスは特定のネットワークを指さず、現在はマルチキャストに使用されています。

- E クラスの IP アドレスでは、最初のバイトの最初の 4 ビットは 1111 でなければなりません。したがって、E クラスの IP アドレスの最初のバイトの範囲は 240〜255 です。これはすべての IP アドレスの約十六分の一を占めます。E クラスアドレスは将来の使用のために予約されています。32 ビットがすべて 1 の IP アドレス（255.255.255.255）はブロードキャストアドレスとして使用されます。

カテゴリー	第一バイト(2進数)	第一バイトの値の範囲	ネットワーク数	ホスト数	適用範囲
A クラス	0XXX XXXX	0~127	125	16777214	大型ネットワーク
B クラス	10XX XXXX	128~191	16368	65534	中型ネットワーク
C クラス	110X XXXX	192~223	2097152	254	小型ネットワーク
D クラス	1110 XXXX	224~239	—	—	マルチキャスト
E クラス	1111 XXXX	240~255	—	—	予約済み

43.2.3 特別な IP アドレス

上述のネットワークアドレスの他にも、いくつかの特別な用途のアドレスがあります。これらのアドレスはどのネットワークのホストにも割り当てられてはいけません。以下は、いくつかの一般的な特別なアドレスについて簡単に説明します。

43.2.3.1 制限されたブロードキャストアドレス

ブロードキャスト通信は一対全ての通信方法です。制限されたブロードキャストアドレスは、インターネット全体を対象としています。デバイスが IP データグラムをネットワーク全体で受信させたい場合、すべての宛先アドレスが 1 のブロードキャストパケットを送信します。しかし、これはインターネット全体に災害的な負荷をもたらすため、ルーターは 255.255.255.255 の宛先アドレスを持つブロードキャストデ

株式会社日昇テクノロジー

ータパケットの転送を禁止します。したがって、このようなデータパケットはローカルネットワーク（LAN）内にのみ存在します。255.255.255.255 はこのネットワークセグメント内のすべてのホストを指し、"家の中の人全員に聞こえるように"すべてのホストに通知します。

注意：イーサネットのブロードキャストアドレス（255-255-255-255-255-255）と混同しないでください。

43.2.3.2 直接ブロードキャストアドレス

制限されたブロードキャストアドレスは、ネットワーク番号とホスト番号が共に 1 のアドレスですが、直接ブロードキャストアドレスはホスト番号が全て 1 であることによって得られるアドレスです。ブロードキャストアドレスは、そのネットワーク内のすべてのホストを代リストします。たとえば、IP アドレスが 192.168.0.181 の場合、これは C クラスアドレスであるため、ホスト番号は 1 バイトのみです。したがって、ホスト番号を全て 1 にすることで、ブロードキャストアドレス 192.168.0.255 が得られ、このアドレスにデータを送信すると、同じネットワーク内のすべてのホストがデータを受信できます。

A、B、C クラスアドレスのブロードキャストアドレスの構造は以下の通りです。

- A クラスアドレスのブロードキャストアドレスは、XXX.255.255.255 です（XXX は A クラスアドレスの最初のバイトの値範囲）。
- B クラスアドレスのブロードキャストアドレスは、XXX.XXX.255.255 です（XXX は B クラスアドレスの最初の 2 バイトの値範囲）。
- C クラスアドレスのブロードキャストアドレスは、XXX.XXX.XXX.255 です（XXX は C クラスアドレスの最初の 3 バイトの値範囲）。

注意：このアドレスは IP データグラムで宛先アドレスとしてのみ使用できます。また、直接ブロードキャストアドレスにより、ネットワークセグメント内でデバイスに割り当てることができるアドレス数が 1 つ減少します。

株式会社日昇テクノロジー

43.2.3.3 マルチキャストアドレス

マルチキャストアドレスは、一対多の通信に使用されます。つまり、1 つの送信者と複数の受信者がいますが、受信者の数に関係なく、送信者はデータパケットを 1 回だけ送信します。マルチキャストアドレスは分類アドレスの D クラスに属しており、D クラスアドレスは宛先アドレスとしてのみ使用でき、ホストのソースアドレスとしては使用できません。

43.2.3.4 ループバックアドレス

127 のネットワークセグメント内のすべてのアドレスはループバックアドレスと呼ばれ、ネットワークプロトコルが正常に動作しているかどうかをテストするために主に使用されます。たとえば、コンピュータで ping コマンドを使用して 127.1.1.1 に ping を送ることで、ローカル TCP/IP プロトコルが正常に動作しているかをテストできます。わかりやすく言えば、「自分自身」です。127 のネットワークセグメントの IP アドレスをホストアドレスとして使用することはできず、したがって A クラスアドレスは利用可能なネットワーク番号が 1 つ減ります。

43.2.3.5 このネットワークのこのホスト

IP アドレスの 32 ビットがすべて 0 のアドレス（0.0.0.0）は、このネットワークのこのホストをリストし、IP データグラムでソース IP アドレスとしてのみ使用できます。これは、デバイスが起動時に自分の IP アドレスを知らない場合に発生します。DHCP を使用して IP アドレスを割り当てるネットワーク環境では、このようなアドレスがよく見られます。ホストが利用可能な IP アドレスを取得するために、DHCP サーバーに IP データグラムを送信し、ソースアドレスとしてこのアドレス（0.0.0.0）を使用し、宛先アドレスとして 255.255.255.255 を使用します（この時点でホストは DHCP サーバーの IP アドレスを知りません）。その後、DHCP サーバーはこのホストが現時点で IP アドレスを持っていないことを知り、ホストに IP アドレスを割り当てます。

株式会社日昇テクノロジー

43.3 UDP プロトコル

UDP は User Datagram Protocol の略称で、日本語でユーザーデータグラムプロトコルと呼ばれています。これは、接続なしで信頼性のないプロトコルであり、単に一方のホストから別のホストへのデータ転送機能を実現するものです。これらのデータは IP 層を介して送信され、ネットワークを通じて転送されますが、目的地のホストに到着する順序は予測不可能です。そのため、アプリケーションにはこれらのデータを並べ替える必要があり、これが大きな不便を引き起こします。さらに、UDP プロトコルには、流量制御や輻輳制御などの機能がありません。送信側では、UDP は上位アプリケーションのデータを UDP パケットに封入し、簡単なエラーチェックを行った後、IP データグラムに封入して送信します。受信側では、データが受信されても、ソースホストに応答を送信することはなく、受信したデータに送信エラーがあった場合、受信側はその UDP パケットを破棄し、ソースホストに通知しません。このようにして転送されるデータの正確性は保証されません。正確性が求められる場合は、アプリケーションによる保証が必要です。UDP プロトコルの特徴は以下の通りです。

1. 接続なし、信頼性がありません。

2. データ提供サービスをできるだけ提供し、エラーが発生した場合は直接破棄し、フィードバックはありません。

3. メッセージ指向で、送信側の UDP は上層データに直接 UDP ヘッダーを追加し、チェックした後、IP 層に提出します。受信側では、UDP パケットを受信した後、簡単にチェックし、データを直接上層アプリケーションに提供します。

4. 一対一、一対多、多対一、多対多のインタラクティブ通信をサポートします。

5. 速度が速く、UDP は TCP のハンドシェイク、確認、ウィンドウ、再送信、輻輳制御などのメカニズムがないため、データ転送時に非常に速いです。ネットワークが混雑していても、UDP は送信データ速度を低下させません。

株式会社日昇テクノロジー

UDP には多くの欠点がありますが、現在のネットワーク環境では、UDP プロトコルによる転送エラーの確率は非常に低く、リアルタイム性が非常に高いため、リアルタイムビデオ転送などによく使用されます。例えば、ライブ配信やインターネット電話などです。データ損失が発生しても、ビデオがフレームアウトしても、それほど大きな問題ではありません。したがって、UDP プロトコルは、転送速度が求められ、エラーを許容できるデータ転送に引き続き使用されます。

43.4 TCP プロトコル

43.4.1 TCP プロトコルの概要

TCP も UDP と同様に、転送層のプロトコルですが、提供するサービスは大きく異なります。UDP が上層アプリケーションに提供するのは、信頼性のない、接続のないサービスですが、TCP は接続指向で信頼性のあるバイトストリーム転送サービスを提供します。TCP は、2 つのホストが接続関係を確立し、アプリケーションデータをデータストリームの形式で転送します。これは UDP プロトコルとは異なります。

- UDP が輸送するデータはメッセージの形式であり、各メッセージはネットワーク内で独立して転送され、UDP はメッセージを受信するたびにそれを上層アプリケーションに提供します。したがって、大量のデータに対しては、アプリケーション層での再構築が非常に面倒です。なぜなら、UDP メッセージがネットワークを通じて目的地のホストに到達する順序が異なるからです。

- 一方で、TCP はデータストリームの形式でデータを転送します。先に送出されたデータは、ネットワーク内で独立して転送されますが、これらのデータには密接な関連情報が含まれています。TCP プロトコルは、転送される各バイトに番号を付け、もちろん、2 つのホスト間のデータの番号は互いに独立しています。転送プロセスでは、送信側はデータの開始番号と長さを TCP パケットに含め、受信側はすべてのデータを番号に従って組み立て、確認応答を返します。すべてのデータが受信された後にのみ、データをアプリケーション層に提供します。

株式会社日昇テクノロジー

43.4.2 TCP の特徴

43.4.2.1 接続メカニズム

TCP は接続指向型プロトコルで、一方が他方にデータを送信する前に、双方間で接続を確立する必要があります。そうでなければ、データを送信することはできません。TCP 接続には双方の IP アドレスとポート番号が必要で、まるで電話をかける際に双方の電話番号を知っている必要があるのと同じです。具体的な接続については後述します。

43.4.2.2 確認と再送

完全な TCP 転送には、データの交換が必要です。受信側はデータを受信した後、正面から確認し、受信結果を送信側に報告する必要があります。送信側はデータを送信した後、受信側の確認を待つ必要があり、送信時にはタイマーが起動され、指定されたタイムアウト時間内に確認が受け取られない場合、送信失敗とみなされ再送されます。TCP は信頼できるトランスポート層を提供しますが、その基盤となる IP 層のサービスは接続なしで信頼性がありません。そのため、確認を通じて受信側がデータを確実に受け取ったことを知ります。しかし、データと確認の両方が失われる可能性があるため、TCP は送信時にタイムアウトメカニズム（タイマー）を設定し、この問題を解決します。タイムアウト時間が経過しても相手からの確認がない場合、データを再送します。

43.4.2.3 バッファリングメカニズム

送信側がデータを送信したい場合、アプリケーションのデータのサイズやタイプは予測できないため、TCP はこれらのデータを処理するためのバッファリングメカニズムを提供します。データ量が少ない場合、TCP はデータをバッファ領域に保存し、データ量が十分に大きくなった時点でデータを送信します。これにより、転送効率が向上し、ネットワーク内の通信量が削減されます。また、データが送信された後も、受信側がデータを正しく受信したことが保証されないため、バッファにデータを保持します。これは、受

株式会社日昇テクノロジー

信側からの確認を待ってからデータを削除する必要があるためです。同様に、受信側にも同じバッファリングメカニズムが必要です。なぜなら、ネットワークを介して転送されるデータパケットの到着時間が異なり、TCP プロトコルはこれらのデータパケットを完全なデータに再構成し、それをアプリケーション層に提供する必要があるからです。

43.4.2.4 全二重通信

TCP 接続が確立されると、二つのホストは等価となり、どちらのホストももう一方にデータを送信できます。データは双方向に流れるため、TCP は全二重プロトコルです。このメカニズムは、TCP プロトコルにおけるデータ転送を大いに便利にします。一般的に、TCP の確認は搭乗方法で実現されます。つまり、受信側は確認情報を逆方向のデータパケットに添付し、確認情報専用のパケットを要求する必要はありません。この搭乗メカニズムにより、ネットワーク内の通信量が削減されます。双方のホストが等価であるため、どちらかが接続を切断することができます。この場合、その方向のデータフローが切断されますが、もう一方の方向のデータは引き続き接続された状態です。これを半二重と呼びます。

43.4.2.5 フロー制御

前述のように、TCP 接続の両側にはバッファ領域が設定されています。受信側がデータを受け取ると、そのデータは受信バッファに格納されます。データが正常であると確認されると、受信側は送信側に確認を返し、関連するアプリケーション層にそのデータを提供します。しかし、データが到着した直後にアプリケーション層に提供されるわけではありません。実際には、受信側のアプリケーションが他のタスクに忙しく、データを処理するまでに長い時間がかかることがあります。このように、受信側がデータを比較的遅く処理する場合、送信側が多すぎるデータを速すぎる速度で送信すると、受信側の受信バッファがオーバーフローする可能性があります。

そのため、TCP はフロー制御サービスを提供し、送信側が受信側のバッファオーバーフローを引き起こす可能性を排除します。フロー制御は、送信側の送信速度が受信側のアプリケーションプログラムの読み

株式会社日昇テクノロジー

取り速度に一致する速度マッチングサービスです。TCP は、受信ウィンドウと呼ばれる変数を送信側が維持することでフロー制御を提供します。はい、受信ウィンドウです。これは、受信側がどれだけのデータを受信できるかを送信側に示すためのものです。受信側はこのウィンドウ値を TCP パケットのヘッダーのウィンドウフィールドに設定し、送信側に送信します。このウィンドウのサイズは、データの送信時に動的に調整されます。

