

オペレーティングシステム
Ver. 0.0.0

徳山工業高等専門学校
情報電子工学科

Copyright © 2017 by
Dept. of Computer Science and Electronic Engineering,
Tokuyama College of Technology, JAPAN

本ドキュメントは CC-BY-SA 4.0 ライセンスによって許諾されています。
本ドキュメントは CC-BY-SA 3.0 de ライセンス、CC-BY-SA 4.0 ライセンスで許諾された著作物
を含みます。
(CC-BY-SA 3.0 de ライセンス全文は <https://creativecommons.org/licenses/by-sa/3.0/de/> で、CC-BY-SA 4.0 ライセンス全文は <https://creativecommons.org/licenses/by-sa/4.0/deed.ja> で確認できます。)

目次

第I部 OS の機能を使用してみよう	1
第II部 OS とは	3
第1章 オペレーティングシステムとは	5
1.1 オペレーティングシステムの役割	5
1.1.1 拡張マシンとしてのオペレーティングシステム	5
1.1.2 ハードウェア管理プログラムとしてのオペレーティングシステム	6
1.2 オペレーティングシステムの歴史	7
1.2.1 第1世代（1945～1955, 真空管の時代）	7
1.2.2 第2世代（1955～1965, ドラムマシンの時代）	7
1.2.3 第3世代（1966～1980, ICとマルチプログラミングの時代）	9
1.2.4 第4世代（1980～現代, PCの時代）	11
1.2.5 インターネット世代	14
1.3 まとめ	14
第2章 前提知識	17
2.1 コンピュータのハードウェア構成	17
2.2 CPUの構成	19
2.3 最近のコンピュータの実際の構成	19
2.3.1 デスクトップ・パーソナルコンピュータ	20
2.3.2 サーバコンピュータ	20
2.4 オペレーティングシステムの構造	20
2.4.1 カーネルの構成	20
2.4.2 カーネルの動作概要	21
2.4.3 プロセスの構造	23
2.5 カーネルの構成方式	24
2.5.1 単層カーネル（モノリシック・カーネル）	24

2.5.2 マイクロカーネル (micro-kernel)	25
2.6 TaC	26
2.6.1 ハードウェア構成	26
2.6.2 TacOS	28
2.7 もう一つの仮想マシン	28
2.7.1 Type 2 ハイパーバイザ	29
2.7.2 Type 1 ハイパーバイザ	29
2.7.3 仮想アプライアンス	30
2.8 まとめ	30
第 III 部 CPU 管理	31
第 3 章 CPU の仮想化	33
3.1 時分割多重	33
3.2 プロセスの状態	34
3.2.1 基本的な三つの状態	34
3.2.2 状態遷移	34
3.3 プロセスの切換え	35
3.3.1 切換えの原因	35
3.3.2 切換え手順	36
3.3.3 切換えの例	37
3.4 PCB (Process Control Block)	39
3.4.1 PCB の内容	39
3.4.2 PCB リスト	40
3.5 TacOS の CPU 仮想化	41
3.5.1 PCB	41
3.5.2 メモリ配置	43
3.5.3 プロセス切換えプログラム	45
3.6 スレッド (Thread)	46
3.6.1 スレッドの役割	46
3.6.2 スレッドの形式	49
3.6.3 スレッドプログラミング	51
第 4 章 CPU スケジューリング	55
4.1 評価基準	55
4.2 システムごとの目標	56
4.3 プロセスの振舞	57
4.3.1 CPU バウンドプロセス	57

