マイクロカーネルの設計と実装

怒田 晟也 著

はじめに

OS カーネルの機能を必要最低限に抑え，ファイルシステムや TCP/IP，デバイスドライバといった主要機能をアプリケーションとして実装する「マイクロカーネル」というカーネル設計手法があります。小さなカーネルを実現するために，モノリシックカーネルには見られない特色・面白さがマイクロカーネルにはあります。

本書では，マイクロカーネルの各機能（プロセス管理・IPC など）を 3 つのステップで学んでいきます。まず，機能の基本的な概念をざっくり解説します。次に，C 言語で新たに書いたシンプル（コア部分の LoC は 3000 行以下）なマイクロカーネル「Resea」のソースコードを用いて，具体的にどう実装するのか雰囲気をつかみます。 そして，L4 や Zircon (Fuchsia) といった実用レベルのカーネルたちと比較しながら，マイクロカーネルの設計と実装を学びます。

ただし，解説は Resea がメインです。L4 などはかなりざっくり説明しています。というのも，L4 や Zircon を細かく解説するとそれ 1 つで何冊も厚い本ができてしまうのです。L4 を始めとする他のマイクロカーネルをもっと知りたいという方は参考文献に載っているユーザマニュアルやドキュメントを読んでみてください。

想定読者

本書は，以下のような人には楽しんで頂けるはずです。

- モノリシックカーネルしか解説しない巷の OS 解説本に飽きた人
- なぜかマイクロカーネルの研究をしようと志している学部生
- マイクロカーネルを自ら設計し実装しようと思っている酔狂な人

前提知識

- CPU・OS カーネルの基本的な仕組み。プロセスとスレッドの違いや仮想メモリの仕組みが分かる。
- C 言語の基本的な読解力
- UNIX の基礎知識

本書で扱うこと

- マイクロカーネルとは何か
- どのような設計がマイクロカーネルで見られるか
- マイクロカーネルの実装例

本書では扱わないこと

- コンテキストスイッチの仕組みといった，モノリシックカーネルと被る内容。
- ページングの仕組みといった，CPU の仕様・仕組み。

電子版のダウンロード

本書の電子版は https://seiya.me/microkernel-book からダウンロードできます。

ライセンス・商標について

本書の本文はクリエイティブ・コモンズ表示 4.0 です。

本書では様々なマイクロカーネルのソースコードを引用しています。以下の通り，Resea とはライセンスが異なるので注意してください。

- Resea: パブリックドメインまたは MIT ライセンス（デュアルライセンス）
- MINIX: BSD ライセンスを基にした独自ライセンス
- Fiasco.OC (L4): GPLv2 ライセンス
- Zircon: MIT ライセンス（ただし Fuchsia 全体は BSD ライセンス等も含む）
- GNU Hurd: GPL ライセンス

また，本文中に登場する社名・製品名等は商標または登録商標です。本文では TM マーク等は省略しています。

目次

第 1 章　マイクロカーネル入門　6
1.1　マイクロカーネルとは　6
1.2　マイクロカーネルの機能　7
1.3　クライアント・サーバモデル　8
1.4　長所・短所　8
1.5　世界を支えるマイクロカーネルたち　9
1.6　歴史　10

第 2 章　Resea 入門　12
2.1　なぜ Resea なのか　12
2.2　Resea の機能　13
2.3　開発環境のセットアップ　13
2.4　ビルド・実行　14
2.5　補足説明　14

第 3 章　プロセス　18
3.1　プロセス・スレッドの中身　18
3.2　マイクロカーネルのプロセス　19
3.3　ページフォルト処理　19
3.4　例外処理　20
3.5　実装（Resea）　20
3.6　実装（Mach）　25
3.7　実装（L4）　25
3.8　実装（MINIX3）　27
3.9　実装（Zircon）　28

第 4 章　システムコール　32
4.1　カーネルサーバ　33
4.2　実装（Resea）　33
4.3　実装（Mach）　39
4.4　実装（L4）　40
4.5　実装（MINIX3）　41

4.6 実装（Zircon） 41

第 5 章 プロセス間通信（IPC） 44
5.1 API 44
5.2 メッセージの内容 45
5.3 間接型 IPC vs. 直接型 IPC 45
5.4 クローズド受信とオープン受信 46
5.5 同期的 IPC vs. 非同期 IPC 46
5.6 通知（Notifications） 46
5.7 タイムアウト 47
5.8 IPC fastpath 47
5.9 実装（Resea） 47
5.10 実装（Mach） 53
5.11 実装（L4） 54
5.12 実装（MINIX3） 56
5.13 実装（Zircon） 56

第 6 章 ユーザランド 58
6.1 ブート処理 58
6.2 シングルサーバ OS vs. マルチサーバ OS 58
6.3 POSIX 互換 59
6.4 IPC スタブ 59
6.5 ユーザランドプログラミングの例（Resea） 63
6.6 ユーザランドプログラミングの例（MINIX3） 67

第 7 章 高度なトピック 70
7.1 非同期 IPC は本当に必要ないのか 70
7.2 ユーザレベルメモリ管理 72
7.3 内部で動的メモリ割当をしないカーネル 72
7.4 ユーザレベルスケジューラ 72
7.5 信頼性の向上 73
7.6 ハードウェア支援による IPC 高速化 74

第 8 章 おわりに 76
8.1 Resea の開発に参加しよう 76
8.2 次に何をするか 77

付録 A 参考文献 78

付録 B Resea カーネルのソースコード 80

1 マイクロカーネル入門

本章ではマイクロカーネルの概要について解説します。

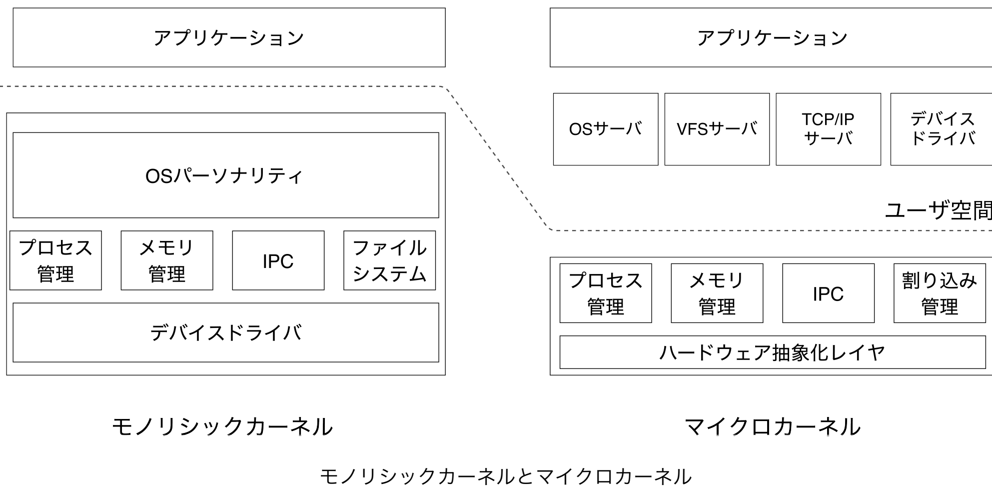
1.1 マイクロカーネルとは

マイクロカーネルは従来のモノリシックカーネルが持っているデバイスドライバや，ファイルシステム，TCP/IP といった主要機能を，ユーザランド（userland）[*1]の普通のプログラムで実装できるようにしたカーネルです。

ただ「マイクロカーネル」は一義的でなく，とてもぼんやりした概念です。何がマイクロカーネルをマイクロカーネルたらしめるのでしょうか。筆者的には，２つの考え方があると思っています。

１つ目は，高機能化に伴いどんどん肥大化していくというモノリシックカーネルの問題への対処としてのマイクロカーネルです。ユーザランドでカーネルの機能を実装できるようにすることで，カーネル本体のコード量を抑えるというのが目標です。

この見方で言うと，Linux でもユーザランドでデバイスドライバを動かせる機能（Userspace I/O）があるので「Linux はマイクロカーネル」と言えてしまいます。しかし，Linux はモノリシックカーネルにこだわりがあるわけではなくマイクロカーネルのいいとこ取りをしているだけと考えると，特にこの解釈も間違いではないと筆者は考えています。



モノリシックカーネルとマイクロカーネル

[*1] ユーザランドの他に「ユーザ空間（userspace）」や単に「ユーザ」と表記されるときもあります。「ユーザアカウント」と混同しないよう注意。

2つ目は「ユーザランドでは実装が難しい機能の詰め合わせ」がマイクロカーネルであるという考え方です。ユーザランドで実装できるものはユーザランドでやることで，できるだけ小さいカーネルを目指します。本書で紹介する Resea と L4 はこちらの考え方ですね。

機能を必要最低限に抑えることに加えて，マイクロカーネルでは「物理メモリをどう配分するか」だとか「次に実行するスレッドをどう選ぶか（スケジューリング）」だといった「決めごと（方針）」をカーネル内で行うことを嫌います。これもユーザランドでやらせたいのです。この考え方が「機構と方針の分離」です。

■ 機構と方針の分離

機構と方針の分離（separation of mechanism and policy）は，マイクロカーネルの基幹となる思想です。要は「カーネルは『仕組み』だけを提供する」ということです。

例として，メモリ割り当てをどう実装するか考えてみましょう。モノリシックカーネルでは，空いているメモリページを見つけて，ページテーブルを更新するという一連の操作を全てカーネル内で完結させます。

対するマイクロカーネルは「ページテーブルの更新」という操作だけを提供し，空いているメモリページの情報をどのようなデータ構造で保持し，どのようなアルゴリズムで選択するのかといった「方針」を全てユーザプロセスに委ねます。

機構と方針の分離を行うことで，様々なニーズに合ったシステムを同一のカーネルで構築できる柔軟性を提供できるのです。いわば「OS のフレームワーク」の側面を持っています。

1.2 マイクロカーネルの機能

マイクロカーネルがどのような機能を持つかは一概に言うことはできませんが，だいたい以下に挙げる機能を備えていることが多いです。

- プロセス・スレッド管理・スケジューラ
- プロセス間通信（IPC）
- アドレス空間（ページテーブル）操作
- 割り込み管理
- タイマー処理

重要なことは，どうしてもカーネル内で処理しなければならないような機能に絞られているという点です。POSIX（Portable Operating System Interface）のような「OS」の基本的な概念・インタフェース（OS パーソナリティと呼びます）は，ユーザランドで実装します。

プロセスやスレッド，スケジューラはモノリシックカーネルと同じ概念です。とはいっても，モノリシックカーネルに比べて非常に限られた簡素な機能のみ持っています。ファイル記述子といった OS の機能は，ユーザランドのプロセス管理サーバで実装されます。

プロセス間通信（IPC）は，マイクロカーネルでは特に重要な要素です。ユーザプロセス同士通信しあって OS の機能を提供するので，通信部分がシステムの性能に大きく影響します。

Linux ではシステムコールを発行してカーネルに処理をしてもらいますが，マイクロカーネルでは

「サーバへのメッセージの送信」と「サーバからのメッセージの受信」という風にメッセージパッシングで実装します。大体何でもメッセージパッシングでやります。デバイスからの割り込みやページフォルトなど本来カーネル内で処理される物も，対応するユーザプロセスへメッセージとして送られ，処理されます。

1.3 クライアント・サーバモデル

マイクロカーネルはユーザプロセスを区別することは特にありませんが，他のプロセスにサービス（ファイルシステム等）を提供するプロセスのことを便宜上「サーバ」と呼びます。ネットワークの文脈において一般的に使われる「サーバ」とは違うので注意してください。UNIX のデーモンが近い概念です。

マイクロカーネルでは，アプリケーションがサーバにリクエストメッセージを送信し，サーバに処理をしてもらい，レスポンスメッセージを受け取る，というクライアント・サーバモデルが一般的です。インターネットでいうところの P2P 通信のような，皆対等に **同じ**処理をするということは マイクロカーネルでは見られません。

1.4 長所・短所

マイクロカーネルの機能について学んだところで，どのような特徴があるのかを見てみましょう。よく言われるマイクロカーネルの長所・短所は以下の通りです。

■ 長所

- 理解しやすく，開発・移植・デバッグしやすい。
- 小さい方が attack surface が狭くなりセキュリティの面で嬉しい。

長所には主観的な部分が含まれています。開発・デバッグのしやすさは状況に依りますし，人によってはモノリシックカーネルの方が優れていると感じるものです。カーネル自体の attack surface は小さくはなるでしょうが，ユーザランドのデバイスドライバや特権をもつサーバも，攻撃の格好の的になります。

とはいえ信頼性という面では，モノリシックカーネルに比べ現状では秀でているとみられています。とりわけ，後ほど紹介する MINIX3 や seL4 は「マイクロカーネルによるシステムの信頼性の向上」を推しています。

■ 短所

大変よく槍玉にあげられる短所は「モノリシックカーネルに比べ性能が劣る」点です。モノリックカーネルでは関数呼び出しで行っているところを，マイクロカーネルではプロセス間通信で置き換えるわけです。プロセス間通信にはシステムコール（カーネル）の呼び出しやアドレス空間の切り替え，コンテキストスイッチなど，様々な時間のかかる処理を行う必要があります。

どう考えてもマイクロカーネルは遅くなりそうです。しかし，性能のことを考えてしっかり設計す

ればマイクロカーネルでもかなり速いことが L4 カーネル（後述）によって示されました。また近年，ハードウェア（CPU）レベルの支援を上手く使って IPC を高速化する研究もあるようです。

また，性能が劣るといっても誰しもが性能を第一に求めている訳ではありません。マイクロカーネルは性能より信頼性を重視する場面でよく使われているようです。つまり，**モノリシックカーネルとマイクロカーネルはライバルというよりは共存関係**にあります。C++ や Java，Go といった様々なプログラム言語が使われているのと同じように，OS カーネルも適材適所というわけです。

1.5 世界を支えるマイクロカーネルたち

マイクロカーネルはただの研究者のおもちゃではありません。気づかないところで今日も元気に世界を支えています。本章では，実世界で利用されているマイクロカーネルを紹介します。

■ Mach

Mach（マーク）[*2]は第 1 世代マイクロカーネルの特に有名な OS です。1980 年代後半からカーネギーメロン大学の Richard Rashid 氏[*3]を中心に開発されました。

当初は，UNIX ライクな OS である 4.2BSD を拡張する形で開発されていたそうです。性能が悪いので「マイクロカーネルというものは遅い」という認識が広がってしまいました。後々，L4 の登場で「大きすぎるマイクロカーネル[*4]」と見られています。

GNU Hurd や macOS や iOS のカーネルである XNU[*5]で採用されていることで有名です。

■ L4

Jochen Liedtke 氏によって開発されたマイクロカーネルです。 *Liedtke's 4th system* を略して L4 なんだそうです。

L4 は「性能を考えた設計をすれば，マイクロカーネルは遅くない」ということを証明したカーネルです。異様に速いです。研究だけではなく実世界でもしっかり使われています。中でも有名なのは iPhone の Secure Enclave です。携帯電話のベースバンドプロセッサの中には，L4 の派生カーネルが使われているものもあるそうです。

L4 には様々な派生（L4 ファミリー）があり，今日では「L4」という言葉を L4 の派生カーネルたちの総称としてよく使われます。C++ で書かれた L4/Fiasco.OC，移植性を考慮した L4Ka::Pistachio，カーネルに対する形式的検証を行い高い安全性を謳う seL4 が有名です。個人的には seL4 が L4 ファミリーの中でも注目されている印象を受けます。

[*2] *Multi-User Communication Kernel* または *Multiprocessor Universal Communication Kernel*，略して「MUCK」のつもりでしたが，イタリア人の Dario Giuse さんがスペルを「Mach」と間違えたのがこの名前になったきっかけのだそうです。
[*3] かの有名な Microsoft Research の設立者です。https://www.microsoft.com/en-us/research/people/rashid/
[*4] 「カーネルの機能をユーザランドに追いやることで肥大化を阻止」するのがマイクロカーネルであるという観点から見ると，カーネルがいくら大きかろうとマイクロカーネルです。
[*5] 厳密に言うと，BSD 系由来のコンポーネントと融合したハイブリッドカーネルです。

■ MINIX

Andrew Stuart Tanenbaum 氏[*6]を中心に開発されているマイクロカーネルベースの OS です。

当初は教育目的として開発され，オペレーティングシステムの分厚い教科書（通称 MINIX 本）で解説されています。MINIX3 からは，信頼性を重視したマイクロカーネルベースの OS を目標に開発されています。最近は NetBSD のアプリケーションが移植できるほどの UNIX ライクな環境に仕上がっています。

Intel ME という機能を実現するチップセットで使われていることで有名です。

本書では，MINIX3 も MINIX3 より前のバージョンも含めて「MINIX」と呼ぶことにします。

■ Zircon（Fuchsia）

Zircon は，Google が開発しているオペレーティングシステム Fuchsia（フューシャ）のカーネルです。

ソースコードはオープンソースとして公開されています。カーネルでは珍しく C++ で書かれています。Little Kernel という小さなカーネルが基になったそうですが，その面影は見えないほどがっつり高機能化されています。ただ，筆者的には不必要に複雑になりすぎているコードであるように感じます。Mach と同じ轍を踏むのか，現代の高速な CPU では性能は問題にならないという話になるのか興味深い試みです。

■ QNX

クローズドソースなので本書では解説しませんが，名実とも世界を支えるマイクロカーネルとして QNX を忘れるわけにはいきません。現在は BlackBerry が持っているマイクロカーネルベースの OS で，ミッションクリティカルな組込みシステムに使われているそうです。

ソースコードは公開されていませんが，ドキュメントは公開されているので気になる人はぜひ読んでみましょう。商用利用されていることもあって，ドキュメントを読んだだけですが非常に良く考えられている設計だと筆者は感じています。

1.6 歴史

マイクロカーネルの歴史を少しだけ見てみましょう。ソースが Wikipedia ではありますが，マイクロカーネルの概念が最初に現れたのは 1969 年に公開された RC4000 マルチプログラミングシステムという OS だそうです。デンマークの Regnecentralen 社で開発された RC4000 というコンピュータ向けの OS のようで「メッセージパッシングによる OS 機能の分離」というまさにマイクロカーネルの考え方が取り入られていたようです。1969 年といえば，ちょうど UNIX の誕生と同時期のようですね。

1970 年代のことは筆者は全く分かりませんが，マイクロカーネル関連の研究はあったようです。

1985 年には Mach が誕生しました。第 1 世代マイクロカーネルの筆頭です。性能は悪いのですが，持っている機能は面白いということで一定の地位はあったようです。ちなみに，MINIX の公開は 1987 年で QNX の登場は 1982 年です。Mach より QNX の方が早かったというのは少し驚きです。

[*6] ネットワークや分散システムの分厚い本の著者としても有名ですね。

1993 年に，第 2 世代マイクロカーネルの L4 が登場します。「性能重視に作ればば性能のよいマイクロカーネルが作れる」ということを L4 は証明しました。Mach より（最大）22 倍速いというのですからびっくりです。第 2 世代は「カーネルの極小性による性能向上」がキーポイントでしょう。

そして，seL4 に代表される最近の第 3 世代マイクロカーネルのキーポイントは「信頼性とセキュリティの向上」です。ケイパビリティ（capability）ベースのアクセス制御，カーネルの形式検証など Trusted Computing Base の一部としての側面が色濃く見られます。

2 Resea 入門

本書では，筆者が趣味で作っているマイクロカーネルベースの OS「Resea（りーせあ）」を用いて，マイクロカーネルの仕組みを具体的なソースコードを用いて解説していきます。

本章では，Resea の概要やビルド・実行方法等を説明します。

2.1 なぜ Resea なのか

マイクロカーネルの解説に Resea を使うのは，筆者の自己満足以外にちゃんとした理由があります。

まず，既存のマイクロカーネルは開発がしにくい点です。使い方の分からない謎の独自ビルドシステムが使われていたり，ビルドできても動かなくなっていたり，そもそもビルドできなくなっていたりと，作りっぱなしでメンテナンスされていないケースが多いのです。Fuchsia といった実用性を重視するカーネルはよくメンテナンスされていますが，実用性がある分，複雑かつ規模が大きいため勉強のために読むのは少々根気がいります。

次に，カーネルのソースコードが公開されていても，肝心のユーザランドの実装がないという問題です。ファイルシステムやデバイスドライバがカーネルの機能をどう使っているのか分からないのです。マイクロカーネルベースの OS ではユーザランドのサーバたちが主役なので，ユーザランド実装を読むことができないのは致命的です。

最後に，研究 OS でよくあることですがコードがあまり読みやすくないのです。色んなラッパー関数や，独特な命名規則，自動生成される IPC スタブのせいで処理の流れが掴みづらい傾向にあります。

これらの問題を念頭に，以下の点に注意して Resea の設計・実装をしました。

- シンプルで理解しやすいコードになるように「良い妥協」をした設計をする。
- ユーザランドの実装もきちんと用意する。
- ビルド環境を簡単に用意できるようにする。
- IPC スタブジェネレータのようなソースコードの自動生成をしない。

Resea は，L4 や MINIX に比べて性能や信頼性は劣ります。しかし，カーネルのコア部分が C 言語で 3000 行以下に収まるほど単純なので，マイクロカーネルへの第一歩には向いています。

付録として，本書の最後にカーネルのコア部分のソースコードを掲載しています。気になるところをぱらぱらとめくりながら読んでみてください。

2.2 Resea の機能

Resea は現在以下の機能を持っています。マイクロカーネルの参考にしたり，ハックして遊んだりするには十分な機能を備えています。

- マイクロカーネル
  - システムコールは 4 つのみ（`ipc`，`taskctl`，`irqctl`，`klogctl`）
  - x86_64 CPU 対応
  - マルチプロセッサ対応
- ユーザランド
  - メモリ管理サーバ
  - キーバリュー・ストア（KVS）サーバ（ファイルシステムの代わり）
  - TCP/IP サーバ
  - ネットワークカードドライバ（e1000）
  - キーボードドライバ（PS/2）
  - シェル
- 独自ライブラリ
  - `printf`，`malloc`，`...`

2.3 開発環境のセットアップ

Resea のビルドには macOS もしくは Linux 環境が必要です。ビルドには新しめの LLVM の仲間たち（LLVM/Clang/LLD 9.x 以降）が必要です。

■ macOS

Homebrew で以下のようにツールチェインをインストールしてください。

```
$ brew install llvm python qemu bochs i386-elf-grub xorriso
```

■ Linux

ここでは，Ubuntu 20.04 の例でのセットアップを解説します。以下のようにパッケージをインストールしてください。

```
$ apt install llvm clang lld python3 qemu-system-x86 bochs grub2 xorriso \
              git make
```