ウィンドウが動的に調整されるということは、ウィンドウが 0 になる可能性があるということです。その場合、送信側は受信側にデータを送信できなくなります。この問題を解決するために、TCP プロトコルの仕様には要件があります。受信側の受信ウィンドウが 0 の場合、送信側は 1 バイトのデータセグメントのみを継続して送信します。これらのセグメントは受信側によって受信され、バッファが空になるまで続きます。そして、確認メッセージに非ゼロの受信ウィンドウ値が含まれます。フロー制御は双方の通信間の制御情報であり、非常に重要です。例えば、能力が異なる 2 つのホストが TCP 接続を確立した場合、一方のホストがデータを継続して送信し、受信側のホストが処理しきれない場合、この処理は最適ではありません。そのため、TCP ではスライディングウィンドウフロー制御メソッドを使用しています。これにより、受信側は自身の処理能力に基づいて受信可能なデータ量を決定し、送信側にどれだけのデータを送信できるかを知らせます。つまり、ウィンドウのサイズです。そして、送信側はできるだけ多くのデータを受信側に送信しようとします。つまり、受信側は送信側に「私はまだこれだけのデータを処理できます。それだけを送ってください。それ以上は処理できません」と伝えるのです。

43.4.2.6 エラー制御

確認と再送のほかに、TCP はチェックサムを使用してデータの有効性を検証します。ホストがデータを受信する際、重複したパケットを破棄し、順序が乱れたパケットを再構成します。また、あるデータセグメントが失われたと判断した場合、送信側に再送を要求します。そのため、TCP が上位層に提供するデータは、順序が整い、エラーのない完全なデータです。

株式会社日昇テクノロジー

43.4.2.7 混雑制御

混雑とは何か？データが大きなパイプ（例えば、高速なローカルエリアネットワーク）から小さなパイプ（例えば、遅い広域ネットワーク）に送信されるとき、混雑が発生します。複数の入力ストリームがルーターに到達し、そのルーターの出力ストリームがこれらの入力ストリームの合計よりも小さい場合にも混雑が発生します。これは、ネットワークの状況によるものです。あるホストが別のホストに大量のデータを送信し、その間のルーターのチャネルが小さく、このような大量のデータ流量を処理できない場合、混雑が発生します。これにより、受信側がタイムアウト内に受信を完了できず（受信側は大量のデータを処理する能力が十分にあるにもかかわらず）、送信側が再送を行い、これによりリンク上の混雑がさらに悪化し、遅延が発生します。送信側は、ネットワーク内で混雑が発生した場合に自身の送信速度を調整する自己適応メカニズムを実装する必要があります。この送信側の制御形態は混雑制御と呼ばれ、前述のフロー制御と非常に似ています。TCP が採用する措置も非常に似ており、送信速度の制限が含まれます。

43.5 ポート番号の概念

TCP 接続には、上層アプリケーション間の接続が含まれます。簡単に言えば、TCP 接続は異なるホストのアプリケーション接続です。トランスポート層と上位層のプロトコルは、ポート番号を通じて識別されます。IP プロトコルが IP アドレスを使用してホストを識別するのと同じように、ポート番号の範囲は 0～65535 です。これらのポートは、上層アプリケーションの異なるスレッドを識別します。1 つのホスト内には 1 つの IP アドレスしかない場合がありますが、複数のポート番号が存在することがあります。各ポート番号は異なるアプリケーションスレッドをリストします。IP アドレスを持つ 1 台のホストは、Web サービス、FTP サービス、SMTP サービスなど、多くのサービスを提供できます。これらのサービスは 1 つの IP アドレスで実現可能です。ホストが異なるネットワークサービスをどのように区別するかは明らかではありません。IP アドレスだけでは、ホストが提供するサービスを識別することはできません。これらのサービスはホスト上のアプリケーションスレッドであり、「IP アドレス+ポート番号」を使用し

株式会社日昇テクノロジー

てホストの異なるスレッドを区別します。

TCP プロトコルの一般的なポート番号には、21、53、80 などがあり、さらに多くのポートについてはリストで詳しく説明されています。中でも 80 番ポートは日常生活で最も一般的に見られるポート番号であり、HTTP サーバーがデフォルトで開放しているポートです。

ポート番号	プロトコル	説明
20/21	FTP	ファイル転送プロトコルは、ホスト間でファイルを共有することを可能にします。
23	Telnet	ユーザーがローカルコンピュータでリモートホストの作業を完了できるようにする、端末リモートログインです。
25	SMTP	シンプルメールトランスファープロトコルは、コンピュータがメールを送信または転送する際に次の目的地を見つけるのを助けます。
69	TFTP	トリビアルファイル転送プロトコル
80	HTTP	ハイパーテキストトランスファープロトコルは、ウェブブラウザ、ウェブクローラー、または他のツールを使用してクライアントが特定のポート（デフォルトは 80 番ポート）に HTTP リクエストをサーバーに送信し、応答するサーバーに HTML ファイルや画像などのリソースが保存されていれば、これらのデータをクライアントに返します。
110	POP3	ポストオフィスプロトコルバージョン 3 は、クライアントがサーバー上の電子メールをリモートで管理するのを支援するプロトコルです。

例として、http://www.firebbs.cn このウェブサイトを訪問する場合、ブラウザに http://www.firebbs.cn:80 と入力しても、80 番ポートにアクセスするため、同じページに入ります。しかし、http://www.firebbs.cn:90 と入力して 90 番ポートにアクセスしようとすると、アクセスは許可されません。

43.6 TCP セグメントの構造

TCP セグメントは、ヘッダ+データ領域から構成されます。TCP セグメントのヘッダを TCP ヘッダと呼びます。ヘッダの内容は非常に豊富で、各フィールドには異なる意味があります。オプションフィールドを除くと、一般的に TCP ヘッダは 20 バイトです。

各 TCP セグメントには、送信側と受信側のアプリケーションスレッドを見つけるためのソースホストと宛先ホストのポート番号が含まれています。これらの値と IP ヘッダのソース IP アドレスおよび宛先 IP

株式会社日昇テクノロジー

アドレスを合わせることで、一意の TCP 接続を特定できます。

株式会社日昇テクノロジー

シーケンス番号フィールドは、TCP 送信側から TCP 受信側へ送信されるデータバイトストリームを識別するために使用され、このセグメントの最初のデータバイトの位置をリストします。受信したデータ領域の長さに基づいて、セグメントの最後のデータのシーケンス番号を計算できます。TCP プロトコルは、送信または受信されるデータをバイト単位で番号付けするため、シーケンス番号を使用して各バイトをカウントすることで、これらのデータを簡単に管理できます。シーケンス番号は 32 ビットの符号なし整数です。

新しい接続が確立されると、TCP セグメントヘッダの SYN フラグが 1 になり、シーケンス番号フィールドには、このホストによってランダムに選択された初期シーケンス番号（ISN）が含まれます。このホストが送信するデータの最初のバイトのシーケンス番号は ISN+1 です。なぜなら、SYN フラグはシーケン

ス番号を占めるためです。ここでは、それについて簡単に理解するだけで十分ですが、後で詳しく説明します。

TCP プロトコルは、転送される各バイトに番号を付けるため、確認番号は受信側が次に期待するシーケンス番号を含みます。したがって、確認番号は、前回正常に受信したデータの最後のバイトのシーケンス番号+1 です。もちろん、ACK フラグが 1 の場合にのみ、確認番号フィールドが有効です。TCP はアプリケーション層に全二重サービスを提供します。これは、データが 2 つの方向で独立して転送できることを意味します。したがって、確認番号は通常、逆方向のデータ（つまり、受信側が送信側に転送するデータ）と同じセグメントにパッケージされます（つまり、荷担）。そのため、接続の各端は、転送データのシーケンス番号の正確性を保持する必要があります。

ヘッダ長フィールドは 4 ビットのスペースを占め、TCP セグメントヘッダの長さをバイト単位で示し、最大で 15*4=60 バイトのヘッダ長を記録できるため、TCP セグメントヘッダの最大長は 60 バイトです。フィールドの後には、使用されていない 6 ビットのスペースがあります。

さらに、TCP セグメントヘッダのフラグフィールドには 6 ビットのスペースがあり、いくつかの情報を示すために使用されます：

- URG：ヘッダの緊急ポインタフィールドフラグ。1 の場合、緊急ポインタフィールドが有効です。
- ACK：ヘッダの確認番号フィールドフラグ。1 の場合、確認番号フィールドが有効です。
- PSH：このフィールドが 1 に設定されている場合、受信側はこのセグメントをできるだけ早くアプリケーション層に渡すべきです。
- RST：TCP 接続を再確立します。- SYN：接続を開始するために同期シーケンス番号を使用します。
- FIN：接続を終了します。

TCP のフロー制御は、確認番号フィールドで示された値から始まる、宣言されたウィンドウサイズによって接続の各端から提供されます。ウィンドウサイズはバイト数で、受信側が期待するデータのシーケンス番号です。送信側はウィンドウサイズに基づいてデータ送信を調整し、フロー制御を実現します。ウィ

株式会社日昇テクノロジー

ンドウサイズは 16 ビットのフィールドで、最大ウィンドウサイズは 65535 バイトです。受信側が送信側に 0 のウィンドウサイズを伝えると、送信側のデータ送信が完全に停止します。

チェックサムは TCP セグメント全体に適用されます：TCP ヘッダと TCP データ領域。送信側によって計算され、記入され、受信側によって検証されます。

URG フラグが 1 に設定されている場合にのみ、緊急ポインタが有効です。緊急ポインタは正のオフセットであり、シーケンス番号フィールドの値に加算されると、緊急データの最後のバイトのシーケンス番号が示されます。簡単に言うと、TCP セグメントの緊急データは、シーケンス番号フィールドから始まり、緊急ポインタの値でオフセットされた場所で終了します。

オプションフィールドはあまり使用されませんので、現時点では気にする必要はありません。

43.7 TCP 接続の確立

TCP は接続指向のプロトコルであり、どちらかの方向からもう一方にデータを送信する前に、双方の間に接続を確立する必要があります。一般に「ハンドシェイク」と呼ばれています。ネットワークを学ぶ前に、「3 ウェイハンドシェイク」や「4 ウェイウェーブ」などの用語を聞いたことがあるかもしれませんが、それらはどのようなものでしょうか？このセクションでは、TCP 接続がどのように確立され、通信が終了した後にどのように終了するかについて詳しく説明します。

「3 ウェイハンドシェイク」での接続確立：

まず、接続の確立プロセスはクライアントから開始され、サーバーは常にクライアントの接続を待っています。そのプロセスは一般に以下のように進みます：

ステップ 1：クライアントの TCP は、サーバーの TCP に特別な TCP セグメントを送信します。このセグメントにはアプリケーション層のデータは含まれていませんが、セグメントのヘッダの SYN フラグが 1 に設定されています。したがって、この特別なセグメントは SYN セグメント（握手リクエストセグメントとも呼ばれます）と呼ばれます。さらに、クライアントはランダムに初期シーケンス番号（ISN、例として A としましょう）を選択し、その番号を SYN セグメントのシーケンス番号

株式会社日昇テクノロジー

フィールドに配置します。しかし、SYN セグメントの ACK フラグは 0 であり、この時点では確認番号フィールドは無効です。このセグメントは IP データグラムにパッケージされ、目的のサーバーに送信されます。

第二ステップ：サーバーがクライアントから送信された SYN パケットを受け取ると、クライアントがハンドシェイクを要求していることがわかります。サーバーは SYN パケットから対応する情報を抽出し、その TCP 接続に TCP バッファと変数を割り当て、クライアントの TCP に接続許可パケット（ハンドシェイク応答パケット）を送信します。このパケットもまた、アプリケーション層のデータを含んでいませんが、パケットのヘッダには 3 つの重要な情報が含まれています。

1　SYN と ACK のフラグは両方とも 1 に設定されます。

2　TCP パケットヘッダの確認番号フィールドを A+1 に設定します（この A（ISN）はハンドシェイク要求パケットから得られます）。

3. サーバーは自身の初期シーケンス番号（ISN、注意：この ISN はサーバー側の ISN、B と仮定）をランダムに選択し、それを TCP パケットヘッダのシーケンス番号フィールドに配置します。

この接続許可パケットは実際には、「あなたの接続設立要求を受け取りました、初期シーケンス番号は A です、私はこの TCP 接続を確立することに同意します、私自身の初期シーケンス番号は B です」という意味をリストしています。この接続許可パケットは時々SYN ACK パケット（SYN ACK segment）と呼ばれ、ACK フラグが 1 であるため、TCP パケットヘッダのウィンドウサイズフィールドは有効です。

株式会社日昇テクノロジー

ユーザ側
サーバー側
SYN
seq:A ack:無効
SYN+ACK
seq:B ack:A+1
ACK
seq:A+1 ack:B+1

第三ステップ：クライアントがサーバーからのハンドシェイク応答メッセージを受け取ると、ACK フラグが設定されます。この時点で、クライアントの TCP セグメントの ACK フラグは 1 に設定され、SYN フラグは接続がすでに確立しているため 0 に設定されます。同時に、クライアントはこの TCP 接続にバッファと変数を割り当て、また、クライアントは応答メッセージセグメントを返す必要があります。このメッセージは、サーバーの応答メッセージセグメントに対する応答で、TCP メッセージセグメントヘッダの確認番号フィールドを B+1 に設定し、同時にサーバーにウィンドウサイズを通知します。