4.3.2 I/O バウンドプロセス	58
4.4 スケジューリング方式	58
4.4.1 First-Come, First-Served (FCFS) スケジューリング	58
4.4.2 Shortest-Job-First (SJF) スケジューリング	59
4.4.3 Shortest-Remaining-Time-First (SRTF) スケジューリング	59
4.4.4 Round-Robin (RR) スケジューリング	60
4.4.5 Priority (優先度順) スケジューリング	61
4.4.6 Multilevel Feedback Queue (FB) スケジューリング	61
4.5 TacOS のスケジューラ	61
第 5 章 プロセス同期	65
5.1 競合 (Race Condition, Competition)	65
5.2 クリティカルセクション (CriticalSection)	66
5.3 相互排除 (Mutual Exclusion)	66
5.3.1 割込み禁止	67
5.3.2 専用命令を用いる方式	67
5.3.3 フラグを用いる方式	70
5.4 セマフォ (Semaphore)	70
5.4.1 概要	71
5.4.2 相互排除問題の解	72
5.4.3 生産者と消費者問題 (Producer-Consumer Problem) の解	72
5.4.4 複数生産者と複数消費者問題 (Producer-Consumer Problem) の解	74
5.4.5 リーダ・ライタ問題 (Readers-Writers Problem) の解	75
5.5 TacOS のセマフォ	77
5.5.1 データ構造	77
5.5.2 使用例	78
5.5.3 割当	79
5.5.4 P 操作ルーチン	81
5.5.5 V 操作ルーチン	82
5.5.6 setPri() 関数	83
5.6 まとめ	84
第 6 章 プロセス間通信	87
6.1 共有メモリ	87
6.1.1 UNIX の共有メモリ関連システムコール等	88
6.1.2 UNIX の共有メモリ使用例	89
6.2 メッセージ通信	89
6.2.1 通信相手の指定方式 (Naming)	91

6.2.2	バッファリング (Buffering)	93
6.2.3	メッセージの形式	93
6.2.4	同期方式 (Synchronization)	93
6.2.5	UNIX のメッセージ通信システムコール	93
6.2.6	UNIX のメッセージ通信プログラム例	95
6.2.7	UNIX のメッセージ通信プログラムの実行例	95
6.3	TacOS のメッセージ通信機構	98
6.3.1	リンク構造体	98
6.3.2	リンクの作成	98
6.3.3	サーバ用のメッセージ通信ルーチン	99
6.3.4	サーバプロセスの例	100
6.3.5	クライアント用のメッセージ通信ルーチン	100
6.3.6	クライアントプロセスの例	101
6.4	まとめ	102
第 7 章 モニタ		105
7.1	概要	105
7.2	構成要素	105
7.2.1	資源 (データ, 変数)	106
7.2.2	手続き (操作, メソッド)	106
7.2.3	ガード	106
7.2.4	条件変数	106
7.2.5	初期化プログラム	106
7.3	相互排除問題の解	106
7.3.1	共有変数の記述	106
7.3.2	共有変数の利用	107
7.4	生産者と消費者問題の解	107
7.4.1	キューの記述	108
7.4.2	生産者と消費者スレッドの記述	110
7.5	セマフォによるモニタの実装	110
7.5.1	モニタ機能の Java による実装	110
7.5.2	モニタ機能の使用	113
7.6	Java のモニタ風機構による生産者と消費者問題の解	113
7.7	まとめ	115
第 IV 部 主記憶管理		117
第 8 章 主記憶 (メモリ)		119

8.1	ハードウェア構成	119
8.2	メモリ保護機構	120
8.2.1	上限・下限レジスタ	120
8.2.2	ロック／キー機構	121
8.3	プログラムの再配置	122
8.3.1	再配置可能オブジェクトファイル	122
8.3.2	リロケーションレジスタ	123
8.4	アドレス空間の仮想化	124
8.4.1	单一仮想記憶	124
8.4.2	多重仮想記憶	124
8.4.3	仮想アドレス空間の配置	124
第 9 章 メモリ割付け方式		127
9.1	固定区画方式	127
9.2	可変区画方式	128
9.3	可変区画方式の空き領域選択方式	129
9.4	空き領域の管理方式	130
第 10 章 メモリ割付けプログラムの例		133
10.1	データ構造の初期化	133
10.2	メモリの割り付け	134
10.3	メモリの解放	136
第 11 章 セグメンテーション		139
11.1	リロケーションレジスタ方式の問題点	139
11.2	セグメント	139
11.3	セグメント番号	140
11.4	セグメンテーション機構	141
11.4.1	セグメントテーブル	141
11.4.2	物理アドレスへの変換	142
11.4.3	セグメントテーブルエントリのキャッシング	142
11.5	セグメンテーション機構による仮想記憶	143
11.5.1	スワップイン	143
11.5.2	スワップアウト	143
11.6	セグメントの共用	144
11.7	セグメンテーションの利点・欠点	144
第 12 章 ページング		147
12.1	基本概念	147