パッケージをインストールしたら準備完了です。さっそくビルドしてみましょう。

2.4 ビルド・実行

Resea のソースコードは GitHub（`https://github.com/nuta/resea`）で公開しています。

```
$ git clone https://github.com/nuta/resea
$ cd resea
$ make
```

`make` コマンドはデフォルトで `build/resea.elf` にマルチブート方式のカーネルイメージを生成します。他にも，ISO ディスクイメージのビルドや QEMU（エミュレータ）上での実行を簡単に行なえます。

- `make iso`
  - ISO ディスクイメージをビルド
- `make run`
  - QEMU 上で実行
- `make run GUI=y`
  - QEMU 上で実行（QEMU の GUI を有効化）
- `make run SMP=4`
  - QEMU 上で実行（4 つの CPU をエミュレート）
- `make bochs`
  - Bochs エミュレータ上で実行

2.5 補足説明

本節では，Resea のソースコードを読む上で役に立つトピックを説明します。

■ ディレクトリ・ファイル構成

Resea のソースコードは主に以下のように構成されています。

- `kernel/`: カーネル
- `kernel/arch/`: カーネル（ハードウェア抽象化レイヤ）
- `libs/common/`: カーネル・ユーザランド共通ライブラリ（型定義，リスト・文字列処理など）
- `libs/std/`: ユーザランド標準ライブラリ
- `servers/`: ユーザランドプログラムたち（TCP/IP，デバイスドライバなど）
- `tools/`: 雑多なビルド・デバッグスクリプトたち

■ Big Kernel Lock

Linux のようなマルチプロセッサに対応したカーネルを読んだことがある方なら Resea を読むときに違和感を感じるかもしれません。マルチプロセッサに対応していると銘打っているのに，ロックが見当たらないのです。

マルチスレッド対応カーネルの場合，複数の CPU が同じ資源（プロセス構造体など）を一度にいじらないように，いたるところでロックを取っています。一般的には，ロックは細かくとる（fine-grained locking）のが性能向上によいとされています。

Resea では，細かいロックの代わりにロックを 1 つだけ（Big Kernel Lock）使います。システムコールや割り込みでカーネル空間に入るときに，そのロックを最初にとります。ロックを確保できる CPU は同時に 1 つのみなので，カーネルのコードが実行されている最中はその CPU 以外はロックが解放されるまで待っています。つまり，ユーザランドのプログラムは並列実行しますがカーネルは並列処理しません。Giant Lock だとか Big Kernel Lock と呼ばれる手法です。

どうしてそんなことをするのかというと，実装が非常に簡単になるからです。きちんとロックをとるのは大抵の人間には相当難しいことです。並列処理のバグは処理や割り込みのタイミングなどで「なぜかたまに起きる謎のバグ」として発生するので本当に厄介です。Big Kernel Lock 方式では，1 つのロックをきちんと扱うだけで済むので，遥かに実装・デバッグが楽になります。[*1]

Resea のソースコードを読む際は，割り込みや他の CPU の影響を受けない（つまり，普通のシングルスレッドプログラミングと同じ）ことを念頭においてください。

■ 双方向連結リスト

Resea のソースコードの随所で，双方向連結リストがリストまたはキューとして使われています。`list_t` がリスト，そして `list_elem_t` がリストの各要素の情報をもつ構造体です。`list_elem_t` はリストの要素になりうる各構造体（`struct task` など）に埋め込まれている，いわゆる intrusive な双方向連結リストです。

特に工夫のない単純な双方向連結リストで，以下の関数・マクロがよく使われます。実装は `libs/common/include/list.h` にあります。

- `list_push_back(list_t *list, list_elem_t *new_tail)`
  - 要素をリストの末尾に追加。
- `list_remove(list_elem_t *elem)`
  - 要素をリストから削除。
- `LIST_POP_FRONT(list, container, field)`
  - リストの先頭の要素を取り出すマクロ。リストが空の場合は `NULL` を返す。
- `LIST_FOR_EACH(elem, list, container, field)`
  - リストの各要素の走査。いわゆる `foreach` 文と同じもの。

`LIST_FOR_EACH` は実装が少し面白いので，興味のある人はソースコードを解読してみてください。

[*1] 面白いことに「Big Kernel Lock 方式でも（よく設計された）マイクロカーネルなら（現実的な状況では）十分性能が出る」ことを主張する論文があります。詳しくは参考文献をどうぞ。

■ printk

printk はカーネルログを出力するための関数です。make run すると，だらだらと流れていくログメッセージがそれです。名前は printf でも別によいのですが，Linux ではなぜか printk なのです。なんとなくかっこいいので Resea も printk という名前にしています。

Resea では printk をそのまま使わず，以下のマクロを代わりに使います。

▼ libs/common/include/print_macros.h

```
#define TRACE(fmt, ...)
#define INFO(fmt, ...)
#define DBG(fmt, ...)
#define WARN(fmt, ...)
#define OOPS(fmt, ...)
#define PANIC(fmt, ...)
```

大体マクロの名前から類推できると思います。DBG は printf デバッグするときに使います。ちょっと表示が強調されて便利です。

■ エラー値の扱い

Resea では，基本的に符号付き整数（signed int）を使っています。例えば，タスク ID は次のように定義されています。

▼ libs/common/include/types.h

```
typedef int tid_t;
```

タスク ID は負の数になり得ません。常に 1 以上の整数です。なぜ unsigned ではなく符号付きを使うのかというと，負の数でエラーを表現したいからです。では，エラー値の定義を見てみましょう。

▼ libs/common/include/types.h

```
typedef int error_t;
#define IS_ERROR(err) ((err) < 0)
#define IS_OK(err)    ((err) >= 0)

#define OK                 (0)
#define ERR_NO_MEMORY      (-1)
#define ERR_NOT_PERMITTED  (-2)
#define ERR_WOULD_BLOCK    (-3)
/* ... */
```

注目して欲しいのが IS_ERROR マクロです。負の数だった場合はエラーとして判定します。次のように使います。

```
tid_t tid_or_err = ipc_lookup("tcpip);
if (IS_ERROR(tid_or_err)) {
    return tid_or_err; // error!
}
```

ipc_lookup 関数は，指定された名前のタスクを検索し，その ID を返します。もし，タスクがない場合場合は，エラー（負値）を返します。成功時の戻り値（正の数）と異常時のエラー内容（負の数）を符号付き整数 1 つで表現するという，C 言語ではよく見られるテクニックです。

3 プロセス

「プロセス」はプログラムの実行を抽象化した概念です。OS によっては「タスク」と呼ばれています。シェルなどでプログラムを起動すると，カーネルはそれに対応するプロセスを生成し，実行を開始します。プロセスには「プログラムの実行単位」を表す 1 つ以上の「スレッド」が所属しており，カーネルは高速にスレッドを切り替え（コンテキストスイッチ）ながら，多くのプログラムが同時に動いているように見せかけています（プリエンプティブマルチタスク）。これらの概念は Linux や Windows といった主要な OS に取り入れられているので，馴染み深いものでしょう。

3.1 プロセス・スレッドの中身

「プログラムの実行を抽象化した概念」だとか「プログラムの実行単位」と説明されただけではプロセス・スレッドが何かを理解するのは難しいものです。そこで，Linux のようなモノリシックカーネルでは各プロセス・スレッドがどのようなデータを保持するのかを見てみましょう。

■ プロセスの中身（モノリシックカーネル）

カーネルに依りますが，各プロセスは大体以下のようなデータを持っています。

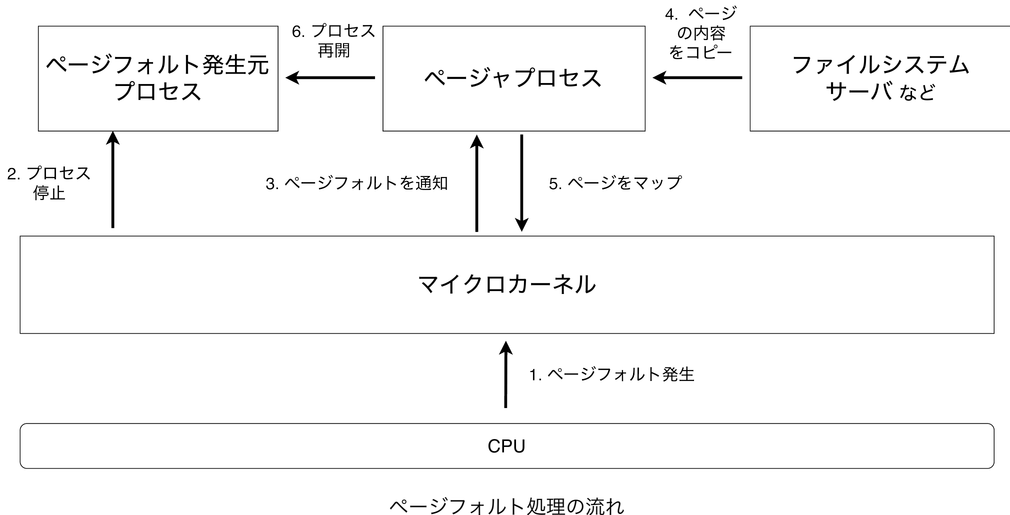
- プロセス ID
- 実行ユーザ
- 親プロセスへのポインタ
- 子プロセスのリスト
- ファイル記述子テーブル（開いているファイルの情報）
- アドレス空間の情報（ページテーブルなど）

ちなみに，プロセスの状態を保持する構造体を「Process Control Block（PCB）」と呼びます。

■ スレッドの中身（モノリシックカーネル）

各スレッドは大体以下のようなデータを持っています。

- スレッド ID
- 状態（実行可，スリープ中など）
- CPU レジスタ
- カーネルスタック（割り込み・システムコール処理に使う）



ページフォルト処理の流れ

- スケジューリングの情報（優先度など）
- 所属するプロセスへのポインタ

■ プロセスとスレッドの違い

プロセスとスレッドはどのような違いがあるのでしょうか。それは，プロセスはファイルやアドレス空間といった「資源」を持つのに対し，スレッドはレジスタといった「実行コンテキスト」を持つ点にあります。同じプロセスのスレッドたちは，ファイルやアドレス空間といった資源を上手く共有しながら，同時にコードを実行することでプログラムの高速化を図ります。

カーネルによっては，スレッドに対応せずプロセスがスレッドの機能を併せ持つことがあります。

3.2 マイクロカーネルのプロセス

マイクロカーネルにもプロセスやスレッドという概念があります。スレッドについては，従来のモノリシックカーネルと大差ありません。

しかし，プロセスは非常にシンプルになります。マイクロカーネルは「ファイル」や「ユーザ管理」といった概念を実装しません。ユーザランドでやります。そのため，マイクロカーネルのプロセスではそのような OS 固有の機能（OS パーソナリティ）は削ぎ落とされ，PCB に含まれるのはアドレス空間やプロセス間通信の管理情報ぐらいです。

3.3 ページフォルト処理

マイクロカーネルは「ファイル」の概念を持ちません。では，どうやって実行ファイルをメモリ上に展開するのでしょうか。マイクロカーネルはファイルシステムをカーネル内で実装する代わりに，ユーザランドのページャ（Pager）と呼ばれるプログラムに実行ファイルのロードを依頼します。具体的には，プログラムが実行する時に発生するページフォルトの情報をページャに通知します。

例えば，プログラムがまだマップされていないページにある関数を実行しようとすると，以下のよう

な流れで該当するページがロードされます。

1. **プログラム:** マップされていないメモリページにアクセス
2. **CPU:** ページフォルトが発生し，カーネルへ処理を移行
3. **カーネル:** 実行中プロセスのページャプロセスにメッセージを送信し，返信を待つ
4. **ページャ:** ページフォルトメッセージを受信し，物理ページの割り当て・データコピーを行って，メッセージを返信
5. **カーネル:** 指定されたページをマップしてプログラムの実行を再開

3.4 例外処理

プロセスは，システムコールを呼び出して自ら終了する場合に加えて，不正な実行によって異常終了する場合があります。不正な実行（ゼロ除算など）のことをここでは「例外（exception）」と呼ぶことにします。UNIX にも「Segmentation Fault」といった例外と同じ概念がありますね。

マイクロカーネルの中には，スレッドに登録されている「例外ハンドラ」と呼ばれる他のスレッドに例外の発生を通知する機能を持っている実装もあります。例外が起きたスレッドをどう扱うかをユーザランドで決められるようにすることで，柔軟性を向上します。

3.5 実装（Resea）

Resea ではプロセスのことを「タスク」と呼んでいます。簡単のため，スレッドは実装していません。各タスクは実行コンテキストを 1 つだけ持ちます。つまり，全てシングルスレッドです。

■ タスク構造体

では実際にどう実装されているのか見てみましょう。まずタスク構造体の定義です。

▼ kernel/task.h

```
struct task {
    struct arch_task arch;
    tid_t tid;
    int state;
    char name[TASK_NAME_LEN];
    caps_t caps;
    struct vm vm;
    tid_t pager;
    unsigned quantum;
    /* ... */
};
```

IPC 関連のメンバは後々説明するので省いています。まず目にとまるのは `arch` メンバです。Resea では，移植を簡単にするために CPU アーキテクチャに依存する部分（HAL: ハードウェア抽象化レイ

ヤ）を分離しています。arch_task 構造体には実行コンテキスト（CPU レジスタ・カーネルスタック）などが入っています。アドレス空間情報（ページテーブルなど）を持つ vm メンバもアーキテクチャ依存なので，これも同様に HAL 内で実装されています。

tid，state，name にはそれぞれタスクの ID，タスクの状態，名前が入っています。タスクの状態には，未使用・実行可能・メッセージ送信中・メッセージ待ちの大きく 4 つあります。

caps は，タスクが持つケイパビリティ（capability）を定義するビットフィールドで，各タスクができること（プロセス間通信，カーネルログの読み書き，I/O 命令など）を指定します。非常に簡素なセキュリティ機構です。

pager は，本章で説明した「ページャ」を担っているタスクを指定します。ページフォルトや例外，タスクの正常終了時にこのフィールドで指定されたタスクにメッセージが送られます。

quantum は，残り実行時間を保持するフィールドです。一定間隔（例: 1 ミリ秒）ごとに減っていき，この値が 0 になるとカーネルは次のタスクに切り替えます（プリエンプティブマルチタスク）。

■ タスク管理構造体

タスク管理には以下の変数が使われています。

▼ kernel/task.c

```
static struct task tasks[TASKS_MAX];
static list_t runqueue;
static struct task *irq_owners[IRQ_MAX];
```

tasks はその名の通り struct task の配列です。Resea では同時に実行できるタスクの数には上限があります（TASKS_MAX 個）。動的に割り当ててもよいのですが，簡単のために静的に割り当てて使っています。

runqueue は実行可能状態にあるタスクをもつキューです。各 CPU はここから次に実行するタスクを選択します。ただし実行可能でも，実行中のタスクは除きます。なぜ除くかというと，実装が若干シンプルかつ高速にできるからです。詳しくは「Benno scheduling」で検索してみましょう。

irq_owners はハードウェア割り込みを受けるタスクです。例えばキーボードを押すと，キーボードコントローラから CPU に割り込みが送られ，カーネルはこのテーブルで指定されているタスク（キーボードのデバイスドライバ）に通知 IPC（後述）を送ります。

■ タスクの生成

次に，タスクの生成を行う task_create 関数を見てみましょう。

▼ kernel/task.c

```
error_t task_create(struct task *task, const char *name, vaddr_t ip,
                    struct task *pager, caps_t caps) {
```

```
    if (task->state != TASK_UNUSED) {
        return ERR_ALREADY_EXISTS;
    }

    // Initialize the page table.
    error_t err;
    if ((err = vm_create(&task->vm)) != OK) {
        return err;
    }

    // Do arch-specific initialization.
    if ((err = arch_task_create(task, ip)) != OK) {
        vm_destroy(&task->vm);
        return err;
    }

    // Initialize fields.
    TRACE("new task #%d: %s", task->tid, name);
    task->state = TASK_CREATED;
    task->caps = caps;
    task->notifications = 0;
    task->pager = pager;
    /* ... */

    // Append the newly created task into the runqueue.
    if (task != IDLE_TASK) {
        task_set_state(task, TASK_RUNNABLE);
    }

    return OK;
}
```

基本的に，引数 task で指定されたタスク構造体を埋めるだけで，特に見どころがありません。初期化した後，task_set_state 関数で生成したタスクを実行可能状態にしてランキューに加えて終わりです。

■ タスクの終了

お次はタスクの終了処理です。Resea では，終了処理が 2 通りあります。正常・異常終了した場合と，強制終了された場合です。まずは，強制終了する task_destroy 関数を見てみましょう。

▼ kernel/task.c

```
error_t task_destroy(struct task *task) {
    ASSERT(task != CURRENT);
    ASSERT(task != IDLE_TASK);
```

```
    if (task->tid == INIT_TASK_TID) {
        TRACE("%s: tried to destroy the init task", task->name);
        return ERR_INVALID_ARG;
    }

    if (task->state == TASK_UNUSED) {
        return ERR_INVALID_ARG;
    }

    TRACE("destroying %s...", task->name);
    list_remove(&task->runqueue_next);
    list_remove(&task->sender_next);
    vm_destroy(&task->vm);
    arch_task_destroy(task);
    task->state = TASK_UNUSED;

    // Abort sender IPC operations.
    LIST_FOR_EACH (sender, &task->senders, struct task, sender_next) {
        notify(sender, NOTIFY_ABORTED);
        list_remove(&sender->sender_next);
    }

    for (unsigned i = 0; i < TASKS_MAX; i++) {
        /* ... */

        // Notify all listener tasks that this task has been aborted.
        if (task->listened_by[i]) {
            notify(task_lookup(i + 1), NOTIFY_ABORTED);
        }

        /* ... */
    }

    /* ... */
    return OK;
}
```

この中でやっているのは，各フィールドの開放と対象のタスクへメッセージを送ろうとしているタスク達への通知です。タスクが終了すると送り先がなくなりデッドロックしてしまうので，NOTIFY_ABORTED を通知して IPC 処理を中断させます。

次に，タスク正常・異常終了時の処理を行う task_exit 関数です。この関数は，タスクが自発的に終了した場合と，ゼロ除算など何らかの異常が起きた際に呼び出されます。

▼ kernel/task.c

```
NORETURN void task_exit(enum exception_type exp) {
    ASSERT(CURRENT != IDLE_TASK);
```

```
    // Tell its pager that this task has exited.
    struct message m;
    m.type = EXCEPTION_MSG;
    m.exception.task = CURRENT->tid;
    m.exception.exception = exp;
    ipc(CURRENT->pager, 0, &m, IPC_SEND | IPC_KERNEL);

    // Wait until the pager task destroys this task...
    CURRENT->state = TASK_EXITED;
    task_switch();
    UNREACHABLE();
}
```

CURRENT マクロは，現在実行中のタスクを保持しています。ここでは，終了したいタスクを指します。

終了処理といいつつ，単にページャにメッセージを送るだけです。メッセージを送った後は，task_switch を実行し他のタスクを実行します。task_switch が戻ってくる（CURRENT タスクが再開される）ことは永遠になく，システムコール経由でページャに強制終了されるのを待つだけです。

■ タスクの切り替え

最後にタスクの切り替えを行う task_switch 関数を見てみましょう。

▼ kernel/task.c

```
void task_switch(void) {
    stack_check();

    struct task *prev = CURRENT;
    struct task *next = scheduler(prev);
    next->quantum = TASK_TIME_SLICE;
    if (next == prev) {
        // No runnable threads other than the current one. Continue executing
        // the current thread.
        return;
    }

    CURRENT = next;
    arch_task_switch(prev, next);

    stack_check();
}
```

次に実行するタスクをスケジューラで選び，HAL で実装されている arch_task_switch 関数で実際の切り替え処理を行っています。arch_task_switch をここでは深く追いませんが，基本的にはペー

ジテーブル・カーネルスタックの切り替えとレジスタの保存・復元を行い，次のタスクの実行を開始します。次にこのタスク（CURRENT）が再開された時には，arch_task_switch から帰ってきたかのように実行が続きます。

stack_check 関数は，カーネルスタックを使い切っていないかチェックするバグ検出用の関数です。スタックを使い切ると，他のカーネルデータの破壊など奇想天外な振る舞いをしてしまいデバッグがかなり大変なので，所々でチェックをしています。

最後にスケジューラの処理です。

▼ kernel/task.c

```
static struct task *scheduler(struct task *current) {
    if (current != IDLE_TASK && current->state == TASK_RUNNABLE) {
        // The current task is still runnable. Enqueue into the runqueue.
        list_push_back(&runqueue, &current->runqueue_next);
    }

    struct task *next = LIST_POP_FRONT(&runqueue, struct task, runqueue_next);
    return (next) ? next : IDLE_TASK;
}
```

実行中のタスクがまだ実行可能なら（ブロックされていないなら）ランキューに戻した後，次に実行するタスクをランキューから取り出しています。実行するタスクがない場合（暇な時）は，代わりにアイドルタスク（IDLE_TASK）に移行します。アイドルタスクは，単に割り込み待ち状態で CPU を休ませるカーネル内の特別なタスクです。

見て分かる通り，まともなスケジューリングアルゴリズムを実装していません。単に順番に実行する素朴なラウンドロビン方式です。

3.6 実装（Mach）

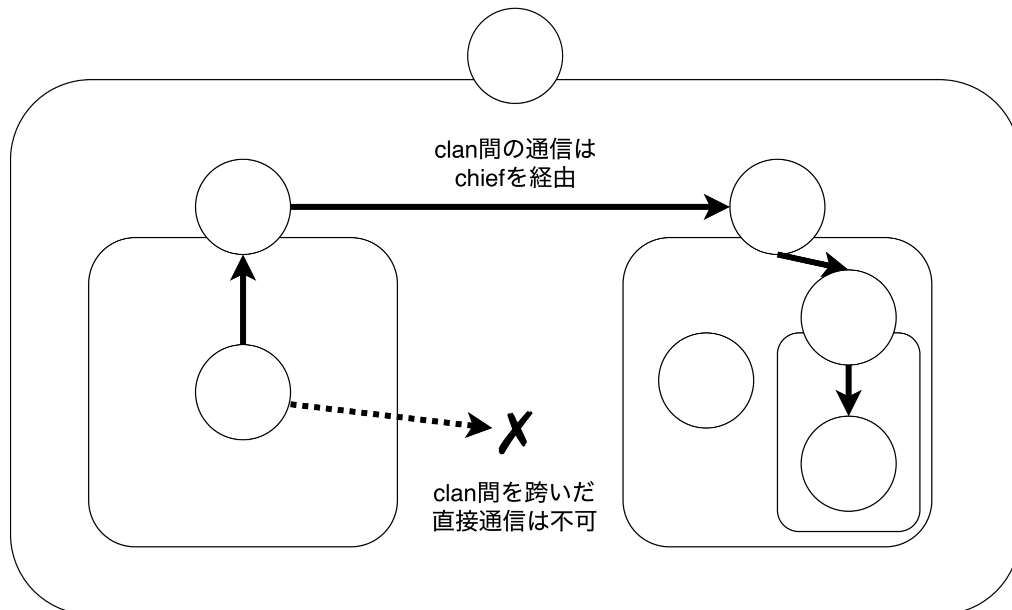
Mach ではプロセスにあたる概念は「タスク」と呼ばれています。各タスクには，スレッド，アドレス空間の情報，ポート（IPC のエンドポイント）などが含まれています。特に書くことがなくて困るほど普通です。