ヒント：ハンドシェイクの第三ステップでは、メッセージセグメントのデータ領域にクライアントからサ

株式会社日昇テクノロジー

ーバーへのデータを含めることができます。

ハ ンドシェイクが完了すると、クライアントとサーバーは接続を確立し、両方とも相手のウィンドウサイズ、シーケンス番号などの情報を得ます。TCP セグメントを転送する際、各 TCP セグメントヘッダの SYN フラグは 0 に設定されます。これは、接続を開始し、シーケンス番号を同期するためだけに使用されます。

43.8 TCP 接続の終了

接続を確立するには三回のハンドシェイクが必要ですが、接続を終了するには四回の振り手（一部の書籍では「四回のハンドシェイク」とも呼ばれます）が必要です。これは TCP の特性によるもので、TCP 接続は全二重接続サービスであるため、各方向の接続は個別に閉じる必要があります。一方がデータ送信タスクを完了すると、FIN メッセージセグメント（接続終了要求とも呼べます、実際には FIN フラグが 1 に設定されています）を送信して、その方向の接続を終了します。他方が FIN メッセージセグメントを受け取ると、アプリケーション層に対して相手がその方向の接続を終了したことを通知する必要があります。FIN メッセージセグメントの送信は通常、アプリケーション層が閉じる結果となります。

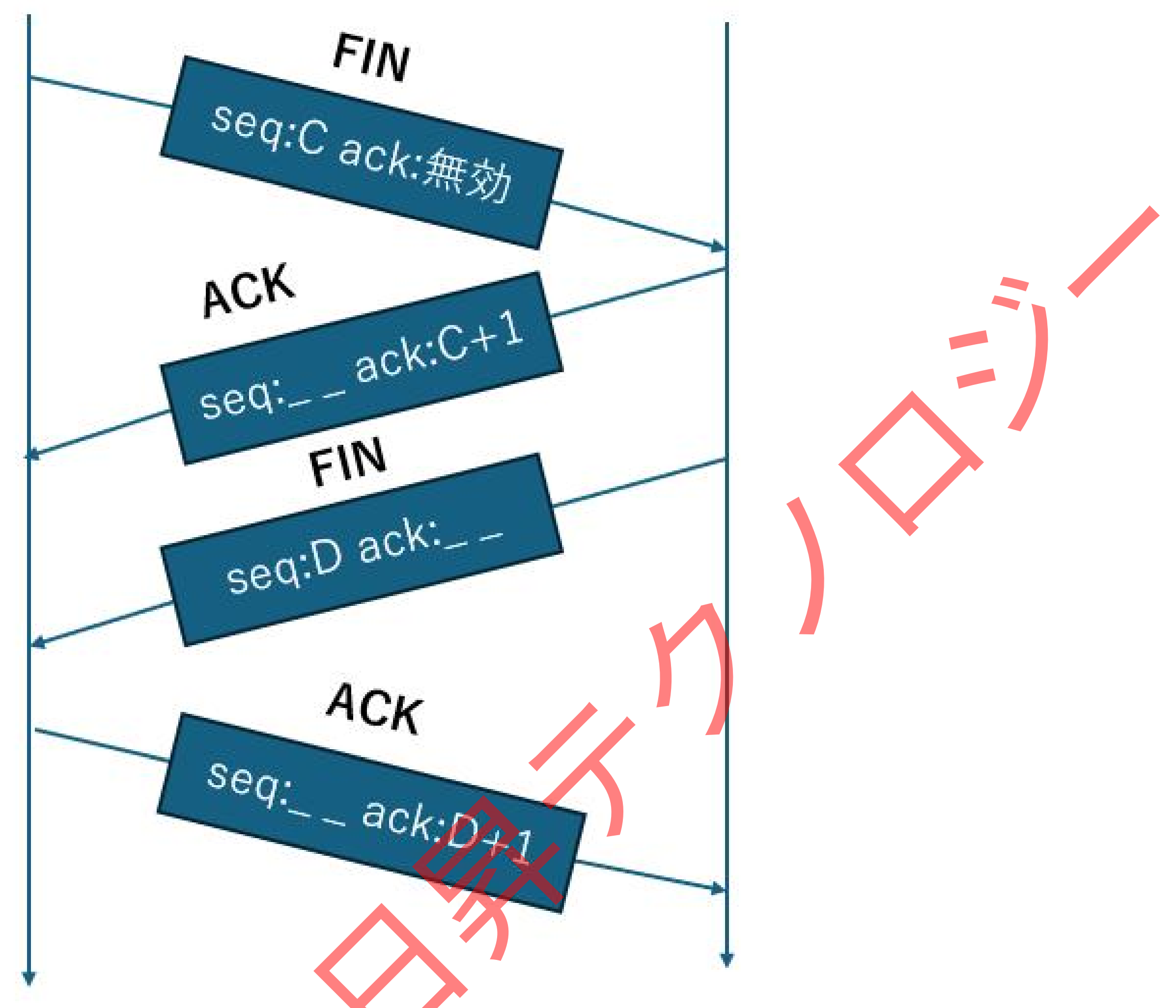
注意：クライアントが FIN メッセージセグメントを送信するということは、その方向にデータが流れないことを意味します。TCP 接続は、FIN を送信した後もデータを受信することができますが、実際のアプリケーションでは、非常に少数の TCP アプリケーションがこれを行います。

「四回の振り手」による接続終了の具体的な手順は以下の通りです：

第一ステップ：クライアントは FIN メッセージセグメントを発行して接続を主動的に閉じます。この時点で、メッセージセグメントの FIN フラグは 1 に設定され、シーケンス番号は C と仮定します。通常、ACK フラグも 1 に設定されますが、確認番号フィールドは無効です。

第二ステップ：サーバーがこの FIN メッセージセグメントを受け取ると、ACK メッセージセグメント（このメッセージセグメントは接続終了応答です）を返します。

株式会社日昇テクノロジー



株式会社日昇テクノロジー

確認番号は受信した番号に 1 を加えたもの（C+1）で、SYN と同様に、FIN は一つの番号を占有します。この時点で、クライアントからサーバへの接続が切断されます。

第三ステップ：サーバはアプリケーションに対して、このクライアントとの接続を閉じるように要求します。その後、サーバは FIN メッセージセグメント（このメッセージセグメントは、サーバがクライアントに対して接続を終了するように要求するもの）を送信します。この時点で、仮にシーケンス番号を D とし、ACK フラグは 1 に設定されますが、確認番号フィールドは無効です。

第四ステップ：クライアントは ACK メッセージセグメントを返して接続終了の要求を確認します。ACK フラグは 1 に設定され、確認番号は受信した番号に 1 を加えたもの（D+1）に設定されます。この時点で、サーバからクライアントへの接続が切断されます。

43.9 TCP の状態

TCP プロトコルは、接続時に受信したメッセージの種類に応じて、それぞれ異なる動作を行います。また、各状態の関係も処理する必要があります。例えば、ハンドシェイクメッセージを受信した時、タイムアウトした時、ユーザが主動的に接続を閉じた時など、それぞれ異なる状態で異なる処理を行う必要があります。

TCP プロトコルの状態は以下の通りです：

- LISTENING 状態：ある種のサービスを提供し、遠隔の TCP ポートの接続要求を待ち受けます。提供するサービスが接続されていない場合、LISTENING 状態になり、ポートは開放され、接続を待ちます。
- SYN_SENT（クライアントの状態）：クライアントは connect()関数を呼び出し、SYN 要求を送信して接続を確立します。接続要求を送信した後、マッチする接続要求を待ちます。この時点での状態は SYN_SENT です。
- SYN_RECEIVED（サーバの状態）：接続要求を受信し、送信した後、相手方からの接続要求の確認を待ちます。サーバがクライアントから送信された同期信号を受信すると、ACK と SYN のフラグを 1 に設定してクライアントに送信します。この時点で、サーバは SYN_RCVD 状態になります。接続が成功すれば ESTABLISHED 状態になります。通常、SYN_RCVD 状態は非常に短いです。
- ESTABLISHED 状態：この状態は、安定した接続状態を示しています。TCP プロトコルの両端のホストは、この状態になります。それらは互いに窓口の大きさ、シーケンス番号、最大セグメントなどの情報を知っています。
- FIN_WAIT_1 および FIN_WAIT_2 状態：この状態は、一方向に接続を終了する要求がある場合に一般的になります。その後、ホストは相手方からの応答を待ちます。相手方が応答を生成すると、ホストの状態は FIN_WAIT_2 に移行します。この時点で、ホストから相手方への TCP 接続が切断されますが、相手方からホストへの接続はまだ存在します。ここで注意すべき点は、ホストが

株式会社日昇テクノロジー

FIN_WAIT_2 状態にある場合、ホストはすでに FIN メッセージセグメントを送信し、相手方もそれを確認していることを意味します。ホストが半閉鎖状態を実行している場合を除き、ホストは相手方のホストがアプリケーション層で接続を閉じるのを待つ必要があります。なぜなら、相手方はすでに FIN メッセージセグメントを受信したことを認識しており、ホストから FIN を送信して、相手方からホストへの接続を閉じる必要があるからです。もう一方の端末がこの閉鎖を完了すると、ホストは FIN_WAIT_2 状態から TIME_WAIT 状態に移行します。そうでなければ、これはホストが恒久的に FIN_WAIT_2 状態を保持することを意味し、もう一方のホストも CLOSE_WAIT 状態になり、アプリケーション層が閉鎖を決定するまでこの状態を維持します。

• CLOSE-WAIT 状態：ローカルユーザからの接続中断要求を待っています。受動的に接続を閉じる端末の TCP は、FIN を受け取った後、ACK を送信して FIN 要求に応答します（その受信は、ファイル終了記号として上層のアプリケーションに伝達されます）、そして CLOSE_WAIT 状態に入ります。

• TIME_WAIT 状態：TIME_WAIT 状態は、2MSL 待機状態とも呼ばれます。各具体的な TCP 接続の実装は、TCP メッセージセグメントの最大生存時間 MSL（Maximum Segment Lifetime）を選択する必要があります。これは、IP データグラムの TTL フィールドのようなもので、メッセージがネットワーク内で生存する時間を示しています。これは、任意のメッセージセグメントがネットワーク内で破棄される前の最長時間で、この時間は有限です。なぜ待つ必要があるのでしょうか？、IP データグラムは信頼できないことを知っていますが、TCP メッセージセグメントは IP データグラムに封入されています。TCP プロトコルは、送信した ACK メッセージセグメントが正確に相手方に受信されることを保証しなければなりません。そのため、この状態のホストは、この状態に最長時間である 2 倍の MSL を保持する必要があります。これは、最後の ACK が失われるのを防ぐためです。なぜなら、TCP プロトコルはデータが正確に目的地に送られることを保証しなければならないからです。

理論をたくさん説明しましたが、以下では図を用いて TCP プロトコルが接続を確立し、接続を終了する際のすべての状態遷移処理を詳しく説明します。具体的には以下の通りです：