12.1.1 ページとフレーム	148
12.1.2 マッピング関数	148
12.1.3 外部フラグメンテーション	148
12.1.4 内部フラグメンテーション	149
12.2 ページング機構	149
12.2.1 ページング機構の概要	149
12.2.2 ページテーブルエントリ	150
12.2.3 ページテーブル	150
12.2.4 TLB (Translation Look-aside Buffer)	151
12.2.5 Page Table Walk	152
12.2.6 TLB エントリのクリア	152
12.3 ページの共用	152
12.4 ページテーブルの編成方法	153
12.4.1 二段のページテーブル	154
12.4.2 多段ページテーブル	155
12.4.3 逆引きページテーブル	156
 参考文献	161
 付録 A TaC に関する資料	165
A.1 CPU の概要	165
A.1.1 データ形式	165
A.1.2 CPU レジスタと PSW	165
A.1.3 機械語命令	166
A.2 メモリマップと I/O マップ	168
A.2.1 メモリ空間	168
A.2.2 I/O 空間	168

第Ⅰ部

OS の機能を使用してみよう

第 II 部

OS とは

第1章

オペレーティングシステムとは

オペレーティングシステム（Operating System : OS）は、Windows, macOS, Linux, FreeBSD, Android, iOS 等である。皆さんは、これらを使用した経験を持っているだろう。そして、これらが次のようなソフトウェアから構成されていることを何となく感じているのではないだろうか。

1. カーネル（OS の本体）
2. ライブラリ（プログラムが使用するサブルーチン, DLL）
3. ユーザインターフェース（GUI, CLI）
4. ユーティリティソフトウェア（ファイル操作, 時計, シェル, システム管理 ...）
5. プログラム開発環境（エディタ, コンパイラ, アセンブラー, リンカ, インタプリタ）

広義では上に列挙した全て^{*1}がオペレーティングシステムの一部である。逆に狭義では「カーネル」だけをオペレーティングシステムと考える。この講義では狭義のオペレーティングシステムの仕組みを勉強する。

1.1 オペレーティングシステムの役割

オペレーティングシステムの重要な役割は次に述べる二つである。

1.1.1 拡張マシンとしてのオペレーティングシステム

OS はハードウェアの機能を抽象化した便利な拡張マシンを提供する。次に抽象化と拡張マシンの例を示す。

例 1 二次記憶装置の抽象化（ファイルシステム）

ハードディスク, USB メモリ, CD-ROM 等の二次記憶装置は、どれもデータを記録する機能を持ったハードウェアである。しかし、それらの制御方法や記録されるデータの構造は全く異なる。オペレーティングシステムは、二次記憶装置をファイルの集合（ファイルシステム）として抽象化してユーザプログラム（アプリケーションプログラム）に提供する。

例 2 コンピュータそのものの抽象化（プロセス）

^{*1} 上に挙げたソフトウェアの中で「プログラム開発環境」は、Linux や FreeBSD では OS に含まれているが、それ以外では別にインストールする必要があり OS の一部とは言い難くなっている。