ページフォルト処理に関しては，アドレス空間の各範囲（例: ファイルがマッピングされている部分）にページャのポートが設定されています。ページフォルト時には，そのポートに対してページフォルト処理要求が飛びます。

3.7 実装（L4）

L4 では，プロセスの代わりに「タスク」という概念があります。各タスクは「アドレス空間と，そのアドレス空間を共有するスレッドたち」で構成されます。

アドレス空間のマッピングは IPC で「flex pages」という特別なデータを送ることで行います。 flex



Clans ＆ Chiefs の概要

pages で共有メモリを作ったりページへのアクセス権を移譲したりできます。
各スレッドには，以下の情報を持っています。

- 識別子（UID）
- CPU レジスタ（実行コンテキスト）
- アドレス空間
- ページフォルトハンドラ
- 例外ハンドラ

■ Clans ＆ Chiefs

初期の L4 では，「Clans & Chiefs」というセキュリティ機構がありました。同じ clan 内では自由に通信が可能ですが，clan の外と通信する場合は，その clan をもつ chief を経由し，アクセス制御等を実現します。いわば，chief はネットワークルータのような役割を果たしています。

便利そうではありますが，いろんなスレッドを経由して通信することによる IPC のオーバヘッドや，そもそもあまり活用されなかったそうで Clans & Chiefs は新しい L4 派生では消えていきました。

seL4 では，代わりにケイパビリティベースのセキュリティ機構を導入しています。「あるプロセスとの通信する権利」を **移譲できる** ようにすることで，clans & chiefs より効率よく・シンプル[*1]にアクセス制御を実現できていると筆者は感じています。

[*1] セキュリティ機構がシンプルであることは非常に大切です。いくら高機能であっても，複雑で難解なものは悲しいかな使われなくなります。筆者のお気に入りは OpenBSD の `pledge` と `unveil` です。とっても手軽に「プログラムが何をするのか」を開発者に意識させる仕組みになっています。https://youtu.be/bXO6nelFt-E

■ ページフォルト処理

L4 では，ページフォルトが起きると，実行中のスレッドに登録されたページャスレッドにメッセージを送信します。メッセージには，アクセスしようとした（ページフォルトの原因の）メモリアドレスと，ページフォルトが起きたプログラムカウンタが含まれています。ページャは特別なデータ（後述する flexpages）を含んだメッセージを返信することでページをマップします。

ゼロ除算といった例外発生時には，L4 カーネルはスレッドに登録されたスレッド ID （例外ハンドラ）にメッセージを送信します。例外ハンドラでは，プログラムカウンタを書き換えたり，スレッドを終了させたりします。

3.8 実装（MINIX3）

MINIX のプロセスは主に以下の情報を持っています。MINIX は UNIX ライクな OS ではありますが，プロセス構造体に UNIX 関連のフィールドが出てこないところからして，カーネル自体は純粋なマイクロカーネルの気質を感じますね。

- プロセス ID
- CPU レジスタ（実行コンテキスト）
- ページテーブルのアドレス
- スケジューリング情報（優先度など）
- メッセージバッファ

ページテーブルの実体は，カーネルではなく VM サーバが管理しています。

■ ページフォルト処理

MINIX のプロセス構造体にはページフォルトハンドラのフィールドがありません。代わりに，ハードコードされたサーバ（`VM_PROC_NR`）にメッセージを送信します。MINIX はカーネル単体ではなくユーザランド含めて 1 つの OS として開発されていることもあってか，カーネル部分は L4 のように極端に柔軟ではありません。このようにハードコードされているところが見られます。柔軟さよりシンプルさ[*2]を重視しているのでしょう。

▼ minix/kernel/arch/i386/exception.c

```
/* tell Vm about the pagefault */
m_pagefault.m_source = pr->p_endpoint;
m_pagefault.m_type   = VM_PAGEFAULT;
m_pagefault.VPF_ADDR = pagefaultcr2;
m_pagefault.VPF_FLAGS = frame->errcode;

if ((err = mini_send(pr, VM_PROC_NR,
```

[*2] 柔軟さを犠牲にしてでもシンプルさをとるというのは，審美的には受け入れがたいのですが非常に大切なことの 1 つです。「worse is better」で検索してみましょう。

```
                    &m_pagefault, FROM_KERNEL))) {
        panic("WARNING: pagefault: mini_send returned %d\n", err);
    }
```

MINIX ではページャからメッセージの返信をまたず，プロセスはページフォルト状態として停止したままになります。

VM サーバでは，ページフォルトのメッセージを受け取ると，アクセスされたページの存在等をチェックした後にページテーブルを更新します。更新後，ページフォルト状態として停止しているプロセスを，システムコールを使って実行を再開させます。

▼ minix/servers/vm/pagefaults.c

```
    /* Pagefault is handled, so now reactivate the process. */
    if((s=sys_vmctl(ep, VMCTL_CLEAR_PAGEFAULT, 0 /*unused*/)) != OK)
        panic("do_pagefaults: sys_vmctl failed: %d", ep);
```

3.9 実装（Zircon）

Zircon のプロセスには，以下のデータが入っています。

- 名前（デバッグ用）
- アドレス空間
- スレッドのリスト
- ハンドルのテーブル

スレッドは，実行コンテキスト（CPU レジスタ），カーネルスタック，スケジューリング情報（優先度など）などが入っています。

L4 や MINIX とは対照的に，マイクロカーネルといえど Zircon の機能は盛りだくさんで以下のような様々なカーネルオブジェクトが提供されています。

- チャネル（メッセージパッシングのエンドポイント）
- ソケット
- FIFO
- プロセス，スレッド，ジョブ
- 仮想メモリ関連
- Futex

カーネルオブジェクトは「ハンドル」を使ってアクセスします。ちょうど UNIX でいうところのファイル記述子のようなものです。

■ ページフォルト処理

Zircon でもページャの概念があります。ページフォルトが起きると以下のメソッドが呼ばれます。

▼ zircon/kernel/vm/vm_aspace.cc

```
zx_status_t VmAspace::PageFault(vaddr_t va, uint flags) {
  /* ... */

  zx_status_t status = ZX_OK;
  PageRequest page_request;
  do {
    {
      // for now, hold the aspace lock across the page fault operation,
      // which stops any other operations on the address space from moving
      // the region out from underneath it
      Guard<fbl::Mutex> guard{&lock_};

      status = root_vmar_->PageFault(va, flags, &page_request);
    }

    if (status == ZX_ERR_SHOULD_WAIT) {
      zx_status_t st = page_request.Wait();
      if (st != ZX_OK) {
        return st;
      }
    }
  } while (status == ZX_ERR_SHOULD_WAIT);

  return status;
}
```

どうやらページフォルト処理は `root_vmar_->PageFault()` で行われ，場合によっては `ZX_ERR_SHOULD_WAIT` が返ってきて待つ（ブロックする）ようです。

その後，カーネルを巡り巡って次のメッセージがページャへ送られます。

```
typedef struct zx_packet_page_request {
    uint16_t command;
    uint16_t flags;
    uint32_t reserved0;
    uint64_t offset;
    uint64_t length;
    uint64_t reserved1;
} zx_packet_page_request_t;
```

ページャはこのメッセージを受け取ると，[offset，offset + length) の範囲を zx_pager_supply_pages システムコールを使ってセットします。

```
zx_status_t zx_pager_supply_pages(zx_handle_t pager,
                                  zx_handle_t pager_vmo,
                                  uint64_t offset,
                                  uint64_t length,
                                  zx_handle_t aux_vmo,
                                  uint64_t aux_offset);
```

このシステムコールは aux_vmo で指定される空間から pager_vmo へ指定されたページ列を移動します。zx_pager_supply_pages でページフォルトが起きたページがセットされると，該当する PageRequest でイベントが発火し，ページフォルトから復帰するようです。

MINIX ではユーザランドでページテーブルを管理しているので，カーネルを呼び出すのはプロセスを復帰させるだけでした。Zircon では対照的にカーネルがページテーブルを管理しているので，カーネルにページを埋めてもらうためのシステムコール（zx_pager_supply_pages）が必要なのです。

4　システムコール

本章では，マイクロカーネルはどのような機能をシステムコールを介して提供しているのかを解説します。ただし，プロセス間通信（IPC）は次章で解説します。

■ マイクロカーネルのシステムコール

マイクロカーネルに依りますが，少なくとも以下のシステムコールを提供しています。

- プロセス・スレッド管理
- タイマー
- メモリページのアンマップ
- プロセス間通信

■ プロセス・スレッド管理

プロセス・スレッドの作成・削除はカーネルにやってもらう必要があります。

また，実行するスレッドを決定する「スケジューラ」のパラメータ（実行優先度など）を設定するインタフェースを備えたカーネルも見られます。

■ タイマー

カーネルが実行するスレッドを高速に（数十ミリ秒間隔）切り替えていくプリエンプティブマルチタスクを実現するために，マイクロカーネルはタイマーデバイスのドライバを備えています。

せっかくタイマーが使えるので，マルチタスクの実装以外にもタイマーを活用することが多いです。例えば，IPC タイムアウト機能です。

■ メモリページのアンマップ

マイクロカーネルの中には，メモリページ単位でデータを送る機能を IPC が持っていることがあります。

受信したメモリページを開放する手法としては 2 つ考えられます。1 つは，メモリ管理をしているサーバに開放したいメモリページを送信（移譲）することで自身のアドレス空間から削除するという方法です。もう 1 つは，専用のシステムコールを導入する手法です。

4.1 カーネルサーバ

ここまで解説したカーネルの機能を，システムコールではなく「カーネル内サーバとの IPC」として提供するカーネルも考えられます。

カーネルサーバとして実装しシステムコールを IPC 関連に限定することで，attack surface の縮小と 一貫性の向上が見込まれます。

4.2 実装（Resea）

ではどのようにシステムコールが実装されているのか Resea で雰囲気をつかみましょう。Resea は，以下の 4 つのシステムコールを提供しています。

▼ libs/std/include/std/syscall.h

```
error_t ipc(tid_t dst, tid_t src, struct message *m, unsigned flags);
tid_t taskctl(tid_t tid, const char *name, vaddr_t ip, tid_t page, caps_t caps);
error_t irqctl(unsigned irq, bool enable);
int klogctl(char *buf, size_t buf_len, bool write);
```

`taskctl` はタスクの作成・削除，`irqctl` は割り込み管理，`klogctl` はカーネルログの読み書きを行うシステムコールです。`ipc` システムコールについては次章で解説します。

では実装を見ていきましょう。まずは，システムコールのエントリーポイントです。

▼ kernel/syscall.c

```
uintmax_t handle_syscall(uintmax_t syscall, uintmax_t arg1, uintmax_t arg2,
                         uintmax_t arg3, uintmax_t arg4, uintmax_t arg5) {
    stack_check();

    uintmax_t ret;
    switch (syscall) {
        case SYSCALL_IPC:
            ret = (uintmax_t) sys_ipc(arg1, arg2, arg3, arg4);
            break;
        case SYSCALL_TASKCTL:
            ret = (uintmax_t) sys_taskctl(arg1, arg2, arg3, arg4, arg5);
            break;
        case SYSCALL_IRQCTL:
            ret = (uintmax_t) sys_irqctl(arg1, arg2);
            break;
        case SYSCALL_KLOGCTL:
            ret = (uintmax_t) sys_klogctl(arg1, arg2, arg3);
            break;
        default:
            return ERR_INVALID_ARG;
```

```
    }

    stack_check();
    return ret;
}
```

システムコール命令が実行されると，アーキテクチャ依存の処理が行われた後にこの handle_syscall 関数が呼び出されます。単に syscall で指定されたシステムコール番号に従って実装を呼び出しているだけです。

■ タスクの生成・削除

次にタスクの生成・削除など，タスク関連の処理をすべて担う taskctl システムコールです。

▼ kernel/syscall.c

```
static tid_t sys_taskctl(tid_t tid, userptr_t name, vaddr_t ip, tid_t pager,
                         caps_t caps) {
    // Since task_exit(), task_self(), and caps_drop() are unprivileged, we
    // don't need to check the capabilities here.
    if (!tid && !pager) {
        task_exit(EXP_GRACE_EXIT);
    }

    if (pager < 0) {
        // Do caps_drop() and task_self() at once.
        CURRENT->caps &= ~caps;
        return CURRENT->tid;
    }

    // Check the capability before handling privileged operations.
    if (!CAPABLE(CAP_TASK)) {
        return ERR_NOT_PERMITTED;
    }

    // Look for the target task.
    struct task *task = task_lookup(tid);
    if (!task || task == CURRENT) {
        return ERR_INVALID_ARG;
    }

    if (pager) {
        struct task *pager_task = task_lookup(pager);
        if (!pager_task) {
            return ERR_INVALID_ARG;
        }

        // Create a task.
```

```
        char namebuf[TASK_NAME_LEN];
        strncpy_from_user(namebuf, name, sizeof(namebuf));
        return task_create(task, namebuf, ip, pager_task, CURRENT->caps & caps);
    } else {
        // Destroy the task.
        return task_destroy(task);
    }
}
```

taskctl は，パラメータの組み合わせによって行う処理を変えます。なぜ別々のシステムコールに分けないのかというと，一緒にした方がコードの量が減るからです。とはいえ，大した減量にはならないので好みの問題です。

ここで注目してほしいのは，引数 name の型である userptr_t です。その名が示す通り，これはユーザプロセスから渡されたポインタです。ここでは，新しく作るタスクの名前の文字列へのポインタが入っています。カーネルは基本的にユーザのアドレス空間にもアクセスできるので，strncpy でも動きます。なぜ strncpy_from_user というものを新たに加えるのでしょうか。それは，ユーザから渡されるポインタは次に説明するように，別途注意が必要な厄介な代物であるからです。

■ ユーザポインタの処理

システムコールにユーザから渡されるポインタは memcpy や strncpy で単純にコピーしてはいけません。以下の点に注意する必要があります。

- ポインタがユーザ空間を指していること。システムコール処理はカーネルが実行しているので，ポインタがカーネル空間を指している場合に例外（ページフォルト）が発生せず，カーネルの内部データが漏れる脆弱性の元となってしまう。
- ポインタがユーザ空間を指していても，コピー中にページフォルトが起きる（まだページがマップされていない）場合がある。その場合は，先にページャを呼び出してページのマッピングをしてもらう必要がある。

後者はなんだか実装が面倒くさそうで仕方がないのですが，ちょっとした実装テクニックを使うと非常に簡単に実装できます。ここではユーザポインタから指定バイト分メモリコピーを行う memcpy_from_user 関数を見てみましょう。

```
static void memcpy_from_user(void *dst, userptr_t src, size_t len) {
    if (is_kernel_addr_range(src, len)) {
        task_exit(EXP_INVALID_MEMORY_ACCESS);
    }

    arch_memcpy_from_user(dst, src, len);
}
```

なんだか意味ありげな if 文があります。アドレスがカーネル空間を指す（is_kernel_addr_range）なら，システムコールを呼び出したタスクを異常終了させるようです。

これだけだとよく分からないので，arch_memcpy_from_user の実装も見てみましょう。

▼ kernel/arch/x64/trap.s

```
arch_memcpy_from_user:
    mov rcx, rdx
    cld
usercopy1:
    rep movsb
    ret
```

cld のコピーの向きを指定する命令で，rep movsb は高速にメモリコピーを行う命令です。普通の memcpy と同一の実装を行っています。とてもシンプルですね。しかし，コピー元のページがすでにマップされているかをチェックしていません。このままでは，rep movsb の部分でページフォルトが起きてしまいます。直感的には，メモリコピーの前にページがすでにマップされているかチェックし，されていなければページャで埋めておく必要があるように思えます。しかし，これだけでもコピー中に起きるページフォルトを華麗に処理できます。そのトリックは usercopy1 ラベルにあります。ここで，ページフォルトハンドラを見てみましょう。

■ ページフォルト処理

ユーザポインタの取り扱いを説明するついでに，ページフォルト処理を見てみましょう。ページフォルトが発生した際にはレジスタの保存を行った後に，次のページフォルトハンドラが呼ばれます。

▼ kernel/arch/x64/interrupt.c

```
void x64_handle_interrupt(uint8_t vec, struct iframe *frame) {
    /* ... */

    switch (vec) {
        case EXP_PAGE_FAULT: {
            vaddr_t addr = asm_read_cr2();
            pagefault_t fault = frame->error;
            uint64_t ip = frame->rip;

            /* ... */

            if (ip == (uint64_t) usercopy1 || ip == (uint64_t) usercopy2) {
                TRACE("page fault in usercopy, handling as user's fault");
                fault |= PF_USER;
                needs_unlock = false;
            }
```

```
            handle_page_fault(addr, fault);
            break;
        }
```

x64_handle_interrupt 関数は，例外の他にハードウェア割り込み（タイマーやキーボードなど）も一緒に受け付けます。vec は割り込みの種類，iframe は例外・割り込みが発生したときのレジスタが入っています。

ここで arch_memcpy_from_user のトリックを見てみましょう。arch_memcpy_from_user の中でページフォルトが起きうる命令は usercopy1 で示される rep movsb のみです。そこで，ページフォルトが usercopy1 で起きた場合に fault |= PF_USER とすることで，ユーザ空間で起きたページフォルトを同じ処理を行います。要はページがマップされているかをコードで愚直にチェックせず，試しにコピーしてみてダメだったら，通常のページフォルトと同じように処理して arch_memcpy_from_user に戻ってコピーを続けるわけです。

次にページフォルト処理本体（handle_page_fault）を見てみましょう。

▼ kernel/memory.c

```
void handle_page_fault(vaddr_t addr, pagefault_t fault) {
    // Ask the associated pager to resolve the page fault.
    vaddr_t aligned_vaddr = ALIGN_DOWN(addr, PAGE_SIZE);
    paddr_t paddr;
    pageattrs_t attrs;
    if (CURRENT->tid == INIT_TASK_TID) {
        paddr = init_task_pager(aligned_vaddr, &attrs);
    } else {
        paddr = user_pager(aligned_vaddr, fault, &attrs);
    }

    vm_link(&CURRENT->vm, aligned_vaddr, paddr, attrs);
}
```

ページフォルトが起きたタスクが最初のタスク（INIT_TASK_TID）である場合は，それ専用のページャ（init_task_pager）を呼び出します。その他のタスクでは次の user_pager 関数でページのマップ先の物理アドレスを返してもらいます。

▼ kernel/memory.c

```
static paddr_t user_pager(vaddr_t addr, pagefault_t fault, pageattrs_t *attrs) {
    struct message m;
    m.type = PAGE_FAULT_MSG;
    m.page_fault.task = CURRENT->tid;
    m.page_fault.vaddr = addr;
    m.page_fault.fault = fault;
```

```
    error_t err = ipc(CURRENT->pager, CURRENT->pager->tid, &m,
                      IPC_CALL | IPC_KERNEL);
    if (IS_ERROR(err)) {
        WARN("%s: aborted kernel ipc", CURRENT->name);
        task_exit(EXP_ABORTED_KERNEL_IPC);
    }

    // Check if the reply is valid.
    if (m.type != PAGE_FAULT_REPLY_MSG) {
        WARN("%s: invalid page fault reply (type=%d, addr=%p, pager=%s)",
             CURRENT->name, m.type, addr, CURRENT->pager->name);
        task_exit(EXP_INVALID_PAGE_FAULT_REPLY);
    }

    *attrs = PAGE_USER | m.page_fault_reply.attrs;
    return m.page_fault_reply.paddr;
}
```

ipc 関数でページフォルト処理要求（PAGE_FAULT_MSG）をページャに送り，返信が来るまで待っているだけです。

■ 割り込み処理

次に割り込みが起きたらどうなるかを見てみましょう。割り込みが発生すると，ページフォルト時と同じようにレジスタを保存し，x64_handle_interrupt を通って handle_irq が呼ばれます。

▼ kernel/task.c

```
void handle_irq(unsigned irq) {
    if (irq == TIMER_IRQ) {
        // Handles timer interrupts. The timer fires this IRQ every 1/TICK_HZ
        // seconds.

        // Handle task timeouts.
        if (mp_is_bsp()) {
            for (int i = 0; i < TASKS_MAX; i++) {
                struct task *task = &tasks[i];
                if (task->state == TASK_UNUSED || !task->timeout) {
                    continue;
                }

                task->timeout--;
                if (!task->timeout) {
                    notify(task, NOTIFY_TIMER);
                }
            }
        }
```

```
        // Switch task if the current task has spend its time slice.
        DEBUG_ASSERT(CURRENT->quantum > 0);
        CURRENT->quantum--;
        if (!CURRENT->quantum) {
            task_switch();
        }
    } else {
        struct task *owner = irq_owners[irq];
        if (owner) {
            notify(owner, NOTIFY_IRQ);
        }
    }
}
```

タイマー処理か否かで処理が別れています。タイマー以外の時の割り込み処理は，irq_owners テーブルを見て，タスクが登録されていれば通知（NOTIFY_IRQ）を送るだけです。なお，irq_owners テーブルへの登録は irqctl システムコールを通じて行います。

タイマー割り込みの場合は大きく 2 つの処理を行います。1 つめは各タスクがセットしているタイマーを更新し，タイムアウトになったらタスクに通知（NOTIFY_TIMER）します。このハンドラは全ての CPU で定期的に呼び出されるため，タイマー更新処理を重複して行わないよう mp_is_bsp() をチェックして最初の CPU でのみ処理するようにしています。