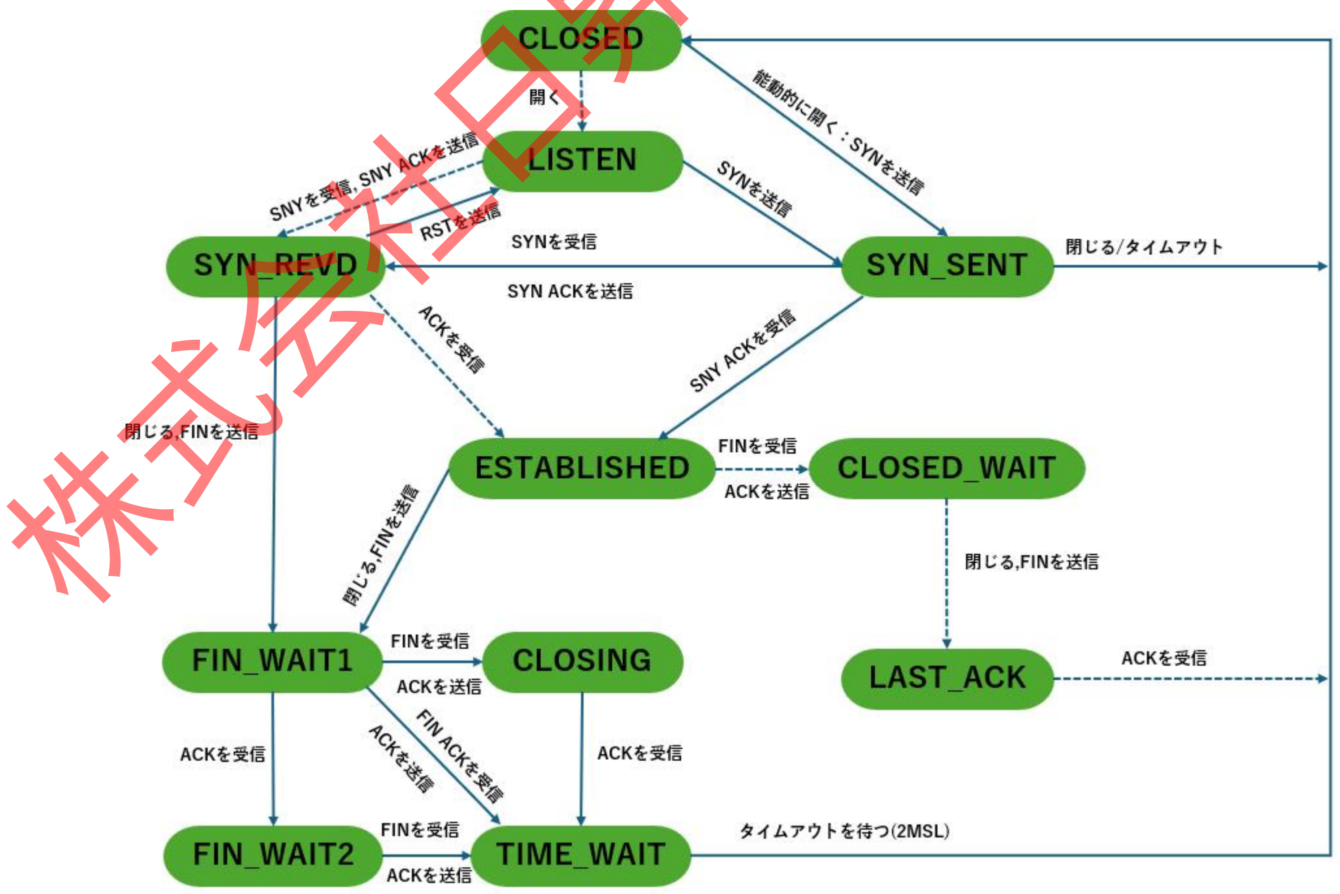
株式会社日昇テクノロジー

図中の補足説明（非常に重要なので、しっかりと覚えておいてください）：

- 破線：サーバーの状態遷移を示します。
- 実線：クライアントの状態遷移を示します。
- 図のすべての「閉じる」、「開く」はアプリケーションが主動的に処理します。
- 図のすべての「タイムアウト」はカーネルがタイムアウトを処理します。

三つのハンドシェイクのプロセス：

- 図(7)：サーバーのアプリケーションが主動的にサーバーをリッスン状態にし、クライアントからの接続要求を待ちます。
- 図(1)：まず、クライアントのアプリケーションが接続を主動的に開始し、SYN メッセージセグメントをサーバーに送信します。送信後、クライアントは SYN_SENT 状態になり、サーバーからの SYN ACK メッセージセグメントの確認を待ちます。指定されたタイムアウト時間内にサーバーから応答確認がない場合、クライアントは接続を閉じます。



株式会社日昇テクノロジー

• 図(8)：リッスン状態のサーバーがクライアントからの接続要求（SYN メッセージセグメント）を受け取ると、サーバーは SYN ACK メッセージセグメントを返してクライアントの応答を確認し、サーバーは SYN_RCVD 状態になります。

• 図(1)：クライアントがサーバーからの SYN ACK メッセージセグメントを受け取った場合、クライアントは ESTABLISHED（安定した接続状態）になり、サーバーに ACK メッセージセグメントを送信します。

• 図(9)：同時に、サーバーはクライアントからの ACK メッセージセグメントを受け取り、接続が成功したことを示し、ESTABLISHED（安定した接続状態）になります。これが接続を確立するための三つのハンドシェイクのプロセスです。

四つの振り手のプロセス：

• 図(3)：一般的には、クライアントが主動的に FIN メッセージセグメントを送信して接続を終了します。この時点で、クライアントは ESTABLISHED（安定した接続状態）から FIN_WAIT_1 状態に遷移し、サーバーからの応答確認を待ちます。

• 図(10)：サーバーが FIN メッセージセグメントを受け取り、クライアントが接続を終了することを知ると、サーバーは ACK メッセージセグメントをクライアントに返して接続終了を確認し、サーバーの状態は安定状態から CLOSE_WAIT（接続終了待ち状態）に遷移します。

• 図(4)：クライアントが確認メッセージセグメントを受け取った後、FIN_WAIT_2 状態に遷移し、サーバーからの主動的な接続終了要求を待ちます。この時点で、{クライアント-> サーバー}方向の接続はすでに切断されています。

• 図(11)：一般的には、クライアントが接続を終了した後、サーバーも{サーバー-> クライアント}方向の接続を終了します。そのため、サーバーの原因プログラムはこの方向の接続を主動的に閉じ、クライアントに FIN メッセージセグメントを送信します。

• 図(5)：FIN_WAIT_2 状態のクライアントが FIN メッセージセグメントを受け取ると、サーバーに

株式会社日昇テクノロジー

ACK メッセージセグメントを送信します。

• 図(12)：サーバーが ACK メッセージセグメントを受け取ると、直接閉じます。この時点で、{サーバー-> クライアント}方向の接続はすでに終了し、CLOSED 状態になります。

• 図(6)：クライアントはさらに 2MSL 待ちます。これは、ACK メッセージセグメントがサーバーに受け取られなかった場合の対策です。これが四つの振り手の全過程です。

注意：(13)(14)(15)のこれらの状態はいくつかの特殊な状態で、一時的にこれらを説明しません。全体的に見て、それらはすべて同じです。

TCP プロトコルの簡単な説明はここまでです。上記で説明した多くの内容は、実際には非常に非常に簡単で、非常に非常に基本的なものです。具体的には、TCP プロトコルのさらに多くの特性を、対応する専門書籍を参照して自分で確認する必要があります。

第 44 章 ソケット

皆さんがメッセージキューの章で言及した一節を覚えているかどうかはわかりません：

Linux のプロセス間通信手段は基本的に Unix プラットフォームのプロセス間通信手段を継承しています。そして、Unix の発展に大きな貢献をした AT&T のベル研究所と BSD（カリフォルニア大学バークレー校のバークレーソフトウェアディストリビューションセンター）は、プロセス間通信における重点が異なります。前者は Unix の初期のプロセス間通信手段をシステム的に改良し拡張し、"system V IPC"を形成しました。通信プロセスは単一のコンピュータ内に限定されています（同一デバイスの異なるプロセス間通信）。一方、後者はこの制限を飛び越え、ソケット（Socket）を基にしたプロセス間通信メカニズム（主に異なるデバイス間のプロセス間通信に使用される）を形成しました。Linux は両者を継承しているため、Linux が最も成功していると言えます。"system V IPC"を持ちながら、Socket もサポートしています。

それは明らかです、前の記事では、単一のコンピュータ内のプロセス間通信を説明しました。すべてのメカニズムは一台のコンピュータシステムの共有リソースに依存しています。ここでのリソースは、ファイ

株式会社日昇テクノロジー

ルシステム空間、共有物理メモリ、またはメッセージキューなどがありますが、同一のマシン上で実行されるプロセスだけがそれらを使用することができます。これは明らかに求めているものを満たしていません。このデバイスが他のデバイスと通信することを望むかもしれません。次に、異なるデバイス間のプロセス間通信について説明します。これはソケット（Socket）を基にしたプロセス間通信メカニズムです。もちろん、Socket は一つのデバイス上のプロセス間通信もサポートしています。それは異なるデバイス間だけではありません。

前のセクションでも、ネットワーク中の UDP と TCP プロトコルについて簡単に説明しました。今日は、ソケットを通じて通信を行います。

44.1 ソケットの概要

ソケット（socket）は一種の通信メカニズムで、このメカニズムを利用すると、クライアント<->サーバモデルの通信は、ローカルデバイス上で行うことも、ネットワークを介して行うこともできます。ソケットの作成と使用はパイプとは異なります。なぜなら、ソケットは明確にクライアントとサーバを区別し、さらに、ソケットメカニズムは複数のクライアントを一つのサーバに接続することができます。

Socket という英語の原意は「穴」または「コンセント」を意味しますが、ネットワークプログラミングでは、通常、「ソケット」と呼ばれます。現在のネットワークの主流のプログラム設計はすべて Socket を使用してプログラミングされています。なぜなら、それは簡単で使いやすく、また、それは一つの標準（BSD Socket）であり、異なるプラットフォームに容易に移植することができます。例えば、あなたの一つのアプリケーションが Socket プログラミングに基づいている場合、それは任何の BSD Socket 標準を実装したプラットフォームに移植することができます。例えば、LwIP は BSD Socket と互換性があります。また、Windows は Socket のソケットインターフェースを実装しています。さらに、国産のオペレーティングシステムである RT-Thread も BSD Socket 標準の Socket インターフェースを実装しています。

Socket では、ネットワークの一つの接続を記録するために一つのソケットを使用します。ソケットは整数で、ファイルを操作するのと同じように、ファイルディスクリプタを利用して、開く、読む、書く、閉じ

株式会社日昇テクノロジー

るなどの操作を行うことができます。同様に、ネットワークでは、Socket ソケットに対してこのような操作を行うことができます。例えば、ネットワークの接続を開始する、接続ホストから送信されたデータを読み取る、接続ホストにデータを送信する、接続を終了するなどの操作です。

ソケットディスクリプタについて理解しましょう。それはファイルディスクリプタと非常に似ています。実際には、それはただの整数です。ソケット API は最初、UNIX オペレーティングシステムの一部として開発されました。そのため、ソケット API はシステムの他の I/O デバイスと統合されています。アプリケーションがネットワーク通信のためにソケット（socket）を作成すると、オペレーティングシステムは整数を返してこのソケットを識別します。その後、アプリケーションはそのディスクリプタを引数として渡し、Socket API インターフェースの関数を呼び出して特定の操作（例えば、ネットワークを介してデータを送信するか、入力データを受信する）を完了します。

次に、Linux システムのソケット関連の関数を説明しますが、注意してください。ネットワークプログラミングでよく使われるヘッダーファイルを含める必要があります：

```
#include <sys/types.h>

#include <sys/socket.h>
```

44.2 socket()

関数のプロトタイプ：

```
int socket(int domain, int type, int protocol);
```

socket() 関数は、ソケットディスクリプタ（socket descriptor）を作成するためのもので、これは一つのソケットを一意に識別します。このソケットディスクリプタはファイルディスクリプタと同じで、後続の操作すべてで使用されます。それをパラメータとして、それを通じていくつかの読み書き操作を行うことができます。

ソケットを作成する際には、異なるパラメータを指定して異なるソケットディスクリプタを作成することもできます。socket 関数の 3 つのパラメータは次の通りです：

株式会社日昇テクノロジー

1. domain：パラメータ domain は、そのソケットが使用するプロトコルファミリを示します。Linux システムでは、多くのプロトコルファミリがサポートされています。TCP/IP プロトコルについては、AF_INET を選択すれば十分です。もちろん、あなたの IP プロトコルのバージョンが IPv6 をサポートしている場合は、AF_INET6 を選択することもできます。選択可能なプロトコルファミリは次の通りです：

- AF_UNIX, AF_LOCAL：ローカル通信
- AF_INET ：IPv4
- AF_INET6 ：IPv6
- AF_IPX ：IPX - Novell プロトコル
- AF_NETLINK ：カーネルユーザインターフェースデバイス
- AF_X25 ：ITU-T X.25 / ISO-8208 プロトコル
- AF_AX25 ：アマチュア無線 AX.25 プロトコル
- AF_ATMPVC ：原始 ATM PVC へのアクセス
- AF_APPLETALK ：AppleTalk
- AF_PACKET ：下層データパケットインターフェース
- AF_ALG ：カーネル暗号化 API の AF_ALG インターフェース

2. type：パラメータ type は、ソケットが使用するサービスタイプを指定します。可能なタイプは次のいくつかです：

- SOCK_STREAM：信頼性のある（つまり、データが正確に相手方に送信されることを保証する）接続指向の Socket サービスを提供します。これは主に資料（例えばファイル）の転送に多く使用されます。例えば、TCP プロトコル。
- SOCK_DGRAM：保証のないメッセージ指向の Socket サービスを提供します。これは主にネットワーク上でブロードキャスト情報を送信するために使用されます。例えば、UDP プロトコルは、接続の

株式会社日昇テクノロジー

ない信頼性のないデータグラム配信サービスを提供します。

• SOCK_SEQPACKET：固定最大長のデータグラムに対して順序付けられた、信頼性のある、双方向接続のデータ転送パスを提供します。

• SOCK_RAW：原始ソケットをリストし、アプリケーションがネットワーク層の原始データパケットにアクセスすることを許可します。このソケットはあまり使われませんので、一時的に無視してください。

• SOCK_RDM：順序を保証しない信頼性のあるデータグラム層を提供します。

3.protocol：パラメータ protocol は、ソケットが使用するプロトコルを指定します。IPv4 では、TCP プロトコルだけが SOCK_STREAM という信頼性のあるサービスを提供し、UDP プロトコルだけが SOCK_DGRAM サービスを提供します。これら 2 つのプロトコルについては、protocol の値は 0 であり、protocol が 0 の場合、自動的に type タイプに対応するデフォルトのプロトコルを選択します。

ソケットの作成が成功した場合、この関数は int 型の値、つまりソケットディスクリプタを返します。この値は 0 以上です。一方、ソケットの作成に失敗した場合は-1 を返します。

44.3 bind()