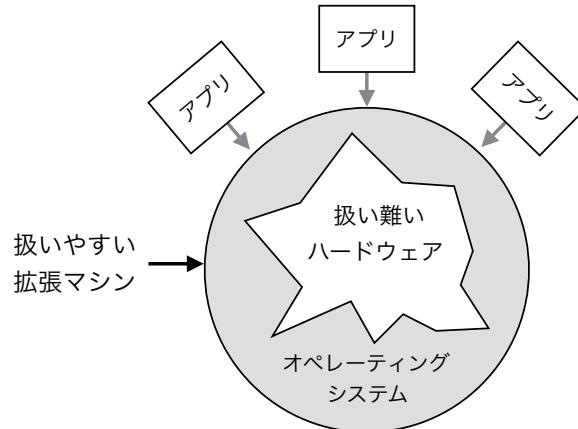


図 1.1 抽象化

プロセスはプロセス専用の仮想 CPU と仮想メモリを持つ。システムコールを通じて入出力も可能である。プロセスは CPU, メモリ, 入出力を持っているので 1 台のコンピュータと考えることもできる。

プロセスはコンピュータを抽象化したものだとも言える。（プロセス＝仮想コンピュータ）

例 3 拡張されたコンピュータ（システムコール）

オペレーティングシステムを備えたコンピュータ上では、アプリケーションプログラムがシステムコールを発行できる。システムコールを追加命令を考えると、オペレーティングシステムを備えたコンピュータは追加命令を実行可能な拡張マシンだと言える。（拡張マシンの命令＝機械語命令 + システムコール）

オペレーティングシステムが拡張マシンをアプリケーションプログラムに提供するイメージを図 1.1 に示す。ハードウェアの複雑で統一されていない凸凹のインターフェースは、オペレーティングシステムによってスッキリした円弧のインターフェース（使いやすい抽象化されたインターフェース）に変換される。

オペレーティングシステムの円がハードウェアの外側にあるのは、オペレーティングシステムによって機能が拡張されたことを示す。ハードウェアを含む円全体が拡張マシンを表している。

1.1.2 ハードウェア管理プログラムとしてのオペレーティングシステム

オペレーティングシステムはハードウェア資源を管理・制御し、アプリケーションプログラムにシステムコール等のサービスを提供する。図 1.2 はカーネルの役割を説明している。

オペレーティングシステムは、管理するハードウェア資源をアプリケーションプログラムに割当てる。複数のアプリケーションプログラムに割り付けるために資源を仮想化して必要な数だけ作り出す。例えば、CPU は時間を区切って複数のプロセスが共有する（時分割多重による仮想化）。メモリはアドレスで区切って複数のプロセスが共有する（空間分割多重による仮想化）。



図 1.2 コンピュータシステムの構成

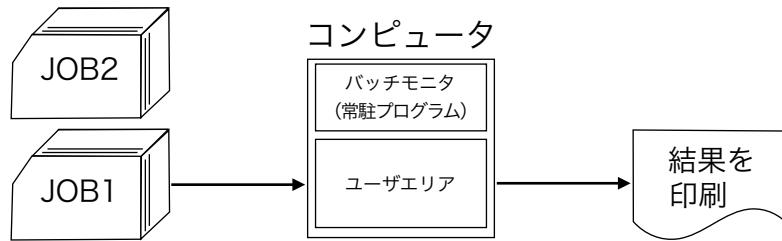


図 1.3 バッチ処理

1.2 オペレーティングシステムの歴史

1.2.1 第1世代（1945～1955, 真空管の時代）

初期のコンピュータはコンソールパネルを通して操作する、巨大な TeC のようなものだった。OS は存在せずプログラマはまさに TeC と同様なプログラミングとデバッグを行っていた。

しかし、当時のコンピュータは TeC と異なり大変高価な装置であった。その高価なコンピュータを一人のプログラマが長時間にわたって独占使用することになる。プログラマがバグの原因を考えている間、とても高価なコンピュータが遊んでしまい勿体ないものであった。

1.2.2 第2世代（1955～1965, ツランジスタの時代）

コンピュータがトランジスタ回路で製作されるようになり、メインフレームと呼ばれる大型コンピュータが、大企業、政府機関や大学等で実用的に使用されるようになった。メインフレームは数百万ドルと高価だったので、ハードウェアを遊ばせること無く使用することが優先課題であった。そこで人手を介すこと無く自動的に次々と連続して処理を行う「コンピュータの自動運転」が行われるようになった。この処理方式のことはバッチ処理と呼ばれた。図 1.3 にバッチ処理の概要を示す。