もう 1 つのタイマー処理は，タスクの切り替えです。実行中タスクの quantum をデクリメントし，0 に達したらタスクの切り替えを行います。

4.3 実装（Mach）

kern/syscall_sw.c に以下のシステムコールが定義されています。大きなカーネルである割には，システムコールはすっきりしている印象を受けますね。

```
evc_wait          // 指定されたイベントを待つ（デバイスドライバ用らしい）
evc_wait_clear
mach_msg_trap     // メッセージの送受信
mach_reply_port
mach_thread_self  // スレッドのポートを返す
mach_task_self
mach_host_self
mach_print        // 文字列を表示（デバッグ用）
device_writev_request
device_write_request
swtch_pri
swtch
thread_switch
```

```
vm_map
vm_allocate
vm_deallocate
task_create
task_terminate
task_suspend
task_set_special_port
mach_port_allocate
mach_port_deallocate
mach_port_insert_right
mach_port_allocate_name
thread_depress_abort
```

4.4 実装（L4）

L4 の初期実装では[*1]以下 7 つのシステムコールのみ提供しています。1 つのシステムコールで複数の操作を一緒にやることで，コードサイズを減らそうという思想が垣間見えますね。

- `task_new`
  - タスクの作成など。対象のタスク ID，1 つ目のスレッドのプログラムカウンタ・スタックポインタ・ページャのスレッド ID など指定。
- `id_nearest`
  - 「Clans/Chiefs モデル」という L4 初期にあったセキュリティ機構に関連するシステムコール。
- `lthread_ex_regs`
  - CPU レジスタ（プログラムカウンタ・スタックポインタ）の変更。
- `thread_switch`
  - 他のスレッドに CPU 時間を受け渡す。`pthread_yield(3)` みたいなやつ。
- `thread_schedule`
  - スケジューラのパラメータの設定。
- `ipc`
  - メッセージの送信・受信。
- `fpage_unmap`
  - メモリページのアンマップ。

[*1] 派生カーネル（特に seL4）ではシステムコール体系が抜本的に変わっています。

4.5 実装（MINIX3）

MINIX のシステムコールは IPC 関連とカーネルコールに分かれています。IPC は次章で解説するので，ここではカーネルコールを見てみましょう。

カーネルコールは，普通のカーネルにおけるシステムコールと同じです。そこそこたくさんあるので，一部のみ抜粋します。

▼ minix/kernel/system.c

```
/* Process management. */
map(SYS_FORK, do_fork);              /* a process forked a new process */
map(SYS_EXEC, do_exec);              /* update process after execute */
map(SYS_CLEAR, do_clear);            /* clean up after process exit */
map(SYS_EXIT, do_exit);              /* a system process wants to exit */
map(SYS_PRIVCTL, do_privctl);        /* system privileges control */
/* ... */

/* Signal handling. */
map(SYS_KILL, do_kill);              /* cause a process to be signaled */
/* ... */

/* Memory management. */
map(SYS_MEMSET, do_memset);          /* write char to memory area */
map(SYS_VMCTL, do_vmctl);            /* various VM process settings */

/* Copying. */
map(SYS_UMAP, do_umap);              /* map virtual to physical address */
map(SYS_UMAP_REMOTE, do_umap_remote); /* do_umap for non-caller process */
map(SYS_VUMAP, do_vumap);            /* vectored virtual to physical map */

/* ... */
```

マイクロカーネルとはいえ, L4 と比べると多くの処理が入っています。fork といった UNIX っぽいものも見られますが, これはカーネル部分で必要な処理だけを行っているだけで, ファイル記述子テーブルのコピーといったサーバが担う処理は行いません。実装（minix/minix/kernel/system/do_fork.c）を読むとびっくりするシンプルさです。

4.6 実装（Zircon）

Zircon にはたくさんのシステムコールが定義されています。ここでは，一部のみ掲載します。Fuchsia は幸いドキュメント[*2]がしっかりしているので，興味のある人はそちらをご覧ください。

[*2] https://fuchsia.dev/fuchsia-src/reference/syscalls

```
zx_channel_call          zx_object_signal          zx_channel_create
zx_object_signal_peer    zx_channel_read_etc       zx_object_wait_async
zx_channel_read          zx_object_wait_many       zx_channel_write_etc
zx_object_wait_one       zx_channel_write          zx_pager_create
zx_clock_adjust          zx_pager_create_vmo       zx_clock_create
zx_pager_detach_vmo      zx_clock_get              zx_pager_supply_pages
zx_fifo_create           zx_port_wait              zx_fifo_read
zx_process_create        zx_fifo_write             zx_process_exit
zx_handle_close          zx_task_suspend           zx_handle_duplicate
zx_task_suspend_token    zx_handle_replace         zx_thread_create
zx_interrupt_ack         zx_thread_exit            zx_interrupt_bind
zx_thread_read_state     zx_vmo_create_contiguous  zx_vmo_set_cache_policy
zx_vmo_create            zx_vmo_set_size           zx_vmo_create_child
zx_vmo_write             zx_vmo_create_physical    zx_vmo_get_size
...
```

Zircon は L4 や MINIX と比べると，大きめのマイクロカーネルです。「極限まで小さい」というよりは「十分に小さい」という印象を受けます。とはいえ「ファイル」という概念をカーネルで見かけない（つまりユーザランドで実装されている）のはマイクロカーネルらしさがありますね。

割り込み関連のシステムコール（`zx_interrupt_bind`）があるところから読み取れるように，L4 のように何でもかんでもメッセージパッシングで実現するのではなく，システムコールを別途用意しているところに思想の小さな違いが垣間見えます。

5 プロセス間通信（IPC）

プロセス間通信（IPC: Inter-Process Communication）は，プロセス間でデータのやり取りをするための仕組みです。Linux といった従来の OS には様々な機構が取り入れられています。パイプやソケット，共有メモリは皆さんにも馴染みの深いものでしょう。

大抵のマイクロカーネルは，IPC としてメッセージパッシングを採用しています。ファイルや共有メモリでもよいのですが[*1]，メッセージパッシングの方が柔軟で扱いやすいものです。

各プロセスは独立したアドレス空間を持つため，その間でデータのやり取りをするにはカーネルが何らかの仲介をする必要があります。例えば，メッセージパッシングならメッセージの相手先のアドレス空間へのコピーをカーネルが行う必要があります。

マイクロカーネルでは，デバイスドライバといった独立したプロセス同士が通信し合って OS の機能を提供します。モノリシックカーネルでは単なる関数呼び出しだった部分で IPC を置き換えます。IPC にはカーネルの仲介が基本的に必要です。そのため，IPC の設計はマイクロカーネルの性能を大きく左右し，またカーネル間の特徴の違いが垣間見える重要なポイントです。

本章では，メッセージパッシングによる IPC を解説します。

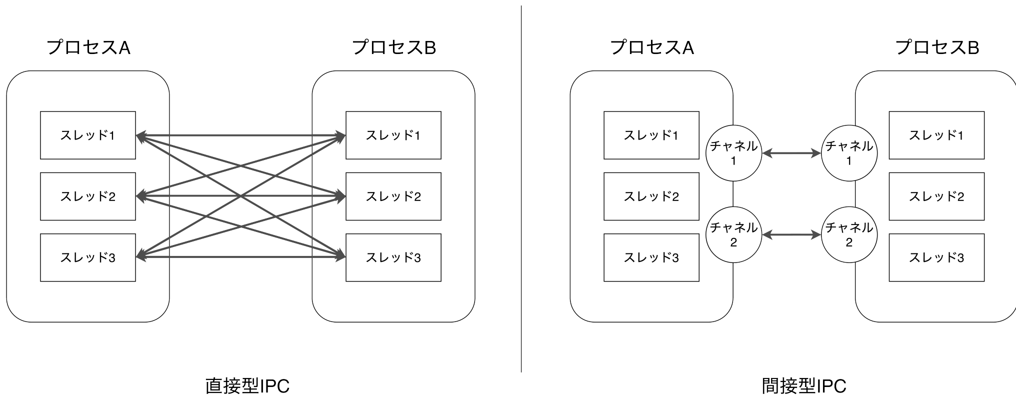
5.1 API

メッセージパッシングはプロセス間で「メッセージ」を送る仕組みです。メッセージの送信といっても多彩な設計が可能です。マイクロカーネルではメッセージパッシングが主な IPC の手段であることもあり，様々な機能を備えていることがあります。

メッセージパッシングのインタフェースとしてよく提供されているのは以下の API です。

- `Send(destination, message)`
  - メッセージの送信
- `Recv(source, message)`
  - メッセージの受信
- `Call(destination, message)`
  - メッセージの送信と受信（クライアントプロセスが使う）
- `ReplyRecv(destination, message)`
  - メッセージの送信と受信（サーバプロセスが使う）

[*1] Rust で OS を書く試みの Redox では，メッセージパッシングの代わりにファイルを使った抽象化を採用しています。https://redox-os.org/



直接型・間接型 IPC の概要。間接型 IPC では必ずチャネルを通して通信し，直接型 IPC では通信相手をスレッド ID で指定する。直接型 IPC では通信先を誰でも指定できるため，通信相手のアクセス制御を考慮する必要がある。

Send と Recv は必要不可欠のように感じますが，Call と ReplyRecv はなぜ必要なのでしょうか。それは，システムコールを呼び出す回数を減らすためです。どちらも Send と Recv を組み合わせることで実現できますが，システムコールの呼び出しは関数呼び出しより時間のかかる処理なのです。そこで，Call と ReplyRecv を導入することで 1 回のシステムコールで済ませてしまおうということです。

メッセージの送信先・受信元（`destination` と `source`）に何を指定するか，メッセージ（`message`）の構造，そして処理が同期的か非同期的かなど，IPC は様々な特徴が現れる部分です。

5.2 メッセージの内容

メッセージパッシングは単にデータのコピーをするだけではありません。効率的な通信を行うために，表現力の高い以下に挙げるようなメッセージ構造を持っていることがあります。

- **インラインデータ:** 単純にコピーされるデータ。
- **間接型データ:** 参照（メモリアドレス）を渡し，カーネルがその参照先のデータをコピーする。
- **メモリページ:** メモリページ単位でのメモリコピーを行う。データをバイト単位でコピーする代わりに，ページテーブルの書き換えを行う。
- **ハンドル:** チャネルといった，カーネルのオブジェクトの受け渡し。

5.3 間接型 IPC vs. 直接型 IPC

メッセージの送信先として何を使えば良いのでしょうか。真っ先に思い浮かぶのは，プロセスごとに複数の（ポートまたはチャネルと呼ばれる）メッセージボックスがあり，その ID を送信先として指定するという設計です。間接型 IPC（indirect IPC）と呼ばれます。[*2]

もう 1 つは，スレッド ID を指定する手法です。柔軟性がなく，スレッド ID がラップアラウンドした時を考慮する必要があります。しかし，チャネル ID から宛先のプロセス・スレッドを探す処理を省

[*2] 「indirect IPC」という言葉を「メッセージの間接型データ（後述）を含む IPC」という意味で用いられている場合もあります。

くことができるため，間接型IPCよりほんの少し高速になります。また，柔軟性がなくとも特定用途の小規模なOSならこれで十分です。直接型IPC（direct IPC）と呼ばれます。

5.4 クローズド受信とオープン受信

前述のCall操作とReplyRecv処理の違いはどこにあるのでしょうか。それは「どこからのメッセージなら受け取るか」が違います。

Call操作は「リクエストメッセージを送って，レスポンスメッセージを受け取る」処理です。レスポンスメッセージを受け取る際に，直接型IPCだと困ったことが起きる恐れがあります。それは，他の第三者スレッドがメッセージを送ってきた場合です。レスポンスの処理をする時に全く関係ないデータがやってくるケースを考えなくてはなりません。

そこで，クローズド受信（closed receive）という仕組みを導入します。クローズド受信では，受け付ける送信元をカーネルに指定し，他のスレッドからのメッセージをブロックします。Call操作は「リクエストメッセージの送り先からのメッセージのみを受け取る」というクローズド受信を行う処理です。

対するオープン受信（open receive）は「どこからでもメッセージを受け取る」という受信操作です。例えばサーバプロセスは「レスポンスメッセージを送信し，次のリクエストメッセージをどこからでも受け付ける」という処理をします。ReplyRecvはこの処理をまとめて行う操作です。

5.5 同期的IPC vs. 非同期IPC

メッセージを送信する際，宛先に受信するスレッドが現れるまで待つ（同期的IPC）か，受信スレッドを待たずメッセージをキューに入れて送信処理を完了する（非同期IPC）かの２つの流派があります。

同期的IPCでは，受信スレッドが受信状態でブロックしているので，カーネルのバッファを介さずに受信スレッドのレジスタやアドレス空間にメッセージを直接コピーしたり，スケジューラを呼ばず受信スレッドに直接コンテキストスイッチ（direct process switch）をしたりといった性能向上手法を実現できます。同期的IPCを「rendezvous-style IPC」とも呼ばれます。

非同期IPCでは，受信スレッドを待たずに実行を続けることができるという長所がありますが，同期処理を別に導入したり，メッセージキューが満杯になった時にどうするかといったことを考えたりする必要があります。

実装のしやすさの観点では，同期的IPCの方が実装・デバッグしやすいです。

5.6 通知（Notifications）

同期的IPCを採用するマイクロカーネルでは，同期的IPCの他に「通知（notiifcations)」と呼ばれる，UNIXのシグナルに似たシンプルな非同期IPCを備えていることがあります。

主な応用先は割り込み通知です。マイクロカーネルでは，割り込みの発生をユーザランドで動くデバイスドライバに通知する必要があります。非同期IPCではカーネルから送信されたメッセージとしてキューに追加すればよいのですが，同期的IPCでこれを愚直に実装すると，割り込みハンドラが受信スレッドを待たなければならなくなり，割り込みハンドラがブロックする事態に陥ってしまいます。

そこで，割り込みといった何らかの「イベント」が起きたことを非同期に通知するための機構を加え

ることでこの問題を解決します。

といっても，メッセージキューを持つのはメモリ割り当てが必要になるので，代わりにビットフィールドを使います。

5.7 タイムアウト

送信先がハングアップしているといった理由で，IPC 処理に時間がかかっている時に，処理を中断する「タイムアウト」機能を備えたマイクロカーネルもあります。プリエンプティブマルチタスクを実現するためにマイクロカーネルは基本的なタイマー処理を実装しているので，タイムアウト機能も簡単に加えることができます。この機能を使って，JavaScript でいう `setTimeout` や `setInterval` のような，タイマー機能をついでに実現できます。

便利そうではありますが，タイムアウトの時間はどれくらいの長さが適切なのでしょうか。正直，筆者は見当がつきません。個人的には，全く待たないか（ノンブロッキング処理），永遠に待つかのどちらかしか使わない気がします。

L4 カーネルでは，当初は IPC タイムアウト機能が導入されていましたが seL4 では，タイムアウト機能を取り入れませんでした。

5.8 IPC fastpath

マイクロカーネルのメッセージパッシングには「よくあるケース」があります。クライアントプロセスは，リクエストをサーバに「送信」してレスポンスを「受信」するという送受信（call）操作がよく使われます。ハンドルといった特別な処理が必要なものがメッセージに含まれるケースはまれで，大抵はインラインデータだけ入っています。また，宛先チャネルでは大抵スレッドが既に受信状態で待っています。

このようなよくあるケースに特化した IPC 実装（IPC fastpath）を加えることで，性能向上を図ります。

詳しくは Blackham らの *Correct, Fast, Maintainable – Choose Any Three! (APSys '12)* という論文によくまとまっています。

5.9 実装（Resea）

では Resea の IPC 実装を見てみましょう。Resea の IPC の仕様は以下の通りです。説明した大体の機能を備えています。

- 直接型・同期型 IPC
- 通知（Notifications）IPC
- IPC タイムアウト対応
- メッセージはインラインデータのみ

なお，メッセージは次のような構造を持っています。`type` はメッセージの種類，`src` は送信元タスク ID を指します。

▼ libs/common/include/message.h

```
/// Message.
struct message {
    int type;
    tid_t src;
    union {
        struct {
            notifications_t data;
        } notifications;

        struct {
            tid_t task;
            enum exception_type exception;
        } exception;

        struct {
            tid_t task;
            vaddr_t vaddr;
            pagefault_t fault;
        } page_fault;

        struct {
            paddr_t paddr;
            pageattrs_t attrs;
        } page_fault_reply;

        /* ... */
```

■ インタフェース

Resea では主に次のような IPC の API を提供しています。

```
error_t ipc_send(tid_t dst, struct message *m);
error_t ipc_send_noblock(tid_t dst, struct message *m);
error_t ipc_recv(tid_t src, struct message *m);
error_t ipc_call(tid_t dst, struct message *m);
error_t ipc_listen(tid_t dst);
```

ipc_call 関数では，送信バッファも受信バッファも同じところ（m）を使います。メモリの節約と実装をシンプルにするためです。

また，メッセージサイズは sizeof(struct message) 固定になっているため，メッセージの長さを指定するフィールドはありません。固定長にすることで，実装が少しすっきりします。

■ IPC システムコール

では実装を見ていきましょう。IPC 関連の操作は次の ipc システムコールで処理されます。

▼ kernel/syscall.c

```
static error_t sys_ipc(tid_t dst, tid_t src, userptr_t m, unsigned flags) {
    struct message buf;

    if (!CAPABLE(CAP_IPC)) {
        return ERR_NOT_PERMITTED;
    }

    if (flags & IPC_KERNEL) {
        return ERR_INVALID_ARG;
    }

    if (src < 0 || src > TASKS_MAX) {
        return ERR_INVALID_ARG;
    }

    struct task *dst = NULL;
    if (flags & (IPC_SEND | IPC_LISTEN)) {
        dst = task_lookup(dst);
        if (!dst) {
            return ERR_INVALID_ARG;
        }
    }

    if (flags & IPC_SEND) {
        memcpy_from_user(&buf, m, sizeof(struct message));
    }

    error_t err = ipc(dst, src, &buf, flags);
    if (IS_ERROR(err)) {
        return err;
    }

    if (flags & IPC_RECV) {
        memcpy_to_user(m, &buf, sizeof(struct message));
    }

    return OK;
}
```

この関数では主に引数のチェックとユーザ空間のバッファの読み書きを行っています。メッセージの送受信といった IPC 処理は全て次の sys_ipc 関数で行われます。

▼ kernel/ipc.c

```
error_t ipc(struct task *dst, tid_t src, struct message *m, unsigned flags) {
    if (flags & IPC_TIMER) {
        CURRENT->timeout = POW2(IPC_TIMEOUT(flags));
    }

    // Register the current task as a listener.
    if (flags & IPC_LISTEN) {
        dst->listened_by[CURRENT->tid - 1] = true;
        return OK;
    }

    // Send a message.
    if (flags & IPC_SEND) {
        /* ... */
    }

    // Receive a message.
    if (flags & IPC_RECV) {
        /* ... */
    }

    return OK;
}
```

sys_ipc は flags のビットフィールドの組み合わせで様々な処理を行います。

IPC_TIMER がセットされている場合，タイムアウト（timeout）を設定します。この timeout は，タイマー割り込み毎にデクリメントされていき，0 に達したらタスクに NOTIFY_TIMER を通知します。

IPC_LISTEN は，送信先タスクの「待ちタスクテーブル」に自身を追加します。送信先タスクが受信状態に入ると，このリストのタスクに通知が来ます。「送れる時にノンブロッキングにメッセージ送信をする」という処理を実現するために使われる機能です。

では IPC の主役，メッセージの送信処理を見てみましょう。

▼ kernel/syscall.c

```
    if (flags & IPC_SEND) {
        // Wait until the destination (receiver) task gets ready for receiving.
        while (true) {
            if (dst->state == TASK_RECEIVING
                && (dst->src == IPC_ANY || dst->src == CURRENT->tid)) {
                break;
            }

            if (flags & IPC_NOBLOCK) {
                return ERR_WOULD_BLOCK;
            }
```

```
            // The receiver task is not ready. Sleep until it resumes the
            // current task.
            task_set_state(CURRENT, TASK_SENDING);
            list_push_back(&dst->senders, &CURRENT->sender_next);
            task_switch();

            if (CURRENT->notifications & NOTIFY_ABORTED) {
                // The receiver task has exited. Abort the system call.
                CURRENT->notifications &= ~NOTIFY_ABORTED;
                return ERR_ABORTED;
            }
        }

        // Copy the message into the receiver's buffer and resume it.
        memcpy(&dst->buffer, m, sizeof(struct message));
        dst->buffer.src = (flags & IPC_KERNEL) ? KERNEL_TASK_TID : CURRENT->tid;
        task_set_state(dst, TASK_RUNNABLE);
    }
```

最初の while 文では，送信先タスクが受信可能状態になるまで待ち続けます。もし，送信先タスクが強制終了された場合は `NOTIFY_ABORTED` 通知を受信します。その場合は送り先が無くなってしまったので IPC を中断します。

送信可能である条件は 2 つあります:

- 送信先タスクが受信状態であること（`dst->state == TASK_RECEIVING`）
- 送信先タスクが送信元タスクを待っている（`dst->src == CURRENT->tid`），もしくはどこからのメッセージも受け付けるか（`dst->src == IPC_ANY`）

2 つ目の条件はそれぞれ，本章で説明したクローズド受信とオープン受信にあたります。

送信先タスクが受信状態になると，メッセージの送信を行います。といっても，送信先のタスク構造体にメッセージやその長さ，送信元タスク ID をコピーするだけです。コピーが終わると，送信元タスクを実行可能状態に移します。

ページフォルトメッセージなど，カーネルからメッセージを送る場合（`IPC_KERNEL`）には，送信元を `KERNEL_TASK_TID` に変更し，確かにカーネルから送信された（偽造されていない）メッセージであることをページャタスクが確認できるようにします。

次に受信処理です。

▼ kernel/syscall.c

```
    if (flags & IPC_RECV) {
        // Check if there're pending notifications.
        if (src == IPC_ANY && CURRENT->notifications) {
            // Receive the pending notifications.
            m->type = NOTIFICATIONS_MSG;
            m->src = KERNEL_TASK_TID;
```

```
            m->notifications.data = CURRENT->notifications;
            CURRENT->notifications = 0;
            return OK;
        }

        // Resume a sender task.
        LIST_FOR_EACH (sender, &CURRENT->senders, struct task, sender_next) {
            if (src == IPC_ANY || src == sender->tid) {
                task_set_state(sender, TASK_RUNNABLE);
                list_remove(&sender->sender_next);
                break;
            }
        }

        // Notify the listeners that this task is now waiting for a message.
        for (unsigned i = 0; i < TASKS_MAX; i++) {
            if (CURRENT->listened_by[i]) {
                notify(task_lookup(i + 1), NOTIFY_READY);
                CURRENT->listened_by[i] = false;
            }
        }

        // Sleep until a sender task resumes this task...
        CURRENT->src = src;
        task_set_state(CURRENT, TASK_RECEIVING);
        task_switch();

        // Received a message. Copy it into the receiver buffer and return.
        memcpy(m, &CURRENT->buffer, sizeof(struct message));
    }
```