関数の原型は以下の通りです：

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

ソケット通信において、ソケットはユーザープログラムとカーネルが情報を交換するための枢要であり、それ自体に多くの情報を持っているわけではなく、ネットワークプロトコルのアドレスやポート番号などの情報も持っていません。通信を行う際には、ソケットを IP アドレスやポート番号に関連付ける必要があり、これがバインドのプロセスです。

bind()関数は、IP アドレスやポート番号をソケットにバインドするために使用されます。多くの場合、カーネルが自動的に IP アドレスやポート番号をバインドしてくれますが、時にはユーザー自身がこのバインドプロセスを行う必要があります。これは、特にサーバープロセスがクライアントの接続を待つために

株式会社日昇テクノロジー

知られているアドレスとポートにバインドする必要がある場合に顕著です。サーバー側ではこのバインド操作が必須であり、クライアント側では必須ではありません。なぜなら、カーネルが適切な IP アドレスとポート番号を自動的に選択してくれるからです。

注意：bind()関数を必ずしも呼び出す必要はありません。ユーザープロセスが特定のアドレスやポートに関連付けたい場合のみ、この関数を呼び出す必要があります。ユーザープロセスにそのような要求がない場合は、カーネルの自動アドレス選択メカニズムに頼ることができます。

パラメータ：

- sockfd：sockfd は socket()関数によって返されるソケット記述子です。
- my_addr：my_addr はソケットアドレス構造体へのポインタです。
- addrlen：addrlen は addr が指すアドレス構造体のバイト長を指定します。

bind()関数が成功すると 0 を返し、エラーが発生した場合は-1 を返します。

sockaddr 構造体の内容は以下の通りです：

sockaddr 構造体：

```
struct sockaddr {
sa_family_t sa_family;
char sa_data[14];
}
```

この構造体を見ると、ユーザーが入力する情報はほとんどありません。実際、IP アドレスやポート番号などの情報はすべて sa_data の 14 バイトに格納されていますが、この構造体はユーザー操作にはあまり友好的ではありません。通常、sockaddr_in 構造体を使用します。sockaddr_in と sockaddr は同じサイズの並列構造で、sockaddr_in 構造体へのポインタは sockaddr 構造体にも指すことができ、置き換えることができます。さらに、sockaddr_in 構造体はユーザーにとってより使いやすいです。

株式会社日昇テクノロジー

sockaddr_in 構造体：

```
struct sockaddr_in {
short int sin_family; /* プロトコルファミリ */
unsigned short int sin_port; /* ポート番号 */
struct in_addr sin_addr; /* IP アドレス */
unsigned char sin_zero[8]; /* sockaddr と sockaddr_in のサイズを同じに保つための予約バイト */
};
```

sockaddr_in 構造体の最初のフィールドは sockaddr 構造体と一致しており、残りのフィールドは sa_data の 14 バイトの情報を新たに定義したメンバ変数に再定義しています。sin_port フィールドは入力が必要なポート番号、sin_addr フィールドは入力が必要な IP アドレス情報です。残りの sin_zero 領域の 8 バイトは使用されていません。

簡単な使用例を挙げます：

```
struct sockaddr_in server;
bzero(&server, sizeof(server));
// IP, PORT を割り当てる
server.sin_family = AF_INET;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(6666);
// 新しく作成されたソケットを指定された IP にバインドし、確認する
bind(sockfd, (struct sockaddr*)&server, sizeof(server));
```

株式会社日昇テクノロジー

44.4 connect()

関数の原型は以下の通りです：

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

この connect()関数はクライアントで使用され、sockfd をリモート IP アドレス、ポート番号にバインドします。TCP クライアントでは、この関数を呼び出すとハンドシェイクプロセスが発生し（TCP 接続要求が送信され）、最終的に TCP 接続が確立します。UDP プロトコルにおいては、この関数を呼び出すと sockfd にリモート IP アドレスとポート番号が記録されるだけで、データは送信されません。

関数が成功すると 0 を返し、失敗すると-1 を返し、エラーの原因は errno に格納されます。

connect()関数はソケット接続操作で、TCP プロトコルにおいては、connect()関数の操作が成功すると、対応するソケットがリモートホストと接続され、データの送受信が可能になります。

UDP プロトコルにおいては、接続の概念がありません。ここでは、リモートホストの IP アドレスとポート番号を記録すると説明できます。UDP プロトコルでは、connect()関数の呼び出し成功後、sendto()関数でデータグラムを送信する際に宛先のアドレスやポートを指定する必要がなくなります。なぜなら、この時点でリモートホストの IP アドレスとポート番号が記録されているからです。UDP プロトコルでは、同じソケットに対して複数回の connect()操作を行うことができますが、TCP プロトコルではできません。TCP は一度の connect 操作のみが可能です。

44.5 listen()

関数は、TCP サーバープロセスでのみ使用でき、サーバープロセスがリスニング状態に入り、クライアントからの接続要求を待ちます。listen()関数は通常、bind()関数の後に呼び出され、accept()関数の前に呼び出されます。その関数原型は以下の通りです：

```
int listen(int s, int backlog);
```

パラメータ：

- sockfd: sockfd は socket()関数によって返されるソケットディスクリプタです。

株式会社日昇テクノロジー

- backlog: sockfd の接続待ちキューが到達可能な最大値を記述するために使用されます。サーバープロセスがクライアントの接続要求を処理している間に、他のクライアントからの接続要求が存在する可能性があります。TCP 接続はプロセスであり、同時に接続を試みるユーザーが多すぎると、サーバープロセスがすべての接続要求を迅速に処理できないため、他のクライアントの接続を直接破棄するわけにはいきません。したがって、カーネルは自身のプロセス空間内でキューを維持し、これらの接続要求をキューに入れます。サーバープロセスは、先着順でこれらの接続要求を処理します。このようなキューをカーネルが任意に大きくすることはできませんので、最大サイズの上限が必要です。この backlog は、キューの上限としてこの数値を使用するようにカーネルに指示します。クライアントの接続要求が到着し、キューが満杯の場合、クライアントは接続失敗のエラーを受け取る可能性があり、この要求は処理されずに破棄されます。

44.6 accept()

関数のプロトタイプ:

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

TCP クライアントがサーバーに正常に接続できるようにするために、サーバーは以下の手順に従って処理する必要があります。

1. socket()関数を呼び出して対応するソケットタイプを作成します。

2. bind()関数を呼び出してソケットをローカルのポートアドレスにバインドします。

3. listen()関数を呼び出してサーバープロセスをリスニング状態にし、クライアントの接続要求を待ちます。

4. accept()関数を呼び出して来た接続要求を処理します。

accept()関数は TCP サーバーで使用され、遠隔ホストからの接続要求を待ち、新しい TCP 接続を確立します。この関数を呼び出す前に、listen()関数を呼び出してサーバーをリスニング状態にする必要があります。未完成の接続ソケットがキューになく、ソケットがノンブロッキングモードとしてマークされてい

株式会社日昇テクノロジー

ない場合、accept()関数の呼び出しは、遠隔ホストとの TCP 接続が確立されるまでアプリケーションをブロックします。ソケットがノンブロッキング式としてマークされており、キューに未完成の接続ソケットがない場合、accept()関数の呼び出しは直ちに EAGAIN を返します。

したがって、accept()関数は接続要求を処理するために使用されます。未完成の接続キューから最初の接続要求を取り出し、パラメータ s と同じ属性の接続ソケットを作成し、このソケットにファイルディスクリプ accept()関数は TCP サーバーで使用され、遠端のホストからの接続要求を待ち、新しい TCP 接続を確立します。この関数を呼ぶ前に、サーバーが listen()関数を呼び出してリスニング状態になっている必要があります。キュー内に未完了の接続ソケットがなく、ソケットが非ブロックモードでマークされていない場合、accept()関数の呼び出しは、遠端のホストと TCP 接続が確立されるまでアプリケーションをブロックします。もしソケットが非ブロック式であり、キュー内に未完了の接続ソケットがない場合、accept()関数は直ちに EAGAIN を返します。

従って、accept()関数は接続要求を処理するために使われます。未完了の接続キューから最初の接続要求を取り出し、パラメータ s と同じ属性を持つ接続ソケットを作成し、このソケットにファイルディスクリプタを割り当て、このディスクリプタで返します。新しく作成されたディスクリプタはリスニング状態ではなくなり、元のソケット s はこの呼び出しの影響を受けずにリスニング状態のままです。s は socket()関数によって作成されたものですが、接続処理時に accept()関数は別のソケットを作成します。

パラメータ addr は接続されたクライアントの IP アドレスとポート番号を返すために使われ、パラメータ addrlen は addr が指すアドレス構造のバイト長を返すために使われます。もしクライアントの IP アドレスとポート番号に興味がなければ、addr と addrlen を両方とも NULL に設定できます。

接続が成功すれば、ソケットディスクリプタ（非負の値）を返し、エラーが発生した場合は-1 を返します。

ヒント: accept()の接続が成功すると、カーネルによって自動的に生成された新しいディスクリプタが返され、これはクライアントとの TCP 接続をリストします。サーバーは通常、1 つのリスニングソケットの

株式会社日昇テクノロジー

みを作成し、そのサーバーの生存期間中ずっと存在します。サーバープロセスによって受け入れられた各クライアント接続に対して、カーネルは接続済みソケットを作成します。

44.7 read()

クライアントとサーバーが TCP 接続を確立した後、sockfd ソケットディスクリプタを通じてデータを送受信できます。これはファイルを読み書きする操作とほぼ同じです。ネットワークからのデータを受信する関数には read()、recv()、recvfrom()などがあります。

関数のプロトタイプ:

```
ssize_t read(int fd, void *buf, size_t count);

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);
```

ps: ssize_t は signed size_t 型をリストします。

read()はディスクリプタ fd（ファイルディスクリプタでもソケットディスクリプタでも良いが、ここではソケットについて説明しています）から count バイトのデータを読み込み、buf から始まるバッファに格納します。read()関数の呼び出しが成功すると、読み込まれたバイト数が返されます。この返り値はファイルの残りのバイト数に制限されます。返り値が指定されたバイト数より小さい場合でも必ずしもエラーを意味するわけではありません。これは、利用可能なバイト数が指定されたバイト数より小さい場合（ファイルの終わりに近づいている、パイプや端末から読み込んでいる、read()関数がシグナルによって中断された等）に起こり得ます。エラーが発生すると-1 を返し、errno を設定します。read を呼び出す前にファイルの終わりに達していれば、この回の read は 0 を返します。

パラメータ:

- fd: ソケットプログラミングではソケットディスクリプタを指定します。

- buf: データを格納するアドレスを指定します。

株式会社日昇テクノロジー

- count: 読み取るバイト数を指定し、読み取ったデータをバッファ buf に保存します。

エラーコード:

- EINTR: データを読み取る前にシグナルによって中断されました。

- EAGAIN: O_NONBLOCK フラグを使用してノンブロッキング入出力を指定しましたが、現在読み取り可能なデータがありません。

- EIO: 入出力エラーが発生しました。これは、バックグラウンドプロセスグループのプロセスが制御端末を読み取ろうとしたが、読み取り操作が無効になっている、または SIGTTIN シグナルによってブロックされている、またはプロセスグループが孤立している、またはディスクやテープドライブなどの低レベル入出力エラーが発生している可能性があります。

- EISDIR: fd がディレクトリを指しています。

- EBADF: fd が有効なソケットディスクリプタではないか、読み取り操作用に開かれていません。

- EINVAL: fd が接続されたオブジェクトを読み取ることができません。

- EFAULT: buf がユーザーがアクセス可能なアドレス空間を超えています。

44.8 recv()

関数のプロトタイプ：

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

クライアントでもサーバアプリケーションでも、recv() 関数を使用して TCP 接続の他端からデータを受信することができます。これは read() 関数の機能とほぼ同じです。

recv() 関数はまず、ソケット s の受信バッファをチェックします。もし s の受信バッファにデータがないか、プロトコルがデータを受信している場合、recv はプロトコルがデータを完全に受信するまで待機します。プロトコルがデータを完全に受信したら、recv() 関数は s の受信バッファのデータを buf にコピーします。ただし、受信したデータは buf の長さを超える可能性があるので、その場合は s の受信バッファのデータを完全にコピーするために、recv() 関数を何度も呼び出す必要があります。recv() 関数は単にデー

株式会社日昇テクノロジー

タをコピーするだけで、実際のデータの受信はプロトコルによって行われます。recv 関数は、実際にコピーしたバイト数を返します。recv() 関数がコピー中にエラーが発生した場合、それは SOCKET_ERROR を返します。recv() 関数がプロトコルがデータを受信している間にネットワークが切断された場合、それは 0 を返します。

パラメータ：

• sockfd：受信側のソケットディスクリプタを指定します。

• buf：データを受信するバッファを指定します。このバッファは、recv() 関数が受信したデータを格納するために使用されます。

• len：recv() 関数がコピーするデータの長さを指定します。

パラメータ flags は通常 0 に設定しますが、他の値の定義は以下の通りです：

• MSG_OOB：out-of-band で送信されたデータを受信します。

• MSG_PEEK：元のデータを保持します。つまり、受信したデータは削除されず、再度 recv() 関数を呼び出すと、同じデータが buf にコピーされます。

• MSG_WAITALL：指定した len サイズのデータを受信した後にのみ戻るように強制します。ただし、エラーやシグナルが発生した場合を除きます。

• MSG_NOSIGNAL：recv() 関数は SIGPIPE シグナルによって中断されません。戻り値が成功すると、受信した文字数が返されます。失敗すると-1 が返され、エラーの原因は errno に格納されます。