プログラマは図 1.4 のような紙カードにプログラムやデータを一行ずつ打込む。100 行のプログラムは 100 枚の紙カードを使用して記録する。このようにして出来た紙カードの束が一つの処理単位（ジョブ）になる。コンピュータではバッチモニタと呼ばれる常駐プログラムが実行される。バッチモニタは紙カードからジョブを読み込み実行させる。ジョブが終了するとバッチモニタに制御が戻り次のジョブが自動的に実行される。バッチモニタが発展してやがて OS になる。

この方式でうまく処理できるように、次のような発明があった。



図 1.4 紙カード

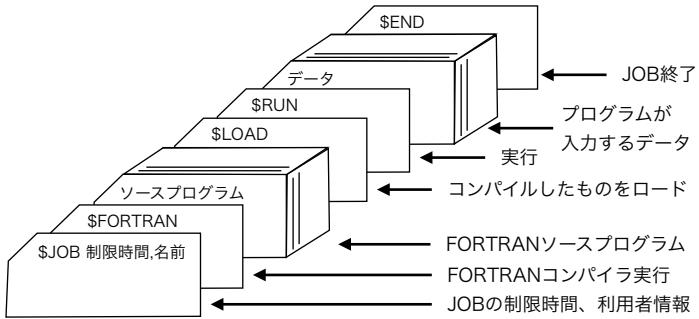


図 1.5 ジョブの構成

1. JOB 制御言語 (JCL : Job Control Language)

バッチモニタを制御するコマンド言語を JOB 制御言語 (JCL) と呼ぶ。JCL コマンドはジョブ途中の紙カードに記載する。図 1.5 に JCL を含むジョブの構成を示す。これは、FORTRAN 言語で記述したプログラムを実行し、後半にあるデータを処理するジョブの例になっている。

2. 実行モード

ユーザプログラム（ジョブ）のバグでバッチモニタが破壊されないようにするために、ユーザプログラム実行中なのかバッチモニタ実行中なのか区別する必要がある。どちらを実行中なのかを示すハードウェアのフラグを導入し、ユーザモードとカールモード（スーパバイザモードとも呼ぶ）を区別するようになった。ユーザモードではハードウェアへのアクセスや、実行できる機械語命令に制限がある。

3. システムコール

ユーザプログラムが直接に入出力装置等にアクセスすることは、バッチ処理を継続できなくなる恐れがあるので許されない。例えばユーザプログラムがハードウェアのモードを切り換えてしまうと、以降のジョブが正常に実行されなくなる恐れがある。そこで、ユーザプログラムはバッチモニタに依頼（システムコール）して入出力をを行う必要がある。



Wikimedia / Bundesarchiv, B 145 Bild-F038812-0014 / Schaack, Lothar / CC-BY-SA 3.0 de

図 1.6 フォルクスワーゲンで使われている System/360

プログラムが終了する時はカーネルモードに切り換えてバッチモニタに戻る必要がある。カーネルモードに切り替える機械語命令をユーザプログラムが実行可能だと、実行モードが無意味になるので許可すべきではない。システムコールを使用してプログラムを終了する。

4. 記憶保護

ユーザプログラムのバグでバッチモニタが破壊されないように、ユーザモードで実行中は主記憶のバッチモニタ領域に書き込みができないようにする。

1.2.3 第3世代（1966～1980, ICとマルチプログラミングの時代）

1960年代のコンピュータは IC (Integrated Circuit) を用いて作られるようになり価格性能比が随分改善された。第3世代と呼ばれる当時のオペレーティングシステムの中には、現代のオペレーティングシステムの先祖であったり、強い影響を与えたものがある。図 1.13 に第3世代から現代に至るまでの系統図を示す。