受信処理は大まかに次の処理を行います。

1. 通知を受信していればそれをメッセージに変換してユーザに返す。
2. 送信状態にあるタスクがあれば 1 つだけ再開させる。
3. 待ちタスクテーブルのタスクたちに受信状態に入ったことを通知する。
4. 受付可能なタスク ID をセットして，受信状態でタスクをスリープ。
5. 送信側タスクによってタスクが再開され，`task_switch` から戻る。
6. メッセージバッファに受信メッセージをコピーする。

■ 通知 IPC

タスクへの通知は次の `notify` 関数で行います。

▼ kernel/ipc.c

```
void notify(struct task *dst, notifications_t notifications) {
    if (dst->state == TASK_RECEIVING && dst->src == IPC_ANY) {
        // Send a NOTIFICATIONS_MSG message immediately.
        dst->buffer.type = NOTIFICATIONS_MSG;
        dst->buffer.src = KERNEL_TASK_TID;
        dst->buffer.notifications.data = dst->notifications | notifications;
        dst->notifications = 0;
        task_set_state(dst, TASK_RUNNABLE);
    } else {
        // The task is not ready for receiving a event message: update the
        // pending notifications instead.
        dst->notifications |= notifications;
    }
}
```

本章で説明した通り，通知は UNIX シグナルのような非同期 IPC です。ブロックしてはいけません。`notify` 関数はもし宛先のタスクが受信状態に入っていれば，そのままメッセージの送信処理を行います。そうでない場合は，`task->notifications` に通知をセットして宛先タスクが `sys_ipc` で受信するまで通知を保留しておきます。

5.10 実装（Mach）

Mach の IPC は間接型メッセージパッシングです。IPC のエンドポイントは「ポート」と呼ばれます。各ポートはメッセージキューを持つことができ，非同期 IPC も可能です。ポートセット（*port set*）という機能を使うと，複数のポートからのメッセージを一度に待つことができます。

また，ポートには *port rights* と呼ばれるケイパビリティ機構があり，「一度だけメッセージを送信できる」といった制約を実現できます。

■ インタフェース

メッセージの送受信は次の `mach_msg` を使います。

▼ include/mach/message.h

```
extern mach_msg_return_t
mach_msg
   (mach_msg_header_t *msg,
    mach_msg_option_t option,
    mach_msg_size_t send_size,
    mach_msg_size_t rcv_size,
    mach_port_t rcv_name,
    mach_msg_timeout_t timeout,
    mach_port_t notify);
```

ひとつずつ引数の定義を見ていきましょう。まず，msg はメッセージのバッファです。option には行う処理（送信・受信など）を指定するビットフィールドです。send_size と rcv_size はそれぞれ送信・受信メッセージの大きさです。rcv_name には受信処理においてメッセージを受け付けるポートまたはポートセットを指定します。timeout はその名の通り IPC タイムアウト（ミリ秒），そして notify は筆者はコードを読んでもよく分かりませんでしたが何か通知を受け取る際に使うそうです。

■ メッセージの内容

メッセージヘッダ（mach_msg_header_t）には以下の情報が載っています。

- メッセージのサイズ
- 宛先ポート
- 送信先からの返信を受け取るポート
- シーケンス ID
- メッセージ種別

メッセージヘッダの後には，データの種類（mach_msg_type_t），その後ろにデータ本体が続きます。[*3]以下のようなデータを送ることができます。

- 整数，文字列など
- Port rights
- メモリ領域

メモリ領域はコピーオンライトで効率的にコピーされます。

5.11 実装（L4）

L4 は実装に依りますが，基本的に IPC は同期的です。また，L4 の初期実装では直接型 IPC ですが，seL4 といった最近の実装では間接型 IPC になっています。

■ インタフェース

ここでは，Fiasco.OC カーネルの IPC システムコールのプロトタイプ宣言を見てみましょう。

```
l4_msgtag_t l4_ipc(l4_cap_idx_t dest,
                   l4_utcb_t *utcb,
                   l4_umword_t flags,
                   l4_umword_t slabel,
                   l4_msgtag_t tag,
                   l4_umword_t *rlabel,
                   l4_timeout_t timeout);
```

[*3] いわゆる *Type-Length-Value*。

L4では，メッセージの送信・受信といったIPCの様々な操作を1つのシステムコールに凝縮しています。`flags`でどのような操作をするかを指定します。これを送信・受信の組み合わせによって，送信や受信，送受信（call）操作を1つのシステムコールで実現できるのです。

`tag`はメッセージの内容の情報とラベルを含んでいます。受信者はラベルによって，メッセージの種類を特定します。IPv4の「プロトコル番号」みたいなものです。

`timeout`ではその名の通りIPCのタイムアウトを指定します。

`slabel`と`rlabel`はIPC Gateというカーネルオブジェクトで使われるもののようです。カーネルが内容を保証してくれるので，セッション管理（ファイル記述子などの管理）のようなことをするためなのでしょう。

■ メッセージの内容

新しめのL4では，システムコールに渡すレジスタに加えてUTCBというメモリ領域を利用してIPCを行います。UTCB（User Thread Control Block）は各スレッドが持つメモリ領域です。UTCBはカーネル・ユーザランド両方からアクセス可能で，以下のようなフィールドがあります。

- メッセージレジスタ
- アクセプタ
- バッファレジスタ
- 例外処理を行うスレッド（例外ハンドラ）のID
- ページフォルト処理を行うスレッド（ページャ）のID

メッセージレジスタは，メッセージのデータを保持するフィールドです。実装によっては，高速化のためにいくつかのメッセージレジスタをCPUのレジスタに入れて転送します。メッセージレジスタにはuntypedとtypedの二種類があり，前者は単にコピーされるだけのデータで，後者はメモリページといったカーネルが管理するオブジェクトを転送するものです。

typed itemには，StringItem，MapItemとGrantItemの3つがあります。

まず，StringItemは間接型データを含んでいます。送信したいメモリアドレスと長さを指定します。メッセージレジスタへコピーする手間を無くすことがStringItemの目的です。

MapItemとGrantItemはそれぞれ，メモリページの共有（同じ物理メモリアドレスを示すページを受信者側に作る）と移譲（メモリページを受信側にマップし，送信側のアドレス空間から消す)。を行います。受信者は`fpage_unmap`システムコールで受信したページをページテーブルから削除できます。

バッファレジスタは，受信したStringItemらをどのメモリアドレスに置くかを指定するフィールドです。また，アクセプタ（Acceptor）はどのようなtyped itemを受理するかを指定するフィールドです。これらは受信者側が設定し，カーネルが利用します。

■ 割り込み処理

L4では割り込みを「割り込み通知用のスレッドIDからのメッセージ」として表現しています。そのため「割り込み番号1番の割り込みを待つ」という操作は，割り込み番号一番に対応するスレッドIDからのメッセージのクローズド受信によって実現できます。

5.12 実装（MINIX3）

MINIX の IPC の API は，メッセージを同期的に送信する SEND，メッセージを受信する RECEIVE，そしてメッセージを送信し返信を待つ SENDREC，通知 IPC の NOTIFY，そして非同期にメッセージを送信する SENDA などが提供されています。

▼ minix/include/minix/ipc.h

```
static inline int _ipc_send(endpoint_t dest, message *m_ptr)
static inline int _ipc_receive(endpoint_t src, message *m_ptr, int *st)
static inline int _ipc_sendrec(endpoint_t src_dest, message *m_ptr)
static inline int _ipc_notify(endpoint_t dest)
```

メッセージは最大 56 バイトのインラインデータのみです。ファイルハンドルやメモリページといったオブジェクトを送信する機能はありません。単にコピーされるだけです。

大きなデータはシステムコール（sys_vircopy）でコピーするか，Memory Grants[*4] という機構が用意されているようです。

5.13 実装（Zircon）

Zircon でも IPC の主役はやはりメッセージパッシングです。Zircon では「チャネル」を通して行う非同期・間接型 IPC です。カーネルがメッセージキューを内部で持っています。

チャネルを用いたメッセージパッシング以外にも，Zircon ではソケットや FIFO といった IPC が提供されています。

ソケットが別途用意されているというのは面白いですね。メッセージパッシングはメッセージ単位でやりとりするデータグラム型の通信なので，TCP ソケットのようなストリーム型のやりとりを実現するのはちょっと面倒[*5]なのです。

■ インタフェース

メッセージパッシングのインタフェースを見てみましょう。まずチャネルの作成です。

```
zx_status_t zx_channel_create(uint32_t options,
                              zx_handle_t* out0,
                              zx_handle_t* out1);
```

2 つのハンドル（out0，out1）が返されます。この 2 つは繋がっており，一方に送信するともう一

[*4] https://wiki.minix3.org/doku.php?id=developersguide:memorygrants
[*5] 受信側のユーザプログラムがデータのバッファリングをする必要があります。

方で受信できます。

次はメッセージの送信です。

```
zx_status_t zx_channel_write(zx_handle_t handle,
                             uint32_t options,
                             const void* bytes,
                             uint32_t num_bytes,
                             const zx_handle_t* handles,
                             uint32_t num_handles);
```

handle で指定される宛先に，インラインデータ（bytes）とハンドルたち（handles）を送るようです。単純明快で直感的なインタフェースですね。

最後にメッセージの受信です。

```
zx_status_t zx_channel_read(zx_handle_t handle,
                            uint32_t options,
                            void* bytes,
                            zx_handle_t* handles,
                            uint32_t num_bytes,
                            uint32_t num_handles,
                            uint32_t* actual_bytes,
                            uint32_t* actual_handles);
```

num_bytes と num_handles には受信バッファの大きさを指定し，actual_bytes と actual_handles に実際の受信したメッセージの大きさが返ってきます。

受信ハンドルを 1 つだけ指定しており，いわばクローズド受信だけ提供されています。複数のハンドルを一度に待つ（オープン受信）には，zx_object_wait_many システムコールにハンドラの配列を与えるか，代わりに「ポート（Port）」というイベント通知を受け取れるオブジェクトを使うようです。

6 ユーザランド

マイクロカーネルベースの OS では，カーネルだけでは何の役にも立ちません。主役はユーザランドのプログラムたちです。

本章では，マイクロカーネルのユーザランドに関する雑多なトピックを解説します。

6.1 ブート処理

マイクロカーネルが起動した後，何をすればよいのでしょうか。Linux では，カーネルが `init` プロセスをルートファイルシステムから探して起動し，雑多な初期化処理を行いログイン画面にたどり着きます。

その一方で，マイクロカーネルはファイルシステムという概念も知りません。そのため，最初のユーザプログラムの実行ファイルをカーネルイメージに埋め込んだり，ブートローダに読み込ませておいたりします。

Resea では，最初のユーザプロセス（`init`）を objcopy[*1]を使って生バイナリ形式に変換したファイル（`initfs.bin`）をカーネルに埋め込み，それをそのままメモリ上に展開して実行しています。生バイナリにすることで，ELF といった実行ファイルのパーサをカーネルが持たずにすみます。

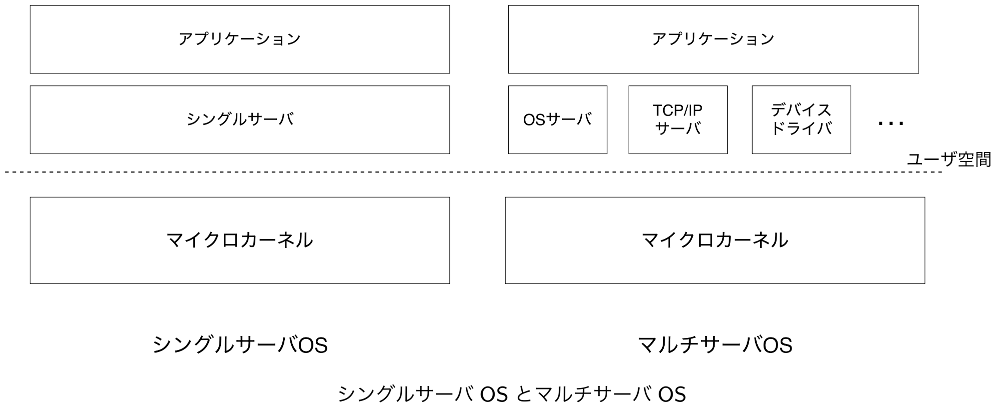
起動された最初のユーザプロセスは，自分の実行ファイルに埋め込まれた他のサーバ（デバイスドライバ等）を読むなどして，ユーザランドを構築します。また，ブート処理に加えて，カーネルによっては最初のユーザプロセスに物理メモリ空間全体を与えています。カーネルが使わないメモリをユーザ空間で柔軟に管理できるようにするためです。

6.2 シングルサーバ OS vs. マルチサーバ OS

マイクロカーネルベースの OS では「複数のユーザプロセスが通信しあって OS を構成する」というマルチサーバ OS と「単一サーバが OS の機能を提供する」というシングルサーバ OS の 2 つの構成が可能です。

シングルサーバ OS は，マイクロカーネルの「カーネルの仕事を複数のユーザプロセスに隔離して安全性・信頼性の向上を狙う」というメリットを潰しています。しかし，L4Linux のように他のモノリシックカーネルをマイクロカーネル上に移植するには良いアプローチです。

*1 「実行ファイルのための十徳ナイフ」のようなツール。ビルドの仕上げによく使われています。



シングルサーバ OS とマルチサーバ OS

6.3 POSIX 互換

1 からユーザランドのプログラムを書いていくのが面倒なのと，わざわざ新しい OS の API を学ぶのは敷居を上げてしまうという問題を解決するために，POSIX 互換を実装する OS が多々あります。POSIX 互換を実現できれば，そこら辺の UNIX プログラムが（多少パッチを当てる必要はあるでしょうが）そのまま動いてくれるはずです。

マイクロカーネルベースの OS では，POSIX の部分はユーザランドで動くサーバとして実装されます。

POSIX 互換を実現する手法として，API レベルまたは ABI レベルの互換性を提供する 2 つの手法があります。

API レベルの互換性は，単に `unistd.h` といったインタフェースやライブラリを用意するという方法です。オーソドックスなやり方ですね。マイクロカーネルに手を加えなくとも実現できるというメリットがありますが，様々な UNIX プログラムのビルド環境を用意する苦行を強いられます。

対する ABI レベルの互換性は，実行バイナリ（例えば Linux の ELF ファイル）をそのまま実行できるようにするという方法です。バイナリ互換であるとも言います。プログラムのリビルドをしなくて済みますが，実現できるかはマイクロカーネルの（特にシステムコール周りの）実装次第です。

6.4 IPC スタブ

ここまでメッセージパッシングの設計について，マイクロカーネルが担当する部分を見てみました。ユーザランドでは，利便性のために IPC システムコールのラッパー関数（IPC スタブ）を導入することがあります。

メッセージの送受信の前後には，メッセージデータの作成（シリアライズ処理）や，受信したメッセージのパース（デシリアライズ処理）が必要となります。JavaScript でいうところの，`JSON.stringify` と `JSON.parse` メソッドがやっているような処理です。

基本的にカーネルはバイト列を単にコピーするだけなので，ユーザランドプログラミングではそのデータ構造をきちんと意識する必要があります。ただ，メッセージごとにシリアライズ・デシリアライズ処理を書いていくのは面倒なので，専用言語でメッセージの内容を記述し，ソースコード（IPC スタブと呼ばれます）を自動生成するという手法がよく使われます。

■ IDL

メッセージパッシングのプロトコルを定義するために，インタフェース記述言語（IDL: Interface Definition Language）というドメイン記述言語が使われます。

IDL ごとに文法は違えど書くことは大体同じで，以下の情報を載せていることが多いです。

- 型の定義。C 言語でいう `struct` や `typedef`。
- メソッド名。C 言語でいう関数名。
- 属性。メソッドの ID 番号など。
- リクエストメッセージのフィールド（関数でいう引数）の定義
- レスポンスメッセージのフィールド（関数でいう戻り値）の定義

メッセージの各フィールドには，大体以下のような情報を定義しています。

- フィールド名
- データ型（整数，文字列，可変長配列，メモリページなど）
- データの「方向」。サーバへ送るデータか，サーバから受け取るデータか，それともその両方か。
- 可変長配列の場合，その長さを示すフィールド名。
- 可変長配列の場合，受け取るためのメモリバッファをどう用意するか（IPC スタブ内部で動的に malloc するか否か）

では，実際の IDL のサンプルを見て雰囲気を感じてみましょう。

■ MIG

Mach Interface Generator（MIG）は Mach カーネルの IPC スタブジェネレータです。専用のインタフェース記述言語から，サーバサイドとクライアントサイドのソースコードを生成します。

```
/* でのインタフェース記述の例（MIGGNU Hurd: hurd/fs.より抜粋）defs */
subsystem fs 20000;
#include <hurd/hurd_types.defs>

routine dir_readdir (
    dir: file_t;
    RPT
    out data: data_t, dealloc[];
    entry: int;
    nentries: int;
    bufsiz: vm_size_t;
    out amount: int);
```

どことなく C 言語のプロトタイプ宣言っぽいですね。IPC スタブの出力例は，載せるには長いので参考文献を参照してください。

■ IDL4

一部の L4 カーネルで使われる IPC スタブジェネレータです。インタフェース記述言語は 2 つの文法に対応しています。特に IDL4 独自言語があるという訳ではないようです。

まずは，CORBA IDL による例です。見た感じ MIG とは違う印象を受けますが，載っている情報は基本的に変わりません。

```
/* CORBA による記述例（IDLIDL4 Version 1.0.0 ' Users Manual より引用） */
module storage {
  interface textfile {
    void readln(
      inout short pos,
      out string line
    );
    void writeln(
      inout short pos,
      in string line
    );
    int get_pos();
  };
};
```

次に，DCE IDL による例です。大体 CORBA IDL と大体同じですね。

```
/* DCE による記述例（IDLIDL4 Version 1.0.0 ' Users Manual より引用） */
library storage {
  interface textfile {
    void readln(
      [in, out] short *pos,
      [out, string] char **line
    );
    void writeln(
      [in, out] short *pos,
      [in, string] char **line
    );
    int get_pos();
  };
};
```

■ FIDL

Fuchsia Interface Definition Language（FIDL）は Fuchsia で使われている高機能なインタフェース記述言語です。

ここでは，イーサネットデバイスのインタフェース定義を見てみましょう。

▼ zircon/system/fidl/fuchsia-hardware-ethernet/ethernet.fidl

```
library fuchsia.hardware.ethernet;

using zx;

struct MacAddress {
    array<uint8>:6 octets;
};

// Info.features bits
const uint32 INFO_FEATURE_WLAN = 0x00000001;
const uint32 INFO_FEATURE_SYNTH = 0x00000002;
const uint32 INFO_FEATURE_LOOPBACK = 0x00000004;

struct Info {
    uint32 features;
    uint32 mtu;
    MacAddress mac;
};

struct Fifos {
    // handles for the rx and tx fifo
    handle<fifo> rx;
    handle<fifo> tx;

    // maximum number of items in rx and tx fifo
    uint32 rx_depth;
    uint32 tx_depth;
};

/* ... */

[Layout = "Simple"]
protocol Device {
    // Obtain information about device
    GetInfo() -> (Info info);

    // Obtain a pair of fifos for queueing tx and rx operations
    GetFifos() -> (zx.status status, Fifos? info);

    // Set the IO Buffer that will provide the data buffers for tx and rx operations
    SetIOBuffer(handle<vmo> h) -> (zx.status status);

    /* ... */
```

```
};
```

型定義（`struct MacAddress` など）が続いた後に，メッセージの内容（`protocol Device`）が記述されています。見た目は違えど，MIG や IDL4 と大雑把には同じですね。

FIDL では，可変長配列（`vector`）や文字列といった少し複雑なデータを簡単に埋め込むことができます。興味のある方はドキュメント[*2]が充実しているのでそちらをご覧ください。

6.5 ユーザランドプログラミングの例（Resea）

Resea では，どのユーザプログラムもシングルスレッドのイベント駆動型プログラミングで実装されています。メッセージを受信し，処理を行った後に返信し，次のメッセージを待つ... という無限ループになっています。イベント駆動型プログラミングは，開発が楽でデバッグしやすい優れた手法[*3]です。

では，Resea におけるユーザランドプログラミングを見てみましょう。ここでは，KVS（Key-Value ストア）サーバを取り上げます。memcached の簡易版みたいなやつです。Resea ではファイルシステムの代わりとして KVS を使っています。

KVS サーバは以下の機能を提供しています。

- `GET`: キー文字列に対応するデータ（任意のバイナリデータ）の取得
- `SET`: キー文字列に対応するデータの追加・更新
- `DELETE`: キーの削除
- `LISTEN`: 指定したキーのデータが更新されたらメッセージを送ってもらう

▼ `servers/kvs/kvs.h`

```
struct entry {
    list_elem_t next;
    char key[KVS_KEY_LEN];
    size_t len;
    list_t listeners;
    uint8_t data[KVS_DATA_LEN_MAX];
};

struct listener {
    list_elem_t next;
    tid_t task;
};
```

[*2] https://fuchsia.dev/fuchsia-src/development/languages/fidl

[*3] 詳しくは *J.K. Ousterhout. Why Threads Are A Bad Idea (for most purposes)* で解説されています。

▼ servers/kvs/main.c

```
/* .. */

static void deferred_work(void) {
    async_flush();
}

void main(void) {
    /* .. */

    while (true) {
        struct message m;
        error_t err = ipc_recv(IPC_ANY, &m);
        ASSERT_OK(err);

        switch (m.type) {
            case NOTIFICATIONS_MSG:
                break;
            case KVS_GET_MSG: {
                struct entry *e = get(m.kvs.get.key);
                if (!e) {
                    ipc_reply_err(m.src, ERR_NOT_FOUND);
                    break;
                }

                TRACE("GET '%s' (len=%d)", e->key, e->len);
                ASSERT(e->len <= KVS_DATA_LEN_MAX);
                m.type = KVS_GET_REPLY_MSG;
                m.kvs.get_reply.len = e->len;
                memcpy(m.kvs.get_reply.data, e->data, e->len);
                memset(&m.kvs.get_reply.data[e->len], 0,
                       KVS_DATA_LEN_MAX - e->len);
                ipc_reply(m.src, &m);
                break;
            }
            case KVS_SET_MSG: {
                if (m.kvs.set.len > KVS_DATA_LEN_MAX) {
                    ipc_reply_err(m.src, ERR_TOO_LARGE);
                    break;
                }

                struct entry *e = get(m.kvs.set.key);
                if (!e) {
                    e = create(m.kvs.set.key);
                }

                TRACE("SET '%s' (len=%d)", e->key, m.kvs.set.len);
                update(e, m.kvs.set.data, m.kvs.set.len);
                ipc_reply_err(m.src, OK);
                changed(e);
                break;
            }
```

```
            /* .. */
            case KVS_LISTEN_MSG: {
                struct entry *e = get(m.kvs.listen.key);
                if (!e) {
                    ipc_reply_err(m.src, ERR_NOT_FOUND);
                    break;
                }

                TRACE("LISTEN '%s' (task=%d)", e->key, m.src);
                listen(e, m.src);
                ipc_reply_err(m.src, OK);
                break;
            }
            /* .. */
            default:
                TRACE("unknown message %d", m.type);
        }

        deferred_work();
    }
}
```

ipc_reply はノンブロッキング IPC のラッパー関数です。クライアントは送受信（call）IPC を使うはず（つまり既に受信状態にある）なので返信は常にノンブロッキングに成功するはずです。普通のブロッキング IPC は，クライアントが受信状態にない場合にサーバの処理が止まってしまうので使いません。