エラーコード：

• EBADF：fd は有効なソケットディスクリプタではない、または読み取り操作のために開かれていない。

• EFAULT：buf がユーザがアクセス可能なアドレス空間を超えています。

• ENOTSOCK：パラメータ s はファイルディスクリプタであり、ソケットではありません。

• EINTR：データを読み取る前にシグナルによって中断されました。

株式会社日昇テクノロジー

• EAGAIN：この操作はプロセスをブロックしますが、パラメータ s のソケットはブロックできません。

• ENOBUFS：buf のメモリ空間が不足しています。

• ENOMEM：メモリが不足しています。

• EINVAL：渡されたパラメータが正しくありません。

44.9 write()

関数のプロトタイプ：

```
ssize_t write(int fd, const void *buf, size_t count);
```

write() 関数は通常、安定した TCP 接続中でデータを転送するために使用されますが、UDP プロトコルでも使用することができます。それはソケットディスクリプタ fd に count バイトのデータを書き込みます。データの開始アドレスは buf で指定されます。関数の呼び出しが成功すると、書き込まれたバイト数が返されます。失敗すると-1 が返され、errno 変数が設定されます。

ネットワークプログラミングでは、ソケットディスクリプタにデータを書き込むときに 2 つの可能性があります：

1. write() 関数の戻り値が 0 より大きい場合、一部のデータまたはすべてのデータが書き込まれたことを示します。このような場合、while ループを使用してデータを継続的に書き込むことができます。ただし、ループ中の buf パラメータと count パラメータは自身が更新する必要があります。つまり、ネットワークプログラミングでは、write 関数はすべてのデータを書き込んだ後に戻ることを保証していません。途中で戻るかもしれません！
2. 戻り値が 0 未満の場合、エラーが発生しました。この場合、エラータイプに基づいて適切な処理を行う必要があります。

したがって、一般的には、データを書き込むときには自分自身で一層をカプセル化することが多いです。これは、データが正しく書き込まれることを保証するためです：

株式会社日昇テクノロジー

```
/* Write "n" bytes to a descriptor. */

ssize_t writen(int fd, const void *vptr, size_t n)

{

    size_t nleft; //書き込む残りのバイト数

    ssize_t nwritten; //既に書き込まれたバイト数

    const char *ptr; //write のバッファ

    ptr = vptr; //書き込むバッファをバックアップする

    nleft = n; //残りの必要なバイト数を初期化する

    //引数で渡された書き込みバイト数の有効性をチェックする

    while (nleft > 0) {

        if ((nwritten = write(fd, ptr, nleft)) <= 0) { //ptr を fd に書き込む

            if (nwritten < 0 && errno == EINTR) //write の戻り値が 0 未満で、信号により中断された場合

                nwritten = 0; /* 再度 write()を呼び出す */

            else

                return(-1); /* エラー 他の 0 未満の場合はエラー */

        }

        nleft -= nwritten; //残りの必要なバイト数 = 現在の残りの必要なバイト数 - この回に書き込んだバイ
ト数

        ptr += nwritten; //次回の書き込み開始位置 = バッファの現在位置 + 既に書き込まれたバイト数

    }

    return(n); //書き込まれたバイト数を返す

}
```

注意: データが比較的単純な場合（例えば単一行データなど）は、そのまま write()を呼び出すだけでも全

株式会社日昇テクノロジー

く問題ありません。状況に応じてコードを書くだけです。上記のコードの包装は、プログラムの堅牢性を保証するためだけです。

注意: この関数によってデータが書き込まれた後、すぐに送信されるわけではありません。いつ送信されるかは TCP/IP プロトコルスタックによって決定されます。

44.10 send()

関数プロトタイプ：

```
int send(int s, const void *msg, size_t len, int flags);
```

クライアントアプリケーションでもサーバーアプリケーションでも、send() 関数を使用して TCP 接続の他端にデータを送信することができます。

パラメータ：

- s：送信側ソケット記述子を指定します。
- msg：送信するデータのバッファを指定します。
- len：実際に送信するデータのバイト数を明示します。
- flags：通常は 0 に設定します

この関数を呼び出すと、send() 関数は最初に待機中のデータの長さ len とソケット s の送信バッファの長さを比較します。len が s の送信バッファの長さより大きい場合、この関数は SOCKET_ERROR を返します。len が s の送信バッファの長さ以下の場合、send() 関数はプロトコルが s の送信バッファのデータを送信中かどうかをまず確認します。送信中であれば、プロトコルがデータを送信し終えるのを待ちます。プロトコルが s の送信バッファのデータの送信を開始していない、または s の送信バッファにデータがない場合は、send() 関数は s の送信バッファの残りの空間と len を比較します。len が残りの空間より大きい場合、send() 関数はプロトコルが s の送信バッファのデータを送信し終えるのを待ち続けます。len が残りの空間より小さい場合、send()関数は buf のデータを s の送信バッファの残りの空間に単にコピーします。

株式会社日昇テクノロジー

send() 関数がデータを成功裏にコピーした場合、実際にコピーしたバイト数を返します。send() 関数がデータをコピー中にエラーが発生した場合、send は SOCKET_ERROR を返します。send がプロトコルがデータを送信中にネットワークが切断された場合も、send 関数は SOCKET_ERROR を返します。

注意: send() 関数が buf のデータを s の送信バッファの残りの空間に成功裏にコピーした後、それらのデータが接続のもう一方の端にすぐに伝送されるわけではありません。

44.11 sendto

関数プロトタイプ：

```
int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
```

sendto() 関数は send 関数に非常に似ていますが、struct sockaddr が指す to 構造体を通じて、どのリモートホストにデータを送信するかを指定します。to パラメータではリモートホストの IP アドレス、ポート番号などを指定する必要があります。tolen パラメータは to 構造体のバイト長を指定します。

44.12 close()

関数のプロトタイプ：

```
int close(int fd);
```

close() 関数は指定したソケットを閉じるために使用されます。ソケットを閉じた後、対応するソケットディスクリプタは使用できなくなります。この関数は比較的シンプルで、特定のソケットディスクリプタを使用する必要がない場合は、それを閉じるだけです。UDP プロトコルでは、close はソケットディスクリプタのリソースを解放します。一方、TCP プロトコルでは、close() 関数を呼び出すと「四方振り手」接続終了が開始され、接続が正式に終了した後、ソケットディスクリプタのリソースが解放されます。

44.13 ioctlsocket()

関数のプロトタイプ：

```
int ioctlsocket( int s, long cmd, u_long *argp);
```

この関数は、ソケットに関連する操作パラメータの取得および設定に使用されます。

株式会社日昇テクノロジー

パラメータ：

1. s：操作するソケットディスクリプタを指定します。

2. cmd：ソケット s に対する操作コマンド。

• FIONBIO：このコマンドは、ソケットの非ブロッキングモードを許可または禁止するために使用されます。このコマンドの下では、argp パラメータは符号なしの長整数を指しており、その値が 0 の場合は非ブロッキングモードを禁止し、その値が非 0 の場合は非ブロッキングモードを許可します。ソケットを作成すると、それはブロックモードになります。つまり、非ブロッキングモードは禁止されています。この状態では、すべての送信/受信関数はブロックされ、送信/受信が成功するまで実行を続けます。一方、非ブロッキングモードでは、すべての送信/受信関数はブロックされず、データが送信できないか受信できない場合は、直接エラーコードをユーザーに返します。これにより、ユーザーはこれらの「予期せぬ」状況を処理し、コードの堅牢性を保証する必要があります。

• FIONREAD：FIONREAD コマンドは、ソケット s が自動的に読み込むデータ量を確定します。これらのデータはすでに受信されていますが、アプリケーションスレッドはまだ読み取っていません。したがって、この関数を使用してこれらのデータの長さを取得することができます。このコマンドの状態では、argp パラメータは符号なしの長整数を指し、関数の戻り値（つまり未読データの長さ）を保存するために使用されます。ソケットが SOCK_STREAM タイプの場合、FIONREAD コマンドは recv()関数が受信したすべてのデータ量を返します。これは通常、ソケットの受信バッファキューにキューイングされているデータの総量と同じです。一方、ソケットが SOCK_DGRAM タイプの場合、FIONREAD コマンドはソケットの受信バッファキューにキューイングされている最初のデータパケットのサイズを返します。

• SIOCATMARK：すべての外部データが読み込まれたかどうかを確認します。

3.　argp：cmd コマンドのパラメータを指すポインタ。

株式会社日昇テクノロジー

実際には、この関数は次のような例を挙げることができます：

```
// ブロックモードに設定します。
u_long mode = 0;
ioctlsocket(s,FIONBIO,&mode);


// 非ブロックモードに設定します。
u_long mode = 1;
ioctlsocket(s,FIONBIO,&mode);
```

44.14 getsockopt()、setsockopt()

```
int getsockopt(int sockfd, int level, int optname,
void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname,
const void *optval, socklen_t optlen);
```

名前からわかるように、この関数はソケットのいくつかのオプションを取得/設定するために使用されます。パラメータ level にはいくつかの一般的なオプションがあります：

- SOL_SOCKET：Socket レベルを示します。
- IPPROTO_TCP：TCP レベルを示します。
- IPPROTO_IP：IP レベルを示します。

株式会社日昇テクノロジー

パラメータ optname は、そのレベルの具体的なオプション名を示します。例えば：

• SOL_SOCKET オプションの場合、SO_REUSEADDR（ローカルアドレスとポートの再利用を許可）、SO_SNDTIMEO（データ送信のタイムアウト時間を設定）、SO_SNDTIMEO（データ受信のタイムアウト時間を設定）、SO_RCVBUF（データ送信バッファのサイズを設定）などがあります。

• IPPROTO_TCP オプションの場合、TCP_NODELAY（Nagle アルゴリズムを使用しない）、TCP_KEEPALIVE（TCP キープアライブ時間を設定）などがあります。

• IPPROTO_IP オプションの場合、IP_TTL（生存時間を設定）、IP_TOS（サービスタイプを設定）などがあります。

44.15 TCP クライアント実験

このセクションでは、socket API 関数を使用して TCP クライアントを実装します。コードの手順はまず、サーバーに接続し、次にクライアントでデータを入力してサーバーに送信し、最後にデータの送信が完了したら接続を終了します。TCP プロトコルのモデルはクライアント<->サーバーであるため、次のセクションでも TCP サーバーを実装し、2 つのプロセス間で相互通信します。

まず、クライアント全体のフロー手順を明確にします：

• 1. socket() 関数を呼び出してソケットディスクリプタを作成します。

• 2. connect() 関数を呼び出して指定したサーバーに接続します。ポート番号はサーバーがリッスンしているポート番号です。

• 3. write() 関数を呼び出してデータを送信します。

• 4. close() 関数を呼び出して接続を終了します。

TCP クライアントのコード:

```
#include <stdio.h>

#include <stdlib.h>
```

株式会社日昇テクノロジー

```
#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>
```

```
#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#define HOST "192.168.0.217" // サーバーの IP アドレスに応じて変更してください

#define PORT 6666 // サーバープロセスがバインドされたポート番号に応じて変更してください

#define BUFFER_SIZ (4 * 1024) // 4k のデータ領域

int main(void)

{

  int sockfd, ret;

  struct sockaddr_in server;

  char buffer[BUFFER_SIZ]; // 入力されたテキストを保存するためのもの

  // 通信のためのエンドポイントを作成

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {

    printf("create an endpoint for communication fail!¥n");

    exit(1);

  }

  bzero(&server, sizeof(server));
```

株式会社日昇テクノロジー

```
server.sin_family = AF_INET;
server.sin_port = htons(PORT);
server.sin_addr.s_addr = inet_addr(HOST);
// TCP 接続の確立
```

```
if (connect(sockfd, (struct sockaddr *)&server, sizeof(struct sockaddr)) == -1) {
    printf("connect server fail...¥n");
    close(sockfd);
    exit(1);
}
printf("connect server success...¥n");
while (1) {
    printf("please enter some text: ");
    fgets(buffer, BUFFER_SIZ, stdin);
    // 「end」が入力されたら、ループから抜け出す（プログラムを終了する）
    if(strncmp(buffer, "end", 3) == 0)
        break;
    write(sockfd, buffer, sizeof(buffer));
}
close(sockfd);
exit(0);
}
```

株式会社日昇テクノロジー

44.16 TCP サーバーの実験

次に、クライアントからの接続を受け入れ、クライアントからのデータを端末に表示するサーバーコードを実装します。

サーバーのコードの流れは以下の通りです:

1. socket() 関数を呼び出して、ソケット記述子を作成します。
2. bind() 関数を呼び出して、監視するポート番号をバインドします。
3. listen() 関数を呼び出して、サーバーを監視状態にします。
4. accept() 関数を呼び出して、クライアントからの接続要求を処理します。
5. read() 関数を呼び出して、クライアントから送信されたデータを受信します。
6. close() 関数を呼び出して、接続を終了します。

サーバーのコード:

```
#include <stdio.h>
#include <netdb.h>
#include <unistd.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#define MAX 10*1024
#define PORT 6666
// Driver function
int main()
```