IBM が開発した System/360 (図 1.6) は高価な大型のものから、安価な小型のものまで同じオペレーティングシステムが使用でき、同じユーザプログラムを実行できるシリーズ化を行い商業的に大成功をおさめた [27]。System/360 はそれ以前のものとは異なり科学技術計算にも事務処理にも使用できる。System/360 のオペレーティングシステムは、1966年にデビューした OS/360 である。図 1.13 に示すように、OS/360 の子孫である z/OS が現代でも使用されている [1]。

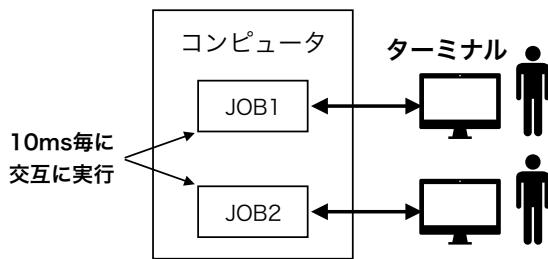
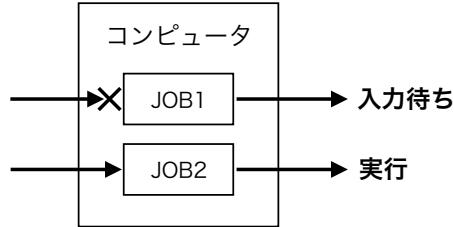
OS/360 を含む第3世代のオペレーティングシステムが実現した重要な新しい機能を紹介する。

- 仮想記憶

主記憶を仮想化し実際より大きい主記憶があるように見せる。実際の主記憶より大きいプログラムが実行可能になる。

- マルチプログラミング

図 1.7 のように、複数のプログラム（ジョブ）を主記憶にロードしておき、その中に実行可能な



ものを選んで実行する。入出力待ち等で実行できなくなったら他のプログラムを実行する。高価な CPU が入出力待ちで停止する可能性を低くすることができた。

- タイムシェアリング (TSS : Time Sharing System)

マルチプログラミングの一種である。図 1.8 のように、複数のターミナルをコンピュータに接続し複数のユーザが同時にコンピュータを使用できるようにする。短時間（例えば 10ms）で処理するジョブを次々に切り換えることで、ユーザは自分がコンピュータを独占しているように感じることができる。なお、ターミナルは図 1.9 のような、キーボードと表示装置だけを備えた安価な装置である。

この時代のオペレーティングシステムやコンピュータシステム、そして、それらの開発プロジェクトの中で、その後のオペレーティングシステムに多くの影響を与えた有名なものを紹介する。

- OS/360

世界初の本格的な商用オペレーティングシステムである。メインフレームの主流 OS となり子孫は現在でも使用されている [1]。

- MULTICS (MULTiplexed Information and Computing Service) プロジェクト [27]

MIT、ベル研究所、General Electric が共同で始めた巨大で強力なコンピュータシステムを構築するプロジェクトである。強力な一台のコンピュータで都市一つ分のコンピュータサービスを提供する構想だった。完成までに長い期間を要し（その間にベルと GE が脱落し）、商業的には失敗であったがその後のオペレーティングシステムに影響を与える多くのアイデアが出てきた。

- UNIX (ユニックス)

MULTICS プロジェクトから抜けたベル研の Ken Thompson らにより開発された [5]。図 1.13 に示すように、現代のオペレーティングシステムの多くが UNIX を起源にしている。子孫ではないものも UNIX の影響を強く受けている。Linux は UNIX 互換のオペレーティングシステムを作ろうとして開発が始まった [19]。Android の中身は Linux である [20]。z/OS は UNIX 互換環境を備えている [4]。Windows にも UNIX 互換環境 (POSIX サブシステム) を利用可能なものがある [22]。

- DynaBook (ダイナブック : OS だけでなくコンピュータ全体を指す) [31]