ここでは，KVS サーバが具体的にどういうことをやっているのかより，main 関数がどのような構造になっているのかを掴んでほしいのです。この関数は以下のような構造になっています。

```
/* メインループ */
while (true) {
    /* メッセージの受信 */
    struct message m;
    tid_t src = ipc_recv(IPC_ANY, &m);

    /* メッセージ種別に依る処理 */
    switch (m.type) {
        case /* メッセージの種類 */:
            // 1. パラメータのバリデーション（長すぎないかなど）
            // 2. 要求内容の処理（データの取得など）
            // 3. 返信メッセージの作成・送信
            break;
    }

    /* 定期ジョブの実行 */
    deferred_work();
```

```
}
```

メッセージを受け取って処理を行い返信する，という流れを永遠と続けるイベント駆動型プログラミングになっていることがわかると思います。

残るは deferred_work 関数です。

```
static void deferred_work(void) {
    async_flush();
}
```

サーバはシングルスレッドのイベント駆動型プログラミングです。ipc_send（同期的 IPC）では，クライアントが受信状態にない場合にサーバの処理が止まってしまいます。

そこで Resea では「送れそうな時にノンブロッキング送信する」という戦略をとっています。サーバがクライアントにメッセージを送る際には，次の async_send() 関数という非同期 IPC ライブラリを代わりに使います。

▼ libs/std/async.c

```
void async_send(tid_t dst, struct message *m) {
    error_t err = ipc_send_noblock(dst, m);
    // TODO: Should we handle other errors?
    switch (err) {
        case OK:
            return;
        case ERR_WOULD_BLOCK: {
            // The receiver is not ready. We need to enqueue it and try later in
            // `async_flush()`.
            struct message *buf = malloc(sizeof(*buf));
            memcpy(buf, m, sizeof(*buf));
            struct async_message *am = malloc(sizeof(*am));
            am->dst = dst;
            am->m = buf;
            list_push_back(&pending, &am->next);
            ipc_listen(dst);
            break;
        }
    }
}
```

もしノンブロッキング送信に失敗した場合（ERR_WOULD_BLOCK）は，メッセージを非同期 IPC ライブラリのキューに入れておき，送信先タスクが受信状態になったら通知 IPC を送ってもらうようカー

ネルに設定します。送信先が受信状態になるとメインループで NOTIFICATIONS_MSG メッセージを受け取るので，async_flush() 関数でキューに入っているメッセージのノンブロッキング送信を試みるという処理を成功するまで続けます。

▼ libs/std/async.c

```
void async_flush(void) {
    LIST_FOR_EACH (am, &pending, struct async_message, next) {
        error_t err = ipc_send_noblock(am->dst, am->m);
        // TODO: Should we handle other errors?
        switch (err) {
            case OK:
                // Hooray! Successfully sent the message. Remove it from the
                // list and free memory.
                list_remove(&am->next);
                free(am->m);
                free(am);
                break;
            case ERR_WOULD_BLOCK:
                // The receiver is still not ready. Try again next time.
                break;
        }
    }
}
```

タスクが増えるほど遅くなるのであまり良い手法ではありませんが，タスク数が少ない場合は一応上手く動いてくれます。 [*4]

6.6 ユーザランドプログラミングの例（MINIX3）

ここでは，J. N. Herder らの *Modular system programming in MINIX 3* で紹介されている，MINIX での簡単なサーバ実装を見てみましょう。かなり簡略化されていますが，雰囲気はつかめると思います。

```
void semaphore_server( ) {
  message m;
  int result;
  /* Initialize the semaphore server. */
  initialize( );
  /* Main loop of server. Get work and process it. */
  while(TRUE) {
    /* Block and wait until a request message arrives. */
```

[*4] 筆者的には，より洗練された手法がないかと頭を悩ませている部分です。

```
    ipc_receive(&m);
    /* Caller is now blocked. Dispatch based on message type. */
    switch(m.m_type) {
      case UP: result = do_up(&m); break;
      case DOWN: result = do_down(&m); break;
      default: result = EINVAL;
    }
    /* Send the reply, unless the caller must be blocked. */
    if (result != EDONTREPLY) {
      m.m_type = result;
      ipc_reply(m.m_source, &m);
    }
  }
}
```

このサーバは名前の通りセマフォを実現するサーバです。各プロセスは UP または DOWN メッセージをこのサーバに送信し，セマフォを操作します。

メッセージを受信し処理した後に返信するという，Resea と同じ流れになっていることが分かると思います。返信用メッセージのバッファを用意せず，m を返信メッセージの作成にも再利用していたり，EDONTREPLY エラーを導入することで返信処理を一箇所にまとめていたりと小技が入っています。

7 高度なトピック

この章では，マイクロカーネルの面白い研究トピックをいくつかご紹介します。

7.1 非同期 IPC は本当に必要ないのか

L4 や MINIX では，同期的 IPC を採用しています。しかし本当に同期的 IPC と通知のみで十分なのでしょうか。筆者は，同期的 IPC と通知だけでは十分ではないと考えています。

例として，Resea で TCP/IP サーバを作る状況を考えてみましょう。TCP/IP サーバは，ネットワークデバイスドライバにパケットの送信をメッセージパッシングで依頼します。いちいちシステムコールを叩いていたら面倒なので，以下のようなラッパー関数（`ethernet_transmit`）を導入すれば便利そうです。

```
void ethernet_transmit(tid_t driver, const void *packet, size_t packet_len) {
    struct message m;
    m.type = M_NET_TX;
    m.len = packet_len;
    memcpy(m.net_tx.payload, packet, packet_len);
    ipc_send(driver, m);
}
```

この関数には 1 つ厄介な問題点があります。それは「ネットワークデバイスドライバが受信状態にない場合にブロックしてしまう」という問題です。Resea はシングルスレッドのイベント駆動型プログラミングを採用しています。そのため，処理（`ipc_send`）がブロックすると他の処理も止まってしまうのです。悪意のあるデバイスドライバが故意に受信状態に入らずブロックさせてくるかもしれません。TCP/IP サーバはデバイスドライバを信頼したくないのです。これは Resea だけでなく L4 や MINIX でも言えることです。

非同期 IPC では，送信先が受信状態にない場合はメッセージキューに貯まるだけでブロックしないので問題ありませんが，同期的 IPC では一工夫必要です。

では，どう解決すればよいのでしょうか。筆者の知る限り 2 つの手法があります。[*1]

[*1] L4 では，この問題に対処しているユーザランドの実装例が見つかりませんでした。L4 は実世界で使われてはいますが，オープンソースにしにくいシステムが多いのかもしれません。

- 送信先が受信状態に入った時に通知を送ってもらい，ノンブロッキング送信[*2]で再送を試みる（Resea）
- ユーザランド側で非同期メッセージのテーブルを持ち，カーネルに送信できるときに適宜送信してもらう（MINIX）

どちらも，カーネルではなくユーザランド側で非同期メッセージのキューやテーブルを管理しています。Resea がとっている手法はユーザランドプログラミングのところ（async_send() と async_flush() 関数）で説明したので割愛します。MINIX では，SENDA という非同期 IPC システムコールが入っています。SENDA は次のようなインタフェースになっています。

▼ minix/include/minix/ipc.h

```
/* Datastructure for asynchronous sends */
typedef struct asynmsg
{
    unsigned flags;
    endpoint_t dst;
    int result;
    message msg;
} asynmsg_t;

static inline int _ipc_senda(asynmsg_t *table, size_t count)
```

大まかには以下の流れで非同期 IPC を実現しています。

1. ユーザランドのライブラリ（asynsend3 関数）が，自身が管理する asyncmsg_t テーブルの各フラグをチェックして，使われていないスロットにメッセージをセット。
2. SENDA システムコールを実行。カーネルは count 個の asyncmsg_t の配列を走査し，メッセージの送信を試みる。無事送信できたら，フラグが更新を更新する。
3. もし送信できなかったメッセージがあれば，送信先に「非同期に送信されるメッセージがある」ことを示すビットをセットし，asyncmsg_t テーブルへのユーザポインタをカーネルの方で持っておく。
4. 送信先の受信処理の際に asyncmsg_t テーブルを走査し非同期メッセージを受信する。そして，送信元の（ユーザの）テーブルのフラグを更新する。

ユーザランド側で asyncmsg_t のテーブルを管理することで，カーネルでメッセージキューを持たずに非同期 IPC を実現しています。テーブルのスロット数は 2*_NR_PROCS となっています。ただし，_NR_PROCS はプロセス数の最大値（デフォルトで 256）です。MINIX ではプロセス数が少ないのでこれで十分なのでしょう。

もしよりよい解決法が思いついたら，ぜひ論文を書いて筆者に教えて下さいね！

[*2] 「ノンブロッキング」と「非同期」は異なる概念です。混同しないよう注意してください。

7.2 ユーザレベルメモリ管理

L4 といった一部のマイクロカーネルでは，カーネルでなくユーザプロセスによって物理メモリを管理しています。どうメモリを使うか（割り当てアルゴリズム）をカーネルに含めず，ユーザランドで柔軟に変えられるようにしようという目的です。

カーネルは，物理メモリを割り当てる代わりに「ページテーブルへのマッピング操作」をユーザランドに提供します。ユーザプロセス（特に最初に起動される `init` プロセスのようなもの）は，自分自身で物理メモリの使用状況を管理し，要求に応じて割り当てるのです。

このように，本来カーネルが担ってきた機能をユーザランドで実装したものをよく「ユーザレベル XXX」と呼びます。

7.3 内部で動的メモリ割当をしないカーネル

ユーザレベルメモリ管理を実現しても，どうしても動的にメモリをカーネル内で確保しなければならないものがあります。例えば，プロセスを作成したならそれを管理する構造体やページテーブルの置き場が必要です。また，ページをマップするだけでも，新たにページテーブルの下層を新たに割り当てる必要があることがあります。

そのため，マイクロカーネルには固定長のヒープを静的に確保しておくものが良く見られます。しかし，カーネルのヒープが満杯になったときの対処を考えなければなりません。

seL4 カーネルでは，ユーザランドのプロセスに「どのメモリページをどのように使うか」を指定させることで，カーネル内部で動的メモリ割当しなくて済むようにしています。

具体的には，各メモリページは最初に untyped オブジェクトとして存在します。ユーザプロセスは，それをスレッド管理構造体やページテーブルといったオブジェクトに「型変換」を行い，カーネルにはその型変換後のオブジェクトをカーネルに指定します。

7.4 ユーザレベルスケジューラ

次にどのスレッドを実行するかを決めるスケジューラには， First Come First Serve（FCFS）や，Shortest Remaining Time First（SRTF），固定優先度スケジューリングなど，様々なアルゴルリズム（方針）があります。ユーザレベルメモリ管理と同じように，どのアルゴリズムを使うかをカーネル内にハードコードするのは柔軟性のことを考えると避けたいことです。

そこで，ユーザランド側で柔軟にスケジューリングアルゴリズムを実装できるように，基本的な操作のみ（機構）を提供しているマイクロカーネルがあります。

例として，MINIX を見てみましょう。MINIX カーネルは以下の通りプロセスを実行します。

- 複数の優先度レベルのあるラウンドロビンスケジューリング。最も優先度の高いプロセスが常に選ばれる。
- 実行時間（`quantum`）を使い切ると，2 通りに処理が分かれる。
  - そのプロセスにスケジューラが設定されているなら，処理を停止し，スケジューラにメッセージを送信する。そして，スケジューラが再び quantum を補充するのを待つ。
  - スケジューラが設定されていないなら，quantum をカーネルが補充する。

プロセスのスケジューリング方針を変える sys_schedule カーネルコールのインタフェースは次の通りです。

▼ minix/minix/lib/libsys/sys_schedule.c

```
sys_schedule(endpoint_t proc_ep, int priority, int quantum, int cpu, int niced)
```

対象のプロセス（proc_ep），優先度（priority），実行時間（quantum），実行する CPU（cpu），優先度の低いプロセス[*3]としてカウントするか（niced）を指定します。

7.5 信頼性の向上

マイクロカーネルの近年のトレンドは「マイクロカーネルによるシステムの信頼性の向上」だと筆者は感じています。性能面ではモノリシックカーネルには到底敵いませんが，「落ちたらまずいシステム」の基盤としてはマイクロカーネルに分があります。モノリシックカーネルに比べカーネルがやることが限られているのでバグが少ないはずです。また，カーネルパニックが起きるとカーネルはどうしようもないですが，ユーザプロセスが落ちたくらいならどうにかなるかもしれません。

■ 信頼性第一の MINIX3

MINIX3 は信頼性に重きを置いたマイクロカーネルベースの OS で，いろいろと面白い研究が行われています。中でも面白いのはクラッシュリカバリーに関する研究でしょう。

クラッシュリカバリーはデバイスドライバといった機能がクラッシュした際に，どうやってシステムの機能を維持するかという問題です。

リカバリー手段として，MINIX3 には Reincarnation[*4]（RS）サーバがあります。　たまに起きるバグや，メモリリークのような時間経過で現れるバグによってサーバ（特にデバイスドライバ）が落ちた際に，RS サーバは決められたポリシーに従ってサーバプログラムの再起動等を行います。

また，別の試みとしてグローバル変数などの「ステート」の複製をとっておくように LLVM を使ってコンパイル時にコードを注入し，サーバがクラッシュしたらステートを復元して再起動する，というびっくりクラッシュリカバリー機能の研究[*5]もあります。

ライブアップデートなど他の研究については参考文献から探してみてください。

[*3] 実行時間の統計をとるときに使われます。何が嬉しいのかは，こちらで分かりやすく説明されています。https://askubuntu.com/a/399384

[*4] *reincarnation* は「生まれ変わり」といった意味をもちます。

[*5] *Cristiano Giuffrida, et al. We Crashed, Now What? HotDep'10.*

■ 形式検証済みが売りの seL4

seL4 はカーネルが「きちんと動く」ことの形式的証明がつけられた L4 派生カーネルです。[*6]筆者は形式検証に明るくないので適当なことは言えませんが，公式ドキュメント[*7] 等によるとデッドロックや NULL ポインタ参照といったバグがないそうです。

7.6 ハードウェア支援による IPC 高速化

近年，ハードウェア支援を活用して意地でも IPC を高速化しようという試みがいくつかあります。ここでは，2 つは紹介します。

まずは Zeyu Mi らの SkyBridge (EuroSys'19) です。VMFUNC 命令を活用して Zircon と L4 ファミリー（seL4 と Fiasco.OC）の IPC を高速化したという論文です。

もう 1 つは Dong Du らの XPC（ISCA'19）です。こちらはハードウェアに新たな独自命令を加えることで，IPC を高速化するという論文です。

詳しい仕組みははここでは解説しませんので，ぜひ論文を読んでみてください。マイクロカーネル研究は信頼性にフォーカスが移っている雰囲気を感じますが，技術の発展に伴って IPC の高速化といったトピックが再び掘り起こされるというのは面白いですね。

[*6] 派生とはいえカーネルのインターフェイスは抜本的に変わっています。
[*7] https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html

8 おわりに

本書では，マイクロカーネルの概念や設計・実装をざっくり見てみました。マイクロカーネルと関わりをもつことになった時にいくらか役立ってくれることでしょう。

8.1 Resea の開発に参加しよう

この本を手にとった読者のみなさんはマイクロカーネルに興味のある大変希少な人材なので宣伝させてください。本書で紹介した筆者が暇つぶしに一人でコツコツと作っているマイクロカーネルベースの OS「Resea」の開発にぜひ参加してください。

Linux や FreeBSD といった成熟したカーネルへの開発は敷居が高く感じるかもしれません。しかし，Resea は暇人が一人で開発している機能もコード量も限られた新米 OS です。TCP/IP サーバといったユーザランドをひっくるめても 1 万行もない非常に小さなソフトウェアなので，全体像をしっかり理解できます。ぜひ Resea でカーネルハッカーデビューしてください。

次のようなことをすると面白いかもしれません。

- **x86_64 以外の CPU への移植:** コンテキストスイッチなどを頑張って実装しましょう。アーキテクチャ依存部分は分離してあります。
- **IPC fastpath の実装:** fastpath の実装といったカーネルの高速化は面白いのでお薦めです。
- **xv6 on Resea:** UNIX ライクな教育用 OS である xv6 を Resea 上で動かしてみましょう。「シングルサーバ OS」として実装することになると思います。
- **Rust で再実装:** Resea は C 言語で開発されていますが，世の中には C 言語以外にも様々なプログラム言語があります。Resea カーネルを Rust や C++ で再実装してみると面白いでしょう。
- **ユーザランドを JavaScript で書く:** 今のところ，ユーザランドは C 言語で書かれています。JavaScript といった他のプログラム言語でもアプリケーションを書けるようにするのも面白いでしょう。簡単なのは組み込み向けのインタプリタ（Lua や JerryScript[*1]など）の移植です。

もし気が向いたらぜひ Pull Request や要望を GitHub（https://github.com/nuta/resea）に送ってくださいね。

[*1] https://github.com/jerryscript-project/jerryscript

8.2 次に何をするか

最後に，本書の次にマイクロカーネルを学ぶには何をするとよいのか，いくつかおすすめをして終わりたいと思います。

1 つ目は，参考文献に上がっているようなマイクロカーネルの論文やドキュメントを読むことです。全て英語で書かれていますが，Google 翻訳で言葉の壁は超えられるはずです。論文を読み終えたら，引用されている他の論文を辿っていきましょう。関連研究のセクションに先行研究が短くまとめられています。研究 OS だけでなく商用利用されているマイクロカーネルも（クローズドソースが多いですが）参考になります。例えば，QNX の IPC のドキュメント[*2]はなかなか読み応えがあります。

2 つ目は，マイクロカーネルの実装を読むことです。論文にはカーネルの大雑把な内容しか書かれていません。同じ機能でも実装は大きく異なることが多く，細かい設計・実装テクニックを学ぶことができます。

最後に，マイクロカーネルを自分で作ってみることです。自分で作ろうとすると，どうしても非常に細かい実装部分まで理解する必要があり，マイクロカーネルの作者たちの気持ちが分かるようになります。

車輪の再発明は車輪を学ぶ上で最適な勉強法です。ぜひ「なぜ車輪は丸いのか」を考える日々を過ごしてみてください。

[*2] https://www.qnx.com/developers/docs/6.5.0SP1.update/com.qnx.doc.neutrino_sys_arch/ipc.html

A 参考文献

マイクロカーネル vs. モノリシックカーネル

- The Tanenbaum-Torvalds Debate
  - MINIX 作者の Andrew S. Tanenbaum と Linux 作者の Linus Torvalds の間で起こった「マイクロカーネルとモノリシックカーネルのどちらがどのように優れているか」の有名な議論です。ちなみに，タネンバウム先生は 20 年以上経ってもこの議論に対するメールを貰っているそうです。
- Biggs, et al. The Jury Is In: Monolithic OS Design Is Flawed. APSys'18.
  - 過去にあった Linux カーネル（モノリシックカーネル）の脆弱性は，マイクロカーネルではどのように安全性への影響を与えるかを検証した論文です。

マイクロカーネルの性能

- Sean Peters, et al. For a Microkernel, a Big Lock Is Fine. APSys'15.
  - 「(性能が悪いとされる）Big Kernel Lock 方式でも，マイクロカーネルなら十分性能がよい」ことを主張する論文です。
- Blackham, et al. Correct, Fast, Maintainable – Choose Any Three! APSys'12.
  - IPC fastpath は，性能のためにアセンブリで書かれるものでした。しかし，Blackham らは「コンパイラの最適化の気持ちを考えて C 言語で書いたら，十分性能が良く（C 言語で書いているので）メンテナンスしやすい IPC fastpath が実装できる」ことを主張しています。
  - IPC fastpath 自体について学ぶためにも有用です。

MINIX

- アンドリュー・タネンバウム，アルバート・ウッドハル．『オペレーティングシステム』 第 3 版
  - ペレーティングシステムの有名な本です。OS の概念と MINIX での実装が解説されています。MINIX のソースコードが付録としてついており，電話帳のように分厚い本です。通称 MINIX 本です。
- Jorrit N. Herder, et al. The Architecture of a Reliable Operating System.
  - MINIX3 の概要が紹介されています。
- https://wiki.minix3.org/doku.php?id=publications
  - MINIX3 関連の論文が紹介されています。
- Andrew S. Tanenbaum. Lessons Learned from 30 Years of MINIX.
  - タネンバウム先生が MINIX の開発を通して得た学びを紹介しています。

Mach

- Keith Loepere. Mach 3 Server Writer's Guide. 1992.
  - Mach のサーバプログラミングのマニュアルです。
- Richard P. Draves, et al. MIG - The MACH Interface Generator. 1989.
  - Mach のインタフェース記述言語 MIG の解説書です。

L4

- Jochen Liedtke. Improving IPC by kernel design. SIGOPS'93.
  - 性能を重視したマイクロカーネルを設計をしたら高速な IPC を実現できたという論文です。
  - 謝辞に *This paper was written using LATEX on top of L3* と書いていてカッコいいのですね。自作 OS の論文をその自作 OS で動く LaTeX 処理系で書くのはなかなか憧れます。
- Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? SOSP'13.
  - L4 カーネルが進化していく中で，どのような設計が改善されてきたのかをまとめたものです。
- Alan Au and Gernot Heiser. L4 User Manual ver 1.15. 1998.
  - その名の通り L4 カーネルの解説書です。L4 の仕組みから簡単なユーザランドプログラミングについて解説しています。
- L4Ka Team. L4 eXperimental Kernel Reference Manual Version X.2 Revision 7-6288f0536ce1. 2011.
  - L4 カーネルの仕様の 1 つです。システムコールやメッセージの具体的な仕様が載っています。
- Andreas Haeberlen. IDL4 Version 1.0.0 User's Manual. 2003.
  - L4 の IPC スタブジェネレータのマニュアルです。
- Tarjei Mandt, et al. Demystifying the Secure Enclave Processor. Blackhat USA'16.
  - iPhone の Secure Enclave で動く L4 カーネルベースの OS を解説しています。
  - つまり，iPhone を買うと L4 がついてくる！