株式会社日昇テクノロジー

```
{
char buff[MAX];
int n;
int sockfd, connfd, len;
struct sockaddr_in server, client;
```

```
// socket create and verification
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
printf("socket creation failed...¥n");
exit(0);
}
printf("socket successfully created..¥n");
bzero(&server, sizeof(server));
// assign IP, PORT
server.sin_family = AF_INET;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(PORT);
// binding newly created socket to given IP and verification
if ((bind(sockfd, (struct sockaddr*)&server, sizeof(server))) != 0) {
printf("socket bind failed...¥n");
exit(0);
}
```

株式会社日昇テクノロジー

```
printf("socket successfully binded..¥n");
// now server is ready to listen and verification
if ((listen(sockfd, 5)) != 0) {
printf("Listen failed...¥n");
exit(0);
}
printf("server listening...¥n");
len = sizeof(client);
// accept the data packet from client and verification
connfd = accept(sockfd, (struct sockaddr*)&client, &len);
if (connfd < 0) {
printf("server acccept failed...¥n");
exit(0);
}
printf("server acccept the client...¥n");
// infinite loop for chat
while(1) {
bzero(buff, MAX);
// read the messtruct sockaddrge from client and copy it in buffer
if (read(connfd, buff, sizeof(buff)) <= 0) {
printf("client close...¥n");
close(connfd);
```

株式会社日昇テクノロジー

```
break;
}
// print buffer which contains the client contents
printf("from client: %s¥n", buff);
// if msg contains "Exit" then server exit and chat ended.
```

```
if (strncmp("exit", buff, 4) == 0) {
printf("server exit...¥n");
close(connfd);
break;
}
}
// After chatting close the socket
close(sockfd);
exit(0);
}
```

44.17 実験現象

それぞれ embed_linux_tutorial/base_code/system_programing/tcp_client と embed_linux_tutorial/base_code/system_programing/tcp_server ディレクトリに移動し、make コマンドを実行してクライアントとサーバーのコードをコンパイルします。次に、2 つのターミナルを開き、まずサーバープログラムを実行し、次にクライアントプログラムを実行します。その後、クライアントプログラムで送信したいデータを入力し、それを送信します。すると、サーバープロセスでデータが表示されることが確認できます。現象は以下の通りです：

株式会社日昇テクノロジー

クライアントプロセス：

```
 tcp_client git:(master) ./targets
connect server success...
please enter some text: abcdefg
please enter some text: aaaaaaaaaaaa
please enter some text: bbbbbbbbbbbbbbbbbb
```

```
please enter some text: 666666666666666666666666
please enter some text: embedfire socket api
please enter some text: csun
please enter some text: exit
 tcp_client git:(master)
```

サーバープロセス：

```
 tcp_server git:(master) ./targets
socket successfully created..
socket successfully binded..
server listening...
server acccept the client...
from client: abcdefg
from client: aaaaaaaaaaaa
from client: bbbbbbbbbbbbbbbbbb
from client: 666666666666666666666666
from client: embedfire socket api
from client: csun
```

株式会社日昇テクノロジー

client close...

tcp_server git:(master)

第 45 章 select、poll、epoll の違いについて深く理解する

45.1 序章

本章では、Linux ネットワーク IO の開発環境について議論します。

45.2 同期 I/O

オペレーティングシステムでは、プログラムの実行空間はカーネル空間とユーザー空間に分かれています。ユーザー空間の全ての IO 操作コード（ファイルの読み書き、ソケットの送受信など）は、システムコールを介してカーネル空間で実際の操作が行われます。

そして、CPU の速度がハードディスクやネットワークなどの I/O よりもはるかに速いことを知っています。あるスレッドで、CPU がコードを実行する速度は非常に速いですが、ファイルの読み書きやネットワークデータの送信など、I/O 操作に遭遇すると、I/O 操作が完了するまで待機しなければならず、その後に次の操作を続行できます。このような状況を同期 I/O と呼びます。

あるアプリケーションが実行されているとき、特定のファイルの読み書きが必要になると、I/O 操作が発生します。I/O 操作のプロセス中に、システムは現在のスレッドを中断し、他の CPU 実行コードは現在のスレッドによって実行されなくなります。これが同期 I/O 操作です。なぜなら、一つの I/O 操作が現在のスレッドをブロックし、他のコードの実行を阻止するため、マルチスレッドやマルチプロセスを使用してコードを並行して実行することができます。つまり、あるスレッド/プロセスが中断されても、他のスレッドやプロセスには影響しません。

マルチスレッドとマルチプロセスはこのような並行問題を解決するものですが、システムは無限にスレッ

株式会社日昇テクノロジー

ド/プロセスを増やすことはできません。システムがスレッド/プロセスを切り替えるコストも非常に大きいため、一度に多くのスレッド/プロセスが存在すると、CPU の時間はスレッド/プロセスの切り替えに費やされ、実際にコードを実行する時間が減り、これによりシステムの性能が大幅に低下する可能性があります。

マルチスレッドとマルチプロセスはこの問題を解決する一つの方法ですが、I/O 問題を解決する別の方法は非同期 I/O です。もちろん、他にも解決方法はあります。

45.3 非同期 I/O

プログラムが I/O 操作を必要とするとき、それは I/O 操作の指示だけを出し、I/O 操作の結果を待たずに他のコードを実行します。一定時間後、I/O が結果を返すと、CPU に処理を通知します。これにより、ユーザ空間のプログラムは、カーネル空間での I/O が実際の操作を完了するのを待つことなく、他のタスクを実行でき、CPU の利用率が向上します。

簡単に言えば、ユーザはカーネルが実際に io の読み書き操作を完了するのを待たずに直接戻るということです。

45.4 ブロッキング I/O

Linux では、デフォルトではすべてのソケットがブロックされています。典型的な読み取り操作の流れは次のようになります：

株式会社日昇テクノロジー

図 1: socket_io001

ユーザプロセスが read()/recvfrom()などのシステムコール関数を呼び出すと、それはカーネル空間に入ります。このネットワーク I/O にデータがない場合、カーネルはデータの到着を待つ必要があります。一方、ユーザプロセスでは、プロセス全体がブロックされ、カーネル空間からデータが返されるまで待ちます。カーネル空間のデータが準備できたら、それはデータをカーネル空間からユーザ空間にコピーします。この時点で、ユーザプロセスはブロック状態を解除し、再度実行を開始します。

したがって、ブロッキング I/O の特徴は、I/O の実行の 2 つの段階（ユーザ空間とカーネル空間）が両方ともブロックされていることです。

株式会社日昇テクノロジー

45.5 ノンブロッキング I/O

Linux では、ソケットを設定して非ブロッキングモードにすることができます。この場合、カーネル空間にデータがない場合、それはすぐに結果を返し、ブロックしません。この時点で、ユーザプロセスはこの結果に基づいて自由に設定できます。たとえば、データのリクエストを続けるか、リクエストを続けないかなどです。非ブロッキングソケットで読み取り操作を実行すると、流れは次のようになります：

ユーザプロセスが read()/recvfrom()などのシステムコール関数を呼び出すと、カーネル空間のデータがまだ準備できていない場合、それはユーザプロセスをブロックせずにすぐにエラーを返します。

アプリケーションプロセスにとって、read()操作を開始した後、待つ必要はありません。すぐに結果を得ることができます。

ユーザプロセスが結果がエラーであると判断した場合、それはカーネルのデータがまだ準備できていないことを知り、それは再度 read()/recvfrom()などの関数を呼び出すことができます。

カーネル空間のデータが準備できたら、それはデータをカーネル空間からユーザ空間にコピーします。この時点で、ユーザプロセスはデータを得ることができます。

したがって、非ブロッキング I/O の特徴は、ユーザプロセスがカーネル空間のデータが準備できているかどうかを絶えずアクティブに問い合わせる必要があることです。

45.6 マルチプレクシング I/O

マルチプレクシング I/O とは、言う select、poll、epoll などの操作のことで、その利点は、単一のプロセスが同時に複数のネットワーク接続の I/O を処理できることです。この機能を実現する原理は、select、poll、epoll などの関数が自分が担当するすべてのソケットを絶えずポーリングし、あるソケットにデータが到着したら、ユーザプロセスに通知することです。

一般的に、I/O マルチプレクシングは以下のような状況で多く使用されます：

1. クライアントが複数のディスクリプタを処理する場合。
2. サーバーが高並行性でネットワーク接続を処理する場合。

株式会社日昇テクノロジー

3. サーバーがリスニングソケットと接続済みソケットの両方を処理する必要がある場合、通常は I/O マルチプレクシングが必要です。

ユーザースペース
カーネルスペース
read () /readform () などの関数を呼び出し
プロセスはブロック状態に入っていない
データがないので直ちに戻り
read () /readform () などの関数を呼び出し
プロセスはブロック状態に入っていない
データがないので直ちに戻り
read () /readform () などの関数を呼び出し
プロセスはブロック状態に入っていない
データがないので直ちに戻り
read () /readform () などの関数を呼び出し
データをカーネルスペースからユーザースペースにコピーします
読み取ったデータを返します
ユーザースペース
カーネルスペース

株式会社日昇テクノロジー

4. サーバーが TCP と UDP の両方を処理する必要がある場合、通常は I/O マルチプレクシングを使用します。

5. サーバーが複数のサービスやプロトコルを処理する必要がある場合、通常は I/O マルチプレクシングを使用します。

マルチプロセスとマルチスレッド技術と比較して、I/O マルチプレクシング技術の最大の利点は、システムのオーバーヘッドが小さいことです。システムはプロセス/スレッドを作成したり、これらのプロセス/

スレッドを維持したりする必要がないため、システムのオーバーヘッドが大幅に削減されます。ただし、select、poll、epoll は本質的に同期 I/O であり、それらは読み書きイベントが準備できた後に自分自身で読み書きを行う必要があります。つまり、この読み書きプロセスはブロッキングのものです。一方、非同期 I/O では、自分自身で読み書きを行う必要はありません。非同期 I/O の実装は、データをカーネルからユーザ空間にコピーする責任があります。

45.6.1 select

直訳すると、select：select メカニズムは、それが担当するすべてのソケットを監視します。一つまたは複数のソケットが読み取り可能または書き込み可能になったとき、それは戻ります。しかし、すべてのソケットが読み取り不可能または書き込み不可能である場合、このプロセスはブロックされ、タイムアウトまたはソケットが読み書き可能になるまで待ちます。select 関数が戻った後、fdset を走査して、準備ができているディスクリプタを見つけることができます。

45.6.1.1 select の全体的な処理手順は次の通りです

1. ユーザプロセスが select()関数を呼び出すと、現在読み書き可能なソケットがない場合、ユーザプロセスはブロック状態になります。

2. カーネル空間については、それはユーザ空間から fd_set をカーネル空間にコピーし、その後カーネル内ですべてのソケットディスクリプタを一度に走査します。条件を満たすソケットディスクリプタがない場合、カーネルはスリープ状態になります。デバイスドライバが自身のリソースが読み書き可能になったとき、それはその待機キュー上のスリープ状態のカーネルプロセスを起こします。つまり、ソケットが読み書き可能になったときに起こす、またはタイムアウトしたときに起こす。

3. select()関数の呼び出し結果をユーザプロセスに返します。準備ができているソケットディスクリプタの数を返します。タイムアウトした場合は 0 を返し、エラーが発生した場合は-1 を返します。

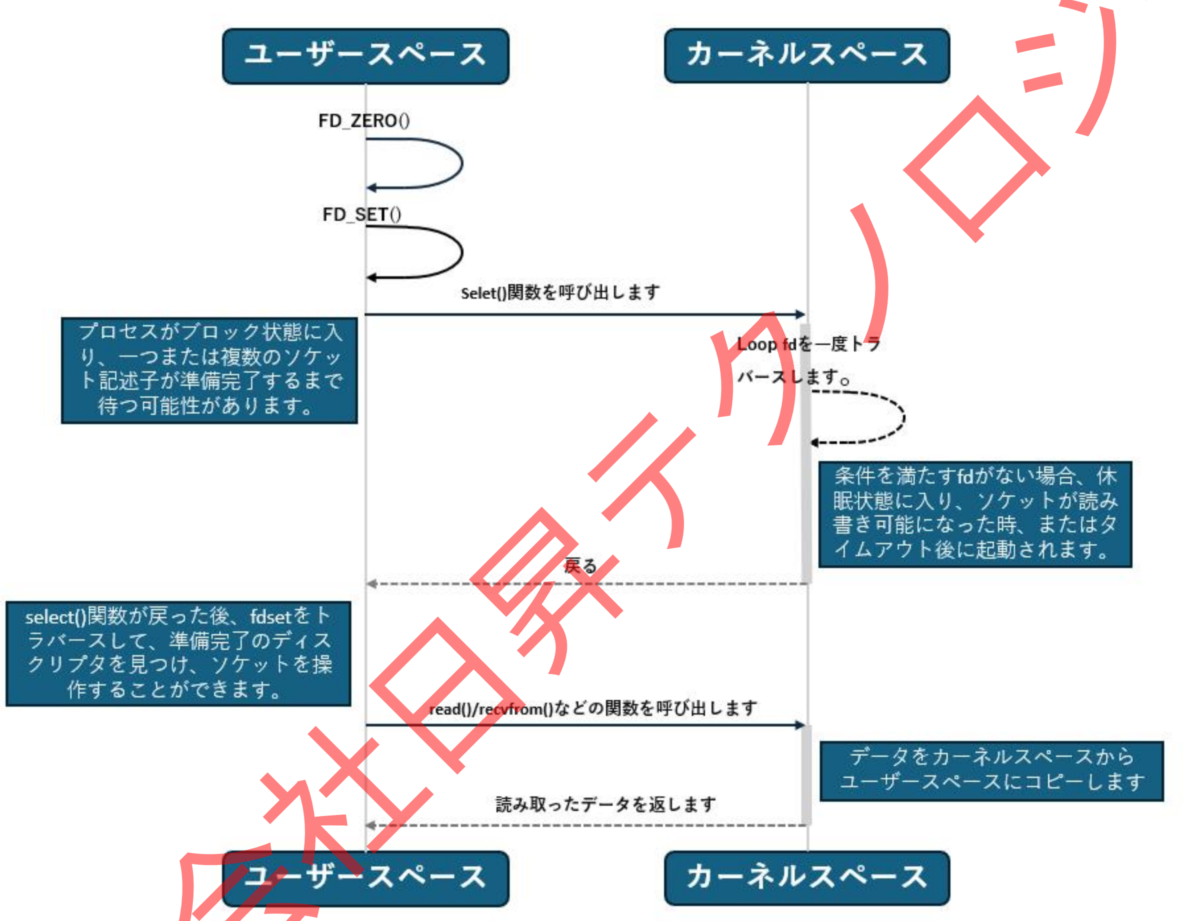
4. 注意点として、select()関数が戻った後も、準備ができているソケットディスクリプタを見つけるため