アラン・ケイが 1972 年に著した「A Personal Computer for Children of All Ages」[29, 30] に登場する理想のパーソナルコンピュータである。アラン・ケイがゼロックスのパロアルト研究所に在籍中の 1970 年代に開発した Alto 上の「暫定ダイナブック環境」(図 1.10) は既に GUI やマウスを使用していた。スティーブ・ジョブスが Alto を見たことが LISA 開発きっかけになったと言われている [16]。

1.2.4 第4世代（1980～現代、PC の時代）

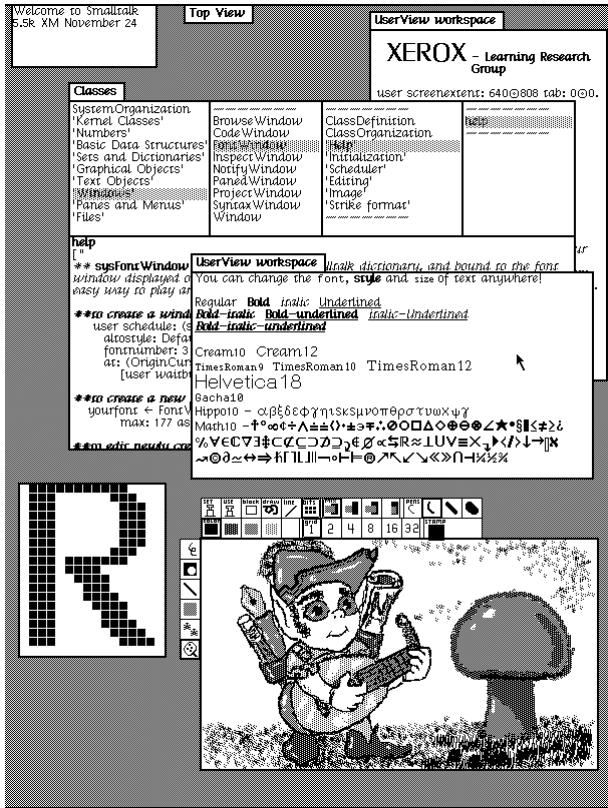
1970 年代に単一の LSI に CPU 全体を集積したマイクロプロセッサが登場した。1970 年代中旬にはマイクロプロセッサを用いて個人向けのコンピュータであるパーソナルコンピュータ（当時はマイクロコンピュータと呼んでいた）を作ることが可能になった。それに伴いパーソナルコンピュータ用のオペレーティングシステムが登場した。

1. 8bit マイクロコンピュータの時代

1977 年に Digital Research 社が CP/M (Control Program for Microcomputer) と呼ばれる 8bit マイクロコンピュータ用の簡単なオペレーティングシステムを開発し成功した。しかしこのオペレーティングは 16bit パーソナルコンピュータの時代には早々に消え去ってしまった [28]。

2. 16bit パーソナルコンピュータの時代

IBM が 1981 年に 16bit パーソナルコンピュータ IBM PC[23] (図 1.11) を発売した。IBM PC は現在の Windows PC の先祖である。IBM PC の子孫は改良や拡張を続けながら現在まで高いシェアを維持し続けている。IBM PC のオペレーティングシステムとして開発されたのが、Microsoft 社の MS-DOS (MicroSoft Disk Operating System) [21] である。バージョン 2 からは UNIX の



ウィキメディア / SUMIM.ST /

Alto や NoteTaker で動作したアラン・ケイ達の暫定 Dynabook 環境 (Smalltalk-76、同-78 の頃) / CC-BY-SA

4.0

図 1.10 Alto (Alto エミュレータ) のスクリーンショット

のような階層ディレクトリやパイプ、リダイレクト等の機能を持っている。図 1.13 に示すように、MS-DOS は Windows に置き換わり Windows ME までバージョンアップが繰り返された。

Apple 社は 1984 年に Macintosh (図 1.12) を発売した。Machintosh の OS である MacOS は LISA を経て DynaBook[29, 30] の影響を受けていると言われている [28]。図 1.13 に示すように、当初の MacOS は MacOS 9[15] まで改良が続けられた。