Fuchsia（Zircon）

- https://fuchsia.dev
  - Fuchsia のドキュメントが載っています。
- https://fuchsia.googlesource.com/fuchsia/
  - Fuchsia のリポジトリです。Zircon カーネルも入っています。

ハードウェア支援による IPC 高速化

- Zeyu Mi, et al. Skybridge: Fast and secure inter-process communication for microkernels. EuroSys'19.
  - `VMFUNC` という仮想化技術に使われる命令を活用してマイクロカーネル（Zircon と L4 ファミリー）の IPC を高速化したという論文です。
- Dong Du, et al. XPC: architectural support for secure and efficient cross process call. ISCA'19.
  - IPC 専用命令を新たに加えることで，FPGA 上の seL4 と Zircon，そして Android Binder の IPC を高速化するという論文です。

B Resea カーネルのソースコード

ファイル

0656 kernel/ipc.c
0641 kernel/ipc.h
1091 kernel/kdebug.c
1076 kernel/kdebug.h
0017 kernel/main.c
0001 kernel/main.h
0100 kernel/memory.c
0064 kernel/memory.h
1025 kernel/printk.c
0998 kernel/printk.h
0790 kernel/syscall.c
0767 kernel/syscall.h
0353 kernel/task.c
0224 kernel/task.h