株式会社日昇テクノロジー

には、引き続きポーリングが必要です。この時点で、ユーザプロセスはソケットを操作することができます。

select は現在ほぼすべてのプラットフォームでサポートされており、その良好なクロスプラットフォームサポートもその一つの利点です。

もちろん、select にも多くの欠点があります：



• 1. 単一のプロセスが監視できるファイルディスクリプタの数には最大限度があり、Linux では通常 1024 です。これは、マクロの定義を変更したり、カーネルを再コンパイルしたりすることで増やすことができますが、これにより効率が低下する可能性があります。

• 2. 大量の fd を格納するためのデータ構造を維持する必要があります。これにより、ユーザ空間とカーネル空間でこの構造を転送する際のコピーオーバーヘッドが大きくなります。

• 3. ソケットディスクリプタがアクティブになったときには、毎回すべての fd を一度に走査してそのディスクリプタを見つける必要があります。これにより、大量の時間が消費されます（時間複雑度は

株式会社日昇テクノロジー

O(n)で、ディスクリプタが増えるにつれて、このオーバーヘッドは線形に増加します）。

カーネルにとって、全体の処理手順は次のようになります：

45.6.1.2 select 関数のプロトタイプ：

```
int select(int maxfdp1,fd_set *readset,fd_set *writeset,fd_set *exceptset,

,→const struct timeval *timeout)
```

パラメータの説明：

- maxfdp1 は、興味のあるソケットディスクリプタの数を指定します。その値は、ソケットの最大ソケットディスクリプタに 1 を加えたものです。ソケットディスクリプタ 0、1、2… maxfdp1-1 はすべて興味があると設定されます（つまり、それらが読み取り可能、書き込み可能であるかどうかを確認します）。
- readset：このソケットディスクリプタが読み取り可能なときにのみ戻ります。
- writeset：このソケットディスクリプタが書き込み可能なときにのみ戻ります。
- exceptset：このソケットディスクリプタが例外状態になったときにのみ戻ります。
- timeout：タイムアウトの時間を指定します。タイムアウトした場合も戻ります。

特定の条件に興味がない場合、それを空のポインタに設定することができます。

戻り値：準備ができているソケットディスクリプタの数を返します。タイムアウトした場合は 0 を返し、エラーが発生した場合は-1 を返します。

株式会社日昇テクノロジー

45.6.1.3 select の欠点

1. select を呼び出すたびに、fd 集合をユーザモードからカーネルモードにコピーする必要があります。これは、fd が多い場合には大きなオーバーヘッドとなります。

2. 同様に、select を呼び出すたびに、カーネルはすべての fd を一度に走査する必要があります。これは、fd が多い場合には大きなオーバーヘッドとなります。

3. select()関数が戻った後も、準備ができているソケットディスクリプタを見つけるためには、引き続きポーリングが必要です。

4. select がサポートするファイルディスクリプタの数は少ない、デフォルトは 1024 です。

株式会社日昇テクノロジー

45.6.2 poll

poll の実装は select と非常に似ていますが、fd 集合の記述方法が異なります。poll は pollfd 構造を使用し、select は fd_set 構造を使用します。poll はソケットディスクリプタの数に制限がありません。これは、それがこれらのソケットディスクリプタをメンテナンスするためにリンクリストを使用しているからです。他の部分はほとんど select()関数と同じで、poll()関数が戻った後、pollfd をポーリングして、準備ができているディスクリプタを見つけることができます。ただし、poll は最大ファイルディスクリプタ数の制限がありません。poll と select には同じ欠点があります。大量のファイルディスクリプタを含む配列がユーザモードとカーネルモードの間で全体的にコピーされ、これらのファイルディスクリプタが準備できているかどうかに関係なく、そのオーバーヘッドはファイルディスクリプタの数が増えるにつれて線形に増大します。

関数のプロトタイプ：

```
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

45.6.3 epoll

45.6.3.1 epoll の原理

実際には、select と poll に比べて、epoll はより柔軟ですが、その核心的な原理は、ソケットディスクリプタが準備できている（読み取り可能、書き込み可能、例外が発生している）ときに、アプリケーションプロセスに通知し、どのソケットディスクリプタが準備できているかを伝えることです。ただし、通知の処理方法が異なるだけです。

epoll は、epfd（epoll ファイルディスクリプタ）を使用して多くのソケットディスクリプタを管理します。epoll はソケットディスクリプタの数に制限がなく、ユーザー空間のソケットディスクリプタのイベントをカーネルのイベントテーブルに置くことで、ユーザー空間とカーネル空間のコピーは一度だけ必要です。epoll が記録したソケットが準備完了になると、epoll はコールバック方式でこの fd をアクティブにし、

株式会社日昇テクノロジー

epoll_wait で通知を受け取り、どのソケットが準備完了になったかをアプリケーション層に知らせます。この通知方式では、どのソケットが準備完了になったかを直接得られるため、select や poll と比較して、ソケットリストを走査する必要がなく、時間の複雑さは O(1)で、記録されたソケットが増えてもオーバーヘッドが増大しない。

45.6.3.2 epoll の操作モード

epoll は、ソケットディスクリプタに対して二つのモード、LT（レベルトリガー）と ET（エッジトリガー）を持ちます。LT モードはデフォルトモードで、LT と ET モードの違いは以下の通りです：

- LT モード：レベルトリガーモードとも呼ばれ、epoll_wait がソケットディスクリプタを準備完了と検出した場合にアプリケーションに通知しますが、アプリケーションはすぐにそれを処理する必要はありません。次回 epoll_wait を呼び出したときに再度通知が発生します。

- ET モード：エッジトリガーモードとも呼ばれ、epoll_wait がソケットディスクリプタを準備完了と検出した場合にアプリケーションに通知し、アプリケーションはそれを即座に処理しなければなりません。処理しない場合、次回 epoll_wait を呼び出しても再度通知は発生しません。

ET モードは epoll イベントが繰り返しトリガされる回数を大幅に減らすため、LT モードよりも効率が高いです。epoll が ET モードで動作する場合は、非ブロッキングソケットを使用する必要があり、一つのファイルハンドルのブロッキング読み取り/書き込み操作が複数のファイルディスクリプタの処理を飢餓させることを避けるためです。

45.6.3.3 epoll の関数

epoll には epoll_create()、epoll_ctl()、epoll_wait()の 3 つのシステムコール関数があります。

45.6.3.3.1 epoll_create()

```
int epoll_create(int size);
```

株式会社日昇テクノロジー

epoll は、epfd（epoll ファイルディスクリプタ、またはハンドルとも呼ばれます）を作成することにより、複数のソケットディスクリプタを管理します。epoll ハンドルを作成した後、それはファイルディスクリプタ（fd）の値を占有し、close()で閉じなければ、ファイルディスクリプタの枯渇を引き起こす可能性があります。これが、epoll が一つの epfd で多くのソケットディスクリプタを管理する理由です。size パラメータは、監視する数を内核に伝えるために使われ、実際には内核が空間を割り当てて、ユーザーが監視したいソケット fd が読み取り可能、書き込み可能、または他の例外があるかどうかを格納します。十分なメモリ空間があれば、size は任意に設定でき、1GB のメモリでは約 10 万のポートを監視できます。

45.6.3.3.2 epoll_ctl()

この関数は、epoll ファイルディスクリプタ上のイベントを制御するために使用され、イベントの登録、変更、削除ができます。

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

パラメータ：

- epfd：epoll_create()関数によって返される epoll ファイルディスクリプタ（ハンドル）。
- op：操作のオプションで、以下の三つのオプションがあります：
- EPOLL_CTL_ADD：epoll ハンドルに監視対象のソケットディスクリプタ fd を登録します。
- EPOLL_CTL_MOD：epoll ハンドルに既に登録されている fd の監視イベントを変更します。
- EPOLL_CTL_DEL：epoll ハンドルから既に登録されているソケットディスクリプタを削除します。
- fd：監視対象のソケットディスクリプタ。
- event：event 構造体は以下の通りです：

株式会社日昇テクノロジー

```
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;


struct epoll_event {
    uint32_t events; /* Epoll イベント */
    epoll_data_t data; /* ユーザーデータ変数 */
};
```

- events は以下のマクロの組み合わせです：

- EPOLLIN：対応するファイルディスクリプタが読み取り可能です（端点の SOCKET が正常に閉じた場合を含む）。

- EPOLLOUT：対応するファイルディスクリプタが書き込み可能です。

- EPOLLPRI：対応するファイルディスクリプタに緊急のデータが読み取り可能です（ここでは帯域外データの到来を意味します）。

- EPOLLERR：対応するファイルディスクリプタにエラーが発生しました。

- EPOLLHUP：対応するファイルディスクリプタが切断されました。

- EPOLLET：EPOLL をエッジトリガーモードに設定します。これはレベルトリガーモードと対比されます。

- EPOLLONESHOT：イベントを一度だけ監視します。このイベントを監視した後、このソケットを

株式会社日昇テクノロジー

引き続き監視したい場合は、再度このソケットを EPOLL キューに追加する必要があります。

45.6.3.3.3 epoll_wait()

epoll_wait()関数は、監視しているイベントが発生するのを待つために使用され、select()関数の呼び出しに似ています。

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

パラメータ：

- epfd：epoll_create()関数によって返される epoll ファイルディスクリプタ（ハンドル）。
- events：カーネルから得られるイベントの集合。
- maxevents：カーネルにこの events の大きさを知らせるもので、この maxevents の値は epoll_create()時に指定した size より大きくてはなりません。
- timeout：タイムアウト時間。
- 関数の戻り値は、処理が必要なイベントの数をリストし、0 が返された場合はタイムアウトを意味します。

45.6.3.4 なぜ epoll はより効率的なのか

1. epoll_wait()関数を呼び出すと、返されるのは実際のディスクリプタではなく、準備ができているディスクリプタの数をリストす値です。この時点で、epoll が指定した配列から対応する数のソケットディスクリプタを順次取得するだけで、すべてのソケットディスクリプタを走査する必要はありません。したがって、ここでの時間複雑度は O(1)です。

2. さらに、メモリマッピング（mmap）技術も使用しています。これにより、システムコール時にこれらのソケットディスクリプタのコピーのオーバーヘッドが完全に省略されます（ユーザ空間からカーネル空間へのコピー操作が必要です）。mmap は、ユーザ空間の一部のアドレスとカーネル空間の一部のアドレスを同じ物理メモリアドレスに同時にマッピングします（ユーザ空間もカーネル空間も仮想アド

株式会社日昇テクノロジー

レスであり、最終的にはアドレスマッピングを通じて物理アドレスにマッピングされます）。これにより、この物理メモリはカーネルとユーザの両方に見え、ユーザモードとカーネルモード間のデータ交換が減少し、コピーに依存しないため、カーネルは直接 epoll が監視するソケットディスクリプタを見ることができ、効率が非常に高いです。

3. もう一つの本質的な改善点は、epoll がイベントベースの準備通知方式を採用していることです。select/poll では、プロセスが特定の方法を呼び出すと、カーネルはすべての監視されているソケットディスクリプタを走査します。しかし、epoll は事前に epoll_ctl()を使ってソケットディスクリプタを登録し、epoll が管理するソケットディスクリプタが準備できていると検出したとき、カーネルは callback のようなコールバックメカニズムを採用して、このソケットディスクリプタを迅速に活性化します。プロセスが epoll_wait()を呼び出すと、通知を受け取ることができます。つまり、epoll の最大の利点は、それが準備ができているソケットディスクリプタだけを管理し、ソケットディスクリプタの総数とは無関係であるということです。

株式会社日昇テクノロジー