関数

0349 arch_disable_irq
0348 arch_enable_irq
0012 arch_idle
0784 arch_memcpy_from_user
0785 arch_memcpy_to_user
1021 arch_printchar
0786 arch_strncpy_from_user
0345 arch_task_create
0346 arch_task_destroy
0347 arch_task_switch
0013 halt
0570 handle_irq
0201 handle_page_fault
0972 handle_syscall
0667 ipc
1123 kdebug_handle_interrupt
1087 kdebug_readchar
0151 kfree
1035 klog_read
1053 klog_write
0028 kmain
0128 kmalloc
0340 lock
0216 memory_init
0342 mp_cpuid
0343 mp_num_cpus
0344 mp_reschedule
0011 mp_start
0042 mpmain
0752 notify
1068 printk
1160 stack_check
1155 stack_set_canary
0385 task_create
0429 task_destroy
0604 task_dump
0482 task_exit
0629 task_init
0540 task_listen_irq
0376 task_lookup
0330 task_notify
0499 task_set_state
0522 task_switch
0555 task_unlisten_irq
0341 unlock
0093 vm_create
0094 vm_destroy
0095 vm_link
0096 vm_unlink

```
0001 /* ============================================================================
0002  * kernel/main.h
0003  * ============================================================================ */
0004 #ifndef __MAIN_H__
0005 #define __MAIN_H__
0006
0007 void kmain(void);
0008 void mpmain(void);
0009
0010 // Implemented in arch.
0011 void mp_start(void);
0012 void arch_idle(void);
0013 void halt(void);
0014
0015 #endif
0016
0017 /* ============================================================================
0018  * kernel/main.c
0019  * ============================================================================ */
0020 #include "main.h"
0021 #include "kdebug.h"
0022 #include "memory.h"
0023 #include "printk.h"
0024 #include "syscall.h"
0025 #include "task.h"
0026
0027 /// Initializes the kernel and starts the first task.
0028 void kmain(void) {
0029     printf("\nBooting Resea...\n");
0030     memory_init();
0031     task_init();
0032     mp_start();
0033
0034     // Create the first userland task (init).
0035     struct task *task = task_lookup(INIT_TASK_TID);
0036     ASSERT(task);
0037     task_create(task, "init", INITFS_ADDR, 0, CAP_ALL);
0038
0039     mpmain();
0040 }
0041
0042 void mpmain(void) {
0043     stack_set_canary();
0044
0045     // Initialize the idle task for this CPU.
0046     IDLE_TASK->tid = 0;
0047     task_create(IDLE_TASK, "(idle)", 0, 0, CAP_IPC);
0048     CURRENT = IDLE_TASK;
0049
0050     // Do the very first context switch on this CPU.
0051     INFO("Booted CPU #%d", mp_cpuid());
0052     task_switch();
0053
```

```
0054     // We're now in the current CPU's idle task.
0055     while (true) {
0056         // Halt the CPU until an interrupt arrives...
0057         arch_idle();
0058         // Handled an interrupt. Try switching into a task resumed by an
0059         // interrupt message.
0060         task_switch();
0061     }
0062 }
0063
0064 /* ============================================================================
0065  * kernel/memory.h
0066  * ============================================================================ */
0067 #ifndef __MEMORY_H__
0068 #define __MEMORY_H__
0069
0070 #include <arch.h>
0071 #include <list.h>
0072 #include <types.h>
0073
0074 /// Unused element.
0075 struct free_list {
0076     uint64_t magic1;
0077     list_elem_t next;
0078     size_t num_pages;
0079     uint64_t magic2;
0080 };
0081
0082 #define FREE_LIST_MAGIC1   0xdeaddead
0083 #define FREE_LIST_MAGIC2   0xbadbadba
0084 #define STACK_CANARY_VALUE 0xdeadca71deadca71ULL
0085
0086 void *kmalloc(size_t size);
0087 void kfree(void *ptr);
0088 void handle_page_fault(vaddr_t addr, pagefault_t fault);
0089 void memory_init(void);
0090
0091 // Implemented in arch.
0092 struct vm;
0093 error_t vm_create(struct vm *vm);
0094 void vm_destroy(struct vm *vm);
0095 error_t vm_link(struct vm *vm, vaddr_t vaddr, paddr_t paddr, pageattrs_t attrs);
0096 void vm_unlink(struct vm *vm, vaddr_t vaddr);
0097
0098 #endif
0099
0100 /* ============================================================================
0101  * kernel/memory.c
0102  * ============================================================================ */
0103 #include "memory.h"
0104 #include <arch.h>
0105 #include <message.h>
0106 #include <string.h>
```

```
0107 #include "ipc.h"
0108 #include "printk.h"
0109 #include "syscall.h"
0110 #include "task.h"
0111
0112 extern char __kernel_heap[];
0113 extern char __kernel_heap_end[];
0114 extern char __initfs[];
0115
0116 static list_t heap;
0117
0118 static void add_free_list(void *addr, size_t num_pages) {
0119     struct free_list *free = addr;
0120     free->num_pages = num_pages;
0121     free->magic1 = FREE_LIST_MAGIC1;
0122     free->magic2 = FREE_LIST_MAGIC2;
0123     list_push_back(&heap, &free->next);
0124 }
0125
0126 /// Allocates a memory space in the kernel heap. The address is always aligned
0127 /// to PAGE_SIZE.
0128 void *kmalloc(size_t size) {
0129     if (list_is_empty(&heap)) {
0130         PANIC("Run out of kernel memory.");
0131     }
0132
0133     struct free_list *free =
0134         LIST_CONTAINER(list_pop_front(&heap), struct free_list, next);
0135
0136     ASSERT(size <= PAGE_SIZE);
0137     ASSERT(free->magic1 == FREE_LIST_MAGIC1);
0138     ASSERT(free->magic2 == FREE_LIST_MAGIC2);
0139     ASSERT(free->num_pages >= 1);
0140
0141     free->num_pages--;
0142     if (free->num_pages > 0) {
0143         list_push_back(&heap, &free->next);
0144     }
0145
0146     void *ptr = (void *) ((vaddr_t) free + free->num_pages * PAGE_SIZE);
0147     return ptr;
0148 }
0149
0150 /// Frees a memory.
0151 void kfree(void *ptr) {
0152     add_free_list(ptr, 1);
0153 }
0154
0155 /// Calls the pager task. It always returns a valid paddr: if the memory access
0156 /// is invalid, the pager kills the task instead of replying the page fault
0157 /// message.
0158 static paddr_t user_pager(vaddr_t addr, pagefault_t fault, pageattrs_t *attrs) {
0159     struct message m;
```

```
0160      m.type = PAGE_FAULT_MSG;
0161      m.page_fault.task = CURRENT->tid;
0162      m.page_fault.vaddr = addr;
0163      m.page_fault.fault = fault;
0164
0165      error_t err = ipc(CURRENT->pager, CURRENT->pager->tid, &m,
0166                        IPC_CALL | IPC_KERNEL);
0167      if (IS_ERROR(err)) {
0168          WARN("%s: aborted kernel ipc", CURRENT->name);
0169          task_exit(EXP_ABORTED_KERNEL_IPC);
0170      }
0171
0172      // Check if the reply is valid.
0173      if (m.type != PAGE_FAULT_REPLY_MSG) {
0174          WARN("%s: invalid page fault reply (type=%d, addr=%p, pager=%s)",
0175               CURRENT->name, m.type, addr, CURRENT->pager->name);
0176          task_exit(EXP_INVALID_PAGE_FAULT_REPLY);
0177      }
0178
0179      *attrs = PAGE_USER | m.page_fault_reply.attrs;
0180      return m.page_fault_reply.paddr;
0181  }
0182
0183  /// Handles page faults in the initial task.
0184  static paddr_t init_task_pager(vaddr_t vaddr, pageattrs_t *attrs) {
0185      paddr_t paddr;
0186      if (INITFS_ADDR <= vaddr && vaddr < INITFS_END) {
0187          // Initfs contents.
0188          paddr = into_paddr(__initfs + (vaddr - INITFS_ADDR));
0189      } else if (STRAIGHT_MAP_ADDR <= vaddr && vaddr < STRAIGHT_MAP_END) {
0190          // Straight-mapping: virtual addresses are equal to physical.
0191          paddr = vaddr;
0192      } else {
0193          PANIC("init_task tried to access invalid memory address %p", vaddr);
0194      }
0195
0196      *attrs = PAGE_USER | PAGE_WRITABLE;
0197      return paddr;
0198  }
0199
0200  /// The page fault handler. It calls a pager and updates the page table.
0201  void handle_page_fault(vaddr_t addr, pagefault_t fault) {
0202      // Ask the associated pager to resolve the page fault.
0203      vaddr_t aligned_vaddr = ALIGN_DOWN(addr, PAGE_SIZE);
0204      paddr_t paddr;
0205      pageattrs_t attrs;
0206      if (CURRENT->tid == INIT_TASK_TID) {
0207          paddr = init_task_pager(aligned_vaddr, &attrs);
0208      } else {
0209          paddr = user_pager(aligned_vaddr, fault, &attrs);
0210      }
0211
0212      vm_link(&CURRENT->vm, aligned_vaddr, paddr, attrs);
```

```
0213 }
0214
0215 /// Initializes the memory subsystem.
0216 void memory_init(void) {
0217     size_t heap_size = (vaddr_t) __kernel_heap_end - (vaddr_t) __kernel_heap;
0218     INFO("kernel heap: %p - %p (%dKiB)", (vaddr_t) __kernel_heap,
0219          (vaddr_t) __kernel_heap_end, heap_size / 1024);
0220     list_init(&heap);
0221     add_free_list((void *) __kernel_heap, heap_size / PAGE_SIZE);
0222 }
0223
0224 /* ==========================================================================
0225  * kernel/task.h
0226  * ========================================================================== */
0227 #ifndef __TASK_H__
0228 #define __TASK_H__
0229
0230 #include <arch.h>
0231 #include <message.h>
0232 #include <types.h>
0233 #include "memory.h"
0234
0235 #define TASK_TIME_SLICE ((10 * TICK_HZ) / 1000) /* 10 milliseconds */
0236 #define TASKS_MAX       16
0237 #define TASK_NAME_LEN   16
0238
0239 //
0240 // Task states.
0241 //
0242
0243 /// The task struct is not being used.
0244 #define TASK_UNUSED 0
0245 /// The task is being created.
0246 #define TASK_CREATED 1
0247 /// The task is running or is queued in the runqueue.
0248 #define TASK_RUNNABLE 2
0249 /// The task is waiting for a receiver task in IPC.
0250 #define TASK_SENDING 3
0251 /// The task is waiting for a sender task in IPC.
0252 #define TASK_RECEIVING 4
0253 /// The task has exited. Waiting for the pager to destructs it.
0254 #define TASK_EXITED 5
0255
0256 /// Determines if the current task has the given capability.
0257 #define CAPABLE(cap) ((CURRENT->caps & (cap)) != 0)
0258
0259 // struct arch_cpuvar *
0260 #define ARCH_CPUVAR (&get_cpuvar()->arch)
0261
0262 /// The current task of the current CPU (`struct task *`).
0263 #define CURRENT (get_cpuvar()->current_task)
0264 /// The idle task of the current CPU (`struct task *`).
0265 #define IDLE_TASK (&get_cpuvar()->idle_task)
```

```
0266
0267 /// The task struct (so-called Task Control Block).
0268 struct task {
0269     /// The arch-specific fields.
0270     struct arch_task arch;
0271     /// The task ID. Starts with 1.
0272     tid_t tid;
0273     /// The state.
0274     int state;
0275     /// The name of task terminated by NUL.
0276     char name[TASK_NAME_LEN];
0277     /// Capabilities (allowed operations).
0278     caps_t caps;
0279     /// The page table.
0280     struct vm vm;
0281     /// The pager task. When a page fault or an exception (e.g. divide by zero)
0282     /// occurred, the kernel sends a message to the pager to allow it to
0283     /// resolve the faults (or kill the task).
0284     struct task *pager;
0285     /// The remaining time slice in ticks. If this value reaches 0, the kernel
0286     /// switches into the next task (so-called preemptive context switching).
0287     unsigned quantum;
0288     /// The message buffer.
0289     struct message buffer;
0290     /// The acceptable sender task ID. If it's IPC_ANY, the task accepts
0291     /// messages from any tasks.
0292     tid_t src;
0293     /// The pending notifications. It's cleared when the task received them as
0294     /// an message (NOTIFICATIONS_MSG).
0295     notifications_t notifications;
0296     /// The IPC timeout in milliseconds. When it become 0, the kernel notify the
0297     /// task with `NOTIFY_TIMER`.
0298     msec_t timeout;
0299     /// The queue of tasks that are waiting for this task to get ready for
0300     /// receiving a message. If this task gets ready, it resumes all threads in
0301     /// this queue.
0302     list_t senders;
0303     /// The table tasks that are waiting for this task to get ready for
0304     /// receiving a message. When this task become TASK_RECEIVING, it notifies
0305     /// all threads registered in this table with `NOTIFY_READY`.
0306     ///
0307     /// Note that task ID is 1-origin, i.e., `listened_by[1]` is used by task
0308     /// #2, not #1.
0309     ///
0310     /// FIXME: This requires O(n) operaions.
0311     bool listened_by[TASKS_MAX];
0312     /// A (intrusive) list element in the runqueue.
0313     list_elem_t runqueue_next;
0314     /// A (intrusive) list element in a sender queue.
0315     list_elem_t sender_next;
0316 };
0317
0318 /// CPU-local variables.
```

```
0319 struct cpuvar {
0320     struct arch_cpuvar arch;
0321     struct task *current_task;
0322     struct task idle_task;
0323 };
0324
0325 error_t task_create(struct task *task, const char *name, vaddr_t ip,
0326                     struct task *pager, caps_t caps);
0327 error_t task_destroy(struct task *task);
0328 NORETURN void task_exit(enum exception_type exp);
0329 void task_set_state(struct task *task, int state);
0330 void task_notify(struct task *task, notifications_t notifications);
0331 struct task *task_lookup(tid_t tid);
0332 void task_switch(void);
0333 error_t task_listen_irq(struct task *task, unsigned irq);
0334 error_t task_unlisten_irq(struct task *task, unsigned irq);
0335 void handle_irq(unsigned irq);
0336 void task_dump(void);
0337 void task_init(void);
0338
0339 // Implemented in arch.
0340 void lock(void);
0341 void unlock(void);
0342 int mp_cpuid(void);
0343 int mp_num_cpus(void);
0344 void mp_reschedule(void);
0345 error_t arch_task_create(struct task *task, vaddr_t ip);
0346 void arch_task_destroy(struct task *task);
0347 void arch_task_switch(struct task *prev, struct task *next);
0348 void arch_enable_irq(unsigned irq);
0349 void arch_disable_irq(unsigned irq);
0350
0351 #endif
0352
0353 /* ===========================================================================
0354  * kernel/task.c
0355  * =========================================================================== */
0356 #include "task.h"
0357 #include <arch.h>
0358 #include <list.h>
0359 #include <string.h>
0360 #include "ipc.h"
0361 #include "kdebug.h"
0362 #include "memory.h"
0363 #include "message.h"
0364 #include "printk.h"
0365 #include "syscall.h"
0366
0367 /// All tasks.
0368 static struct task tasks[TASKS_MAX];
0369 /// A queue of runnable tasks excluding currently running tasks.
0370 static list_t runqueue;
0371 /// IRQ owners.
```

```
0372 static struct task *irq_owners[IRQ_MAX];
0373
0374 /// Returns the task struct for the task ID. It returns NULL if the ID is
0375 /// invalid.
0376 struct task *task_lookup(tid_t tid) {
0377     if (tid <= 0 || tid > TASKS_MAX) {
0378         return NULL;
0379     }
0380
0381     return &tasks[tid - 1];
0382 }
0383
0384 /// Initializes a task struct.
0385 error_t task_create(struct task *task, const char *name, vaddr_t ip,
0386                     struct task *pager, caps_t caps) {
0387     if (task->state != TASK_UNUSED) {
0388         return ERR_ALREADY_EXISTS;
0389     }
0390
0391     // Initialize the page table.
0392     error_t err;
0393     if ((err = vm_create(&task->vm)) != OK) {
0394         return err;
0395     }
0396
0397     // Do arch-specific initialization.
0398     if ((err = arch_task_create(task, ip)) != OK) {
0399         vm_destroy(&task->vm);
0400         return err;
0401     }
0402
0403     // Initialize fields.
0404     TRACE("new task #%d: %s", task->tid, name);
0405     task->state = TASK_CREATED;
0406     task->caps = caps;
0407     task->notifications = 0;
0408     task->pager = pager;
0409     task->timeout = 0;
0410     task->quantum = 0;
0411     strncpy(task->name, name, sizeof(task->name));
0412     list_init(&task->senders);
0413     list_nullify(&task->runqueue_next);
0414     list_nullify(&task->sender_next);
0415
0416     for (unsigned i = 0; i < TASKS_MAX; i++) {
0417         task->listened_by[i] = false;
0418     }
0419
0420     // Append the newly created task into the runqueue.
0421     if (task != IDLE_TASK) {
0422         task_set_state(task, TASK_RUNNABLE);
0423     }
0424
```

```
0425     return OK;
0426 }
0427
0428 /// Frees the task data structures and make it unused.
0429 error_t task_destroy(struct task *task) {
0430     ASSERT(task != CURRENT);
0431     ASSERT(task != IDLE_TASK);
0432
0433     if (task->tid == INIT_TASK_TID) {
0434         TRACE("%s: tried to destroy the init task", task->name);
0435         return ERR_INVALID_ARG;
0436     }
0437
0438     if (task->state == TASK_UNUSED) {
0439         return ERR_INVALID_ARG;
0440     }
0441
0442     TRACE("destroying %s...", task->name);
0443     list_remove(&task->runqueue_next);
0444     list_remove(&task->sender_next);
0445     vm_destroy(&task->vm);
0446     arch_task_destroy(task);
0447     task->state = TASK_UNUSED;
0448
0449     // Abort sender IPC operations.
0450     LIST_FOR_EACH (sender, &task->senders, struct task, sender_next) {
0451         notify(sender, NOTIFY_ABORTED);
0452         list_remove(&sender->sender_next);
0453     }
0454
0455     for (unsigned i = 0; i < TASKS_MAX; i++) {
0456         // Ensure that this task is not a pager task.
0457         if (tasks[i].pager == task) {
0458             PANIC("a pager task '%s' (#%d) is being killed", task->name,
0459                   task->tid);
0460         }
0461
0462         // Notify all listener tasks that this task has been aborted.
0463         if (task->listened_by[i]) {
0464             notify(task_lookup(i + 1), NOTIFY_ABORTED);
0465         }
0466
0467         // Unlisten from each task.
0468         tasks[i].listened_by[task->tid - 1] = false;
0469     }
0470
0471     for (unsigned irq = 0; irq < IRQ_MAX; irq++) {
0472         if (irq_owners[irq] == task) {
0473             arch_disable_irq(irq);
0474             irq_owners[irq] = NULL;
0475         }
0476     }
0477
```

```
0478     return OK;
0479 }
0480
0481 /// Exits the current task. `exp` is the reason why the task is being exited.
0482 NORETURN void task_exit(enum exception_type exp) {
0483     ASSERT(CURRENT != IDLE_TASK);
0484
0485     // Tell its pager that this task has exited.
0486     struct message m;
0487     m.type = EXCEPTION_MSG;
0488     m.exception.task = CURRENT->tid;
0489     m.exception.exception = exp;
0490     ipc(CURRENT->pager, 0, &m, IPC_SEND | IPC_KERNEL);
0491
0492     // Wait until the pager task destroys this task...
0493     CURRENT->state = TASK_EXITED;
0494     task_switch();
0495     UNREACHABLE();
0496 }
0497
0498 /// Updates a task's state.
0499 void task_set_state(struct task *task, int state) {
0500     DEBUG_ASSERT(task->state != state);
0501
0502     task->state = state;
0503     if (state == TASK_RUNNABLE) {
0504         list_push_back(&runqueue, &task->runqueue_next);
0505         mp_reschedule();
0506     }
0507 }
0508
0509 /// Picks the next task to run.
0510 static struct task *scheduler(struct task *current) {
0511     if (current != IDLE_TASK && current->state == TASK_RUNNABLE) {
0512         // The current task is still runnable. Enqueue into the runqueue.
0513         list_push_back(&runqueue, &current->runqueue_next);
0514     }
0515
0516     struct task *next = LIST_POP_FRONT(&runqueue, struct task, runqueue_next);
0517     return (next) ? next : IDLE_TASK;
0518 }
0519
0520 /// Do a context switch: save the current register state on the stack and
0521 /// restore the next thread's state.
0522 void task_switch(void) {
0523     stack_check();
0524
0525     struct task *prev = CURRENT;
0526     struct task *next = scheduler(prev);
0527     next->quantum = TASK_TIME_SLICE;
0528     if (next == prev) {
0529         // No runnable threads other than the current one. Continue executing
0530         // the current thread.
```

```
0531            return;
0532        }
0533
0534        CURRENT = next;
0535        arch_task_switch(prev, next);
0536
0537        stack_check();
0538    }
0539
0540    error_t task_listen_irq(struct task *task, unsigned irq) {
0541        if (irq >= IRQ_MAX) {
0542            return ERR_INVALID_ARG;
0543        }
0544
0545        if (irq_owners[irq]) {
0546            return ERR_ALREADY_EXISTS;
0547        }
0548
0549        irq_owners[irq] = task;
0550        arch_enable_irq(irq);
0551        TRACE("enabled IRQ: task=%s, vector=%d", task->name, irq);
0552        return OK;
0553    }
0554
0555    error_t task_unlisten_irq(struct task *task, unsigned irq) {
0556        if (irq >= IRQ_MAX) {
0557            return ERR_INVALID_ARG;
0558        }
0559
0560        if (irq_owners[irq] != task) {
0561            return ERR_NOT_PERMITTED;
0562        }
0563
0564        arch_disable_irq(irq);
0565        irq_owners[irq] = NULL;
0566        TRACE("disabled IRQ: task=%s, vector=%d", task->name, irq);
0567        return OK;
0568    }
0569
0570    void handle_irq(unsigned irq) {
0571        if (irq == TIMER_IRQ) {
0572            // Handles timer interrupts. The timer fires this IRQ every 1/TICK_HZ
0573            // seconds.
0574
0575            // Handle task timeouts.
0576            if (mp_is_bsp()) {
0577                for (int i = 0; i < TASKS_MAX; i++) {
0578                    struct task *task = &tasks[i];
0579                    if (task->state == TASK_UNUSED || !task->timeout) {
0580                        continue;
0581                    }
0582
0583                    task->timeout--;
```

```
0584                if (!task->timeout) {
0585                    notify(task, NOTIFY_TIMER);
0586                }
0587            }
0588        }
0589
0590        // Switch task if the current task has spend its time slice.
0591        DEBUG_ASSERT(CURRENT->quantum > 0);
0592        CURRENT->quantum--;
0593        if (!CURRENT->quantum) {
0594            task_switch();
0595        }
0596    } else {
0597        struct task *owner = irq_owners[irq];
0598        if (owner) {
0599            notify(owner, NOTIFY_IRQ);
0600        }
0601    }
0602 }
0603
0604 void task_dump(void) {
0605     const char *states[] = {
0606         [TASK_UNUSED] = "unused",         [TASK_CREATED] = "created",
0607         [TASK_EXITED] = "exited",         [TASK_RUNNABLE] = "runnable",
0608         [TASK_RECEIVING] = "receiveing", [TASK_SENDING] = "sending",
0609     };
0610
0611     for (unsigned i = 0; i < TASKS_MAX; i++) {
0612         struct task *task = &tasks[i];
0613         if (task->state == TASK_UNUSED) {
0614             continue;
0615         }
0616
0617         DPRINTK("#%d %s: state=%s, src=%d\n", task->tid, task->name,
0618                 states[task->state], task->src);
0619         if (!list_is_empty(&task->senders)) {
0620             DPRINTK("  senders:\n");
0621             LIST_FOR_EACH (sender, &task->senders, struct task, sender_next) {
0622                 DPRINTK("    - #%d %s\n", sender->tid, sender->name);
0623             }
0624         }
0625     }
0626 }
0627
0628 /// Initializes the task subsystem.
0629 void task_init(void) {
0630     list_init(&runqueue);
0631     for (int i = 0; i < TASKS_MAX; i++) {
0632         tasks[i].state = TASK_UNUSED;
0633         tasks[i].tid = i + 1;
0634     }
0635
0636     for (int i = 0; i < IRQ_MAX; i++) {
```

```
0637         irq_owners[i] = NULL;
0638     }
0639 }
0640
0641 /* ============================================================================
0642  * kernel/ipc.h
0643  * ============================================================================ */
0644 #ifndef __IPC_H__
0645 #define __IPC_H__
0646
0647 #include <types.h>
0648
0649 struct task;
0650 struct message;
0651 error_t ipc(struct task *dst, tid_t src, struct message *m, unsigned flags);
0652 void notify(struct task *dst, notifications_t notifications);
0653
0654 #endif
0655
0656 /* ============================================================================
0657  * kernel/ipc.c
0658  * ============================================================================ */
0659 #include <list.h>
0660 #include <string.h>
0661 #include <types.h>
0662 #include "ipc.h"
0663 #include "printk.h"
0664 #include "task.h"
0665
0666 /// Sends and receives a message.
0667 error_t ipc(struct task *dst, tid_t src, struct message *m, unsigned flags) {
0668     if (flags & IPC_TIMER) {
0669         CURRENT->timeout = POW2(IPC_TIMEOUT(flags));
0670     }
0671
0672     // Register the current task as a listener.
0673     if (flags & IPC_LISTEN) {
0674         dst->listened_by[CURRENT->tid - 1] = true;
0675         return OK;
0676     }
0677
0678     // Send a message.
0679     if (flags & IPC_SEND) {
0680         // Wait until the destination (receiver) task gets ready for receiving.
0681         while (true) {
0682             if (dst->state == TASK_RECEIVING
0683                 && (dst->src == IPC_ANY || dst->src == CURRENT->tid)) {
0684                 break;
0685             }
0686
0687             if (flags & IPC_NOBLOCK) {
0688                 return ERR_WOULD_BLOCK;
0689             }
```

```
0690
0691                // The receiver task is not ready. Sleep until it resumes the
0692                // current task.
0693                task_set_state(CURRENT, TASK_SENDING);
0694                list_push_back(&dst->senders, &CURRENT->sender_next);
0695                task_switch();
0696
0697                if (CURRENT->notifications & NOTIFY_ABORTED) {
0698                    // The receiver task has exited. Abort the system call.
0699                    CURRENT->notifications &= ~NOTIFY_ABORTED;
0700                    return ERR_ABORTED;
0701                }
0702            }
0703
0704            // Copy the message into the receiver's buffer and resume it.
0705            memcpy(&dst->buffer, m, sizeof(struct message));
0706            dst->buffer.src = (flags & IPC_KERNEL) ? KERNEL_TASK_TID : CURRENT->tid;
0707            task_set_state(dst, TASK_RUNNABLE);
0708        }
0709
0710        // Receive a message.
0711        if (flags & IPC_RECV) {
0712            // Check if there're pending notifications.
0713            if (src == IPC_ANY && CURRENT->notifications) {
0714                // Receive the pending notifications.
0715                m->type = NOTIFICATIONS_MSG;
0716                m->src = KERNEL_TASK_TID;
0717                m->notifications.data = CURRENT->notifications;
0718                CURRENT->notifications = 0;
0719                return OK;
0720            }
0721
0722            // Resume a sender task.
0723            LIST_FOR_EACH (sender, &CURRENT->senders, struct task, sender_next) {
0724                if (src == IPC_ANY || src == sender->tid) {
0725                    task_set_state(sender, TASK_RUNNABLE);
0726                    list_remove(&sender->sender_next);
0727                    break;
0728                }
0729            }
0730
0731            // Notify the listeners that this task is now waiting for a message.
0732            for (unsigned i = 0; i < TASKS_MAX; i++) {
0733                if (CURRENT->listened_by[i]) {
0734                    notify(task_lookup(i + 1), NOTIFY_READY);
0735                    CURRENT->listened_by[i] = false;
0736                }
0737            }
0738
0739            // Sleep until a sender task resumes this task...
0740            CURRENT->src = src;
0741            task_set_state(CURRENT, TASK_RECEIVING);
0742            task_switch();
```

```
0743
0744         // Received a message. Copy it into the receiver buffer and return.
0745         memcpy(m, &CURRENT->buffer, sizeof(struct message));
0746     }
0747
0748     return OK;
0749 }
0750
0751 // Notifies notifications to the task.
0752 void notify(struct task *dst, notifications_t notifications) {
0753     if (dst->state == TASK_RECEIVING && dst->src == IPC_ANY) {
0754         // Send a NOTIFICATIONS_MSG message immediately.
0755         dst->buffer.type = NOTIFICATIONS_MSG;
0756         dst->buffer.src = KERNEL_TASK_TID;
0757         dst->buffer.notifications.data = dst->notifications | notifications;
0758         dst->notifications = 0;
0759         task_set_state(dst, TASK_RUNNABLE);
0760     } else {
0761         // The task is not ready for receiving a event message: update the
0762         // pending notifications instead.
0763         dst->notifications |= notifications;
0764     }
0765 }
0766
0767 /* ============================================================================
0768  * kernel/syscall.h
0769  * ============================================================================ */
0770 #ifndef __SYSCALL_H__
0771 #define __SYSCALL_H__
0772
0773 #include <types.h>
0774
0775 /// A pointer given by the user. Don't reference it directly; access it through
0776 /// safe functions such as memcpy_from_user and memcpy_to_user!
0777 typedef vaddr_t userptr_t;
0778
0779 uintmax_t handle_syscall(uintmax_t syscall, uintmax_t arg1, uintmax_t arg2,
0780                          uintmax_t arg3, uintmax_t arg4, uintmax_t arg5);
0781
0782 // Implemented in arch.
0783 struct task;
0784 void arch_memcpy_from_user(void *dst, userptr_t src, size_t len);
0785 void arch_memcpy_to_user(userptr_t dst, const void *src, size_t len);
0786 void arch_strncpy_from_user(char *dst, userptr_t src, size_t max_len);
0787
0788 #endif
0789
0790 /* ============================================================================
0791  * kernel/syscall.c
0792  * ============================================================================ */
0793 #include <arch.h>
0794 #include <list.h>
0795 #include <string.h>
```

```
0796 #include <types.h>
0797 #include "interrupt.h"
0798 #include "ipc.h"
0799 #include "kdebug.h"
0800 #include "memory.h"
0801 #include "printk.h"
0802 #include "syscall.h"
0803 #include "task.h"
0804
0805 /// Copies bytes from the userspace. If the user's pointer is invalid, this
0806 /// function or the page fault handler kills the current task.
0807 static void memcpy_from_user(void *dst, userptr_t src, size_t len) {
0808     if (is_kernel_addr_range(src, len)) {
0809         task_exit(EXP_INVALID_MEMORY_ACCESS);
0810     }
0811
0812     arch_memcpy_from_user(dst, src, len);
0813 }
0814
0815 /// Copies bytes into the userspace. If the user's pointer is invalid, this
0816 /// function or the page fault handler kills the current task.
0817 static void memcpy_to_user(userptr_t dst, const void *src, size_t len) {
0818     if (is_kernel_addr_range(dst, len)) {
0819         task_exit(EXP_INVALID_MEMORY_ACCESS);
0820     }
0821
0822     arch_memcpy_to_user(dst, src, len);
0823 }
0824
0825 /// Copy a string terminated by NUL from the userspace. If the user's pointer is
0826 /// invalid, this function or the page fault handler kills the current task.
0827 static void strncpy_from_user(char *dst, userptr_t src, size_t max_len) {
0828     if (is_kernel_addr_range(src, max_len)) {
0829         task_exit(EXP_INVALID_MEMORY_ACCESS);
0830     }
0831
0832     arch_strncpy_from_user(dst, src, max_len);
0833 }
0834
0835 static error_t sys_ipc(tid_t dst, tid_t src, userptr_t m, unsigned flags) {
0836     struct message buf;
0837
0838     if (!CAPABLE(CAP_IPC)) {
0839         return ERR_NOT_PERMITTED;
0840     }
0841
0842     if (flags & IPC_KERNEL) {
0843         return ERR_INVALID_ARG;
0844     }
0845
0846     if (src < 0 || src > TASKS_MAX) {
0847         return ERR_INVALID_ARG;
0848     }
```

```
0849
0850     struct task *dst_task = NULL;
0851     if (flags & (IPC_SEND | IPC_LISTEN)) {
0852         dst_task = task_lookup(dst);
0853         if (!dst_task) {
0854             return ERR_INVALID_ARG;
0855         }
0856     }
0857
0858     if (flags & IPC_SEND) {
0859         memcpy_from_user(&buf, m, sizeof(struct message));
0860     }
0861
0862     error_t err = ipc(dst_task, src, &buf, flags);
0863     if (IS_ERROR(err)) {
0864         return err;
0865     }
0866
0867     if (flags & IPC_RECV) {
0868         memcpy_to_user(m, &buf, sizeof(struct message));
0869     }
0870
0871     return OK;
0872 }
0873
0874 /// The taskctl system call does all task-related operations. The operation is
0875 /// determined as below:
0876 ///
0877 ///        | task_create | task_destroy | task_exit | task_self | caps_drop
0878 ///  ------+-------------+--------------+-----------+-----------+-----------
0879 ///  tid   |     > 0     |      > 0     |     0     |    ---    |     ---
0880 ///  pager |     > 0     |       0      |     0     |    -1     |     -1
0881 ///
0882 static tid_t sys_taskctl(tid_t tid, userptr_t name, vaddr_t ip, tid_t pager,
0883                          caps_t caps) {
0884     // Since task_exit(), task_self(), and caps_drop() are unprivileged, we
0885     // don't need to check the capabilities here.
0886     if (!tid && !pager) {
0887         task_exit(EXP_GRACE_EXIT);
0888     }
0889
0890     if (pager < 0) {
0891         // Do caps_drop() and task_self() at once.
0892         CURRENT->caps &= ~caps;
0893         return CURRENT->tid;
0894     }
0895
0896     // Check the capability before handling privileged operations.
0897     if (!CAPABLE(CAP_TASK)) {
0898         return ERR_NOT_PERMITTED;
0899     }
0900
0901     // Look for the target task.
```

```
0902     struct task *task = task_lookup(tid);
0903     if (!task || task == CURRENT) {
0904         return ERR_INVALID_ARG;
0905     }
0906
0907     if (pager) {
0908         struct task *pager_task = task_lookup(pager);
0909         if (!pager_task) {
0910             return ERR_INVALID_ARG;
0911         }
0912
0913         // Create a task.
0914         char namebuf[TASK_NAME_LEN];
0915         strncpy_from_user(namebuf, name, sizeof(namebuf));
0916         return task_create(task, namebuf, ip, pager_task, CURRENT->caps & caps);
0917     } else {
0918         // Destroy the task.
0919         return task_destroy(task);
0920     }
0921 }
0922
0923 static error_t sys_irqctl(unsigned irq, bool enable) {
0924     if (!CAPABLE(CAP_IO)) {
0925         return ERR_NOT_PERMITTED;
0926     }
0927
0928     if (enable) {
0929         return task_listen_irq(CURRENT, irq);
0930     } else {
0931         return task_unlisten_irq(CURRENT, irq);
0932     }
0933 }
0934
0935 static int sys_klogctl(userptr_t buf, size_t buf_len, bool write) {
0936     if (!CAPABLE(CAP_KLOG)) {
0937         return ERR_NOT_PERMITTED;
0938     }
0939
0940     if (write) {
0941         char kbuf[256];
0942         int remaining = buf_len;
0943         while (remaining > 0) {
0944             int copy_len = MIN(remaining, (int) sizeof(kbuf));
0945             memcpy_from_user(kbuf, buf, copy_len);
0946             for (int i = 0; i < copy_len; i++) {
0947                 printk("%c", kbuf[i]);
0948             }
0949             remaining -= copy_len;
0950         }
0951
0952         return OK;
0953     } else {
0954         char kbuf[256];
```

```
0955        int remaining = buf_len;
0956        while (remaining > 0) {
0957            int read_len = klog_read(kbuf, MIN(remaining, (int) sizeof(kbuf)));
0958            if (!read_len) {
0959                break;
0960            }
0961
0962            memcpy_to_user(buf, kbuf, read_len);
0963            buf += read_len;
0964            remaining -= read_len;
0965        }
0966
0967        return buf_len - remaining;
0968    }
0969 }
0970
0971 /// The system call handler.
0972 uintmax_t handle_syscall(uintmax_t syscall, uintmax_t arg1, uintmax_t arg2,
0973                          uintmax_t arg3, uintmax_t arg4, uintmax_t arg5) {
0974     stack_check();
0975
0976     uintmax_t ret;
0977     switch (syscall) {
0978         case SYSCALL_IPC:
0979             ret = (uintmax_t) sys_ipc(arg1, arg2, arg3, arg4);
0980             break;
0981         case SYSCALL_TASKCTL:
0982             ret = (uintmax_t) sys_taskctl(arg1, arg2, arg3, arg4, arg5);
0983             break;
0984         case SYSCALL_IRQCTL:
0985             ret = (uintmax_t) sys_irqctl(arg1, arg2);
0986             break;
0987         case SYSCALL_KLOGCTL:
0988             ret = (uintmax_t) sys_klogctl(arg1, arg2, arg3);
0989             break;
0990         default:
0991             return ERR_INVALID_ARG;
0992     }
0993
0994     stack_check();
0995     return ret;
0996 }
0997
0998 /* ==========================================================================
0999  * kernel/printk.h
1000  * ========================================================================== */
1001 #ifndef __PRINTK_H__
1002 #define __PRINTK_H__
1003
1004 #include <print_macros.h>
1005 #include <types.h>
1006
1007 #define KLOG_BUF_SIZE 4096
```

```
1008
1009  /// The kernel log (ring) buffer.
1010  struct klog {
1011      char buf[KLOG_BUF_SIZE];
1012      size_t head;
1013      size_t tail;
1014  };
1015
1016  void klog_write(char ch);
1017  size_t klog_read(char *buf, size_t buf_len);
1018  void printk(const char *fmt, ...);
1019
1020  // Implemented in arch.
1021  void arch_printchar(char ch);
1022
1023  #endif
1024
1025  /* ============================================================================
1026   * kernel/printk.c
1027   * ============================================================================ */
1028  #include "printk.h"
1029  #include <string.h>
1030  #include <vprintf.h>
1031
1032  static struct klog klog;
1033
1034  /// Reads the kernel log buffer.
1035  size_t klog_read(char *buf, size_t buf_len) {
1036      size_t remaining = buf_len;
1037      if (klog.tail > klog.head) {
1038          int copy_len = MIN(remaining, KLOG_BUF_SIZE - klog.tail);
1039          memcpy(buf, &klog.buf[klog.tail], copy_len);
1040          buf += copy_len;
1041          remaining -= copy_len;
1042          klog.tail = 0;
1043      }
1044
1045      int copy_len = MIN(remaining, klog.head - klog.tail);
1046      memcpy(buf, &klog.buf[klog.tail], copy_len);
1047      remaining -= copy_len;
1048      klog.tail = (klog.tail + copy_len) % KLOG_BUF_SIZE;
1049      return buf_len - remaining;
1050  }
1051
1052  /// Writes a character into the kernel log buffer.
1053  void klog_write(char ch) {
1054      klog.buf[klog.head] = ch;
1055      klog.head = (klog.head + 1) % KLOG_BUF_SIZE;
1056      if (klog.head == klog.tail) {
1057          // The buffer is full. Discard a character by moving the tail.
1058          klog.tail = (klog.tail + 1) % KLOG_BUF_SIZE;
1059      }
1060  }
```

```
1061
1062 static void printchar(UNUSED struct vprintf_context *ctx, char ch) {
1063     arch_printchar(ch);
1064     klog_write(ch);
1065 }
1066
1067 /// Prints a message. See vprintf() for detailed formatting specifications.
1068 void printk(const char *fmt, ...) {
1069     struct vprintf_context ctx = { .printchar = printchar };
1070     va_list vargs;
1071     va_start(vargs, fmt);
1072     vprintf(&ctx, fmt, vargs);
1073     va_end(vargs);
1074 }
1075
1076 /* ============================================================================
1077  * kernel/kdebug.h
1078  * ============================================================================ */
1079 #ifndef __KDEBUG_H__
1080 #define __KDEBUG_H__
1081
1082 void kdebug_handle_interrupt(void);
1083 void stack_check(void);
1084 void stack_set_canary(void);
1085
1086 // Implemented in arch.
1087 int kdebug_readchar(void);
1088
1089 #endif
1090
1091 /* ============================================================================
1092  * kernel/kdebug.c
1093  * ============================================================================ */
1094 #include "kdebug.h"
1095 #include <string.h>
1096 #include "task.h"
1097
1098 static void quit(void) {
1099 #ifdef __x86_64__
1100     // Quit the QEMU (-device isa-debug-exit).
1101     __asm__ __volatile__("outw %%ax, %%dx" ::"a"(0x2000), "Nd"(0x604));
1102 #endif
1103
1104     PANIC("halted by the kdebug");
1105 }
1106
1107 static void run(const char *cmdline) {
1108     if (strcmp(cmdline, "help") == 0) {
1109         DPRINTK("Kernel debugger commands:\n");
1110         DPRINTK("\n");
1111         DPRINTK("  ps - List tasks.\n");
1112         DPRINTK("  q  - Quit the emulator.\n");
1113         DPRINTK("\n");
```

```
1114      } else if (strcmp(cmdline, "ps") == 0) {
1115          task_dump();
1116      } else if (strcmp(cmdline, "q") == 0) {
1117          quit();
1118      } else {
1119          WARN("Invalid debugger command: '%s'.", cmdline);
1120      }
1121  }
1122
1123  void kdebug_handle_interrupt(void) {
1124      static char cmdline[128];
1125      static unsigned long cursor = 0;
1126      int ch;
1127      while ((ch = kdebug_readchar()) > 0) {
1128          if (ch == '\r') {
1129              printk("\n");
1130              cmdline[cursor] = '\0';
1131              if (cursor > 0) {
1132                  run(cmdline);
1133                  cursor = 0;
1134              }
1135              DPRINTK("kdebug> ");
1136              continue;
1137          }
1138
1139          DPRINTK("%c", ch);
1140          cmdline[cursor++] = (char) ch;
1141
1142          if (cursor > sizeof(cmdline) - 1) {
1143              WARN("Too long kernel debugger command.");
1144              cursor = 0;
1145          }
1146      }
1147  }
1148
1149  static uint64_t *get_canary_ptr(void) {
1150      uint64_t rbp = (uint64_t) __builtin_frame_address(0);
1151      return (uint64_t *) ALIGN_DOWN(rbp, PAGE_SIZE);
1152  }
1153
1154  /// Writes the stack canary at the borrom of the current kernel stack.
1155  void stack_set_canary(void) {
1156      *get_canary_ptr() = STACK_CANARY_VALUE;
1157  }
1158
1159  /// Checks that the kernel stack canary is still alive.
1160  void stack_check(void) {
1161      if (*get_canary_ptr() != STACK_CANARY_VALUE) {
1162          PANIC("the kernel stack has been exhausted");
1163      }
1164  }
1165
```

著者：怒田 晟也 <nuta@seiya.me>
発行：2020 年 2 月 29 日