3. 32bit パーソナルコンピュータの時代

1990 年頃には 32bit のマイクロプロセッサがパーソナルコンピュータにも使用されるようになった。32bit のマイクロプロセッサは実行モードを備え、またメモリ管理ユニットも利用可能であった。つまり、カーネルモードとユーザモードを使い分けたり仮想記憶を利用する本格的な第 3 世代のオペレーティングシステムを実行できる環境がパーソナルコンピュータにも整った。

そこで、従来ワークステーションやミニコンで使用されていた UNIX を安価なパーソナルコンピュータ（特に IBM PC 互換機）で動くようにする人たちが現れ、オープンソースソフトウェアとして Linux や FreeBSD 等の開発が始まった。また、もともとパーソナルコンピュータ用の Windows や Mac OS も 32bit マイクロプロセッサの機能を使いこなす本格的なオペレーティング



Wikimedia / Bundesarchiv, B 145 Bild-F077948-0006 / Engelbert Reineke / CC-BY-SA 3.0 de

図 1.11 IBM PC



Wikimedia / w:User:Grm wnr / File:Macintosh 128k transparency.png /GFDL

図 1.12 初代 Macintosh

システムに生まれ変わった。

- Linux

1991 年に開発が始まった Linux は UNIX 互換のオペレーティングシステムをパーソナルコンピュータ（IBM PC 互換機）用に独自に作成したものである [19]。Linux は改良され続け、現在ではパーソナルコンピュータだけでなく、スーパーコンピュータ「京」のオペレーティングシステム [32] から、スマートフォンのオペレーティングシステムである Android[20]、テレビ等の組込みシステムのオペレーティングシステムまで、広く使われるようになっている。

- BSD 系の UNIX

386BSD[11] は BSD UNIX を Intel 80386 CPU を搭載したパーソナルコンピュータ（IBM PC 互換機）で動作するようにしたものである。386BSD は FreeBSD 等に受継がれるが UNIX のライセンス問題が発生する [5]。ライセンス問題が片付き安心して使用できるようになった

4.4BSD-Lite Release 2[5] をベースに FreeBSD, NetBSD, OpenBSD 等の多くの BSD 系 PC-UNIX が開発された。

その後、FreeBSD は MacOS X に取り込まれている。また、FreeBSD に ZFS が移植された [33] のでファイルサーバ用に特化した FreeNAS[13] にも使用されている。なお、徳山工業高等専門学校・情報電子工学科のパソコン室では 1993 年 10 月に 386BSD の利用を開始して以来、2014 年 3 月まで FreeBSD を学生用 PC やサーバのオペレーティングシステムとして使用してきた [25]。

- System V 系の UNIX

System V の流れを汲む Solaris[6] は、RISC マイクロプロセッサ SPARC を搭載するサーバやワークステーションでも、パーソナルコンピュータ（IBM PC 互換）でも使用できる。

- 従来のパーソナルコンピュータ用オペレーティングシステム

従来の Windows や Mac OS は CPU の実行モード等を使用していなかったので、アプリケーションプログラムのバグによりシステム全体が停止するようなトラブルを防ぐことができなかった。そこで、32bit マイクロプロセッサの使用を前提に新しく作り直された。

新しく作り直された 32bit の Windows NT 系列の製品は、徐々に従来の Windows を置換えた。（図 1.13 参照）。現在（2017 年 10 月）の最新版は Windows 10 である。

MacOS は、2001 年に UNIX の流れを汲み安定して動作する OPENSTEP ベースの MacOS X[17] に置き換わった（図 1.13 参照）。その後、名称が OS X, macOS と変更されたがこれらは MacOS X の改良版である。現在（2017 年 10 月）の最新版は macOS 10.13 High Sierra である。iPhone の iOS は MacOS X をタッチパネル用に再構成したものである [18]。

1.2.5 インターネット世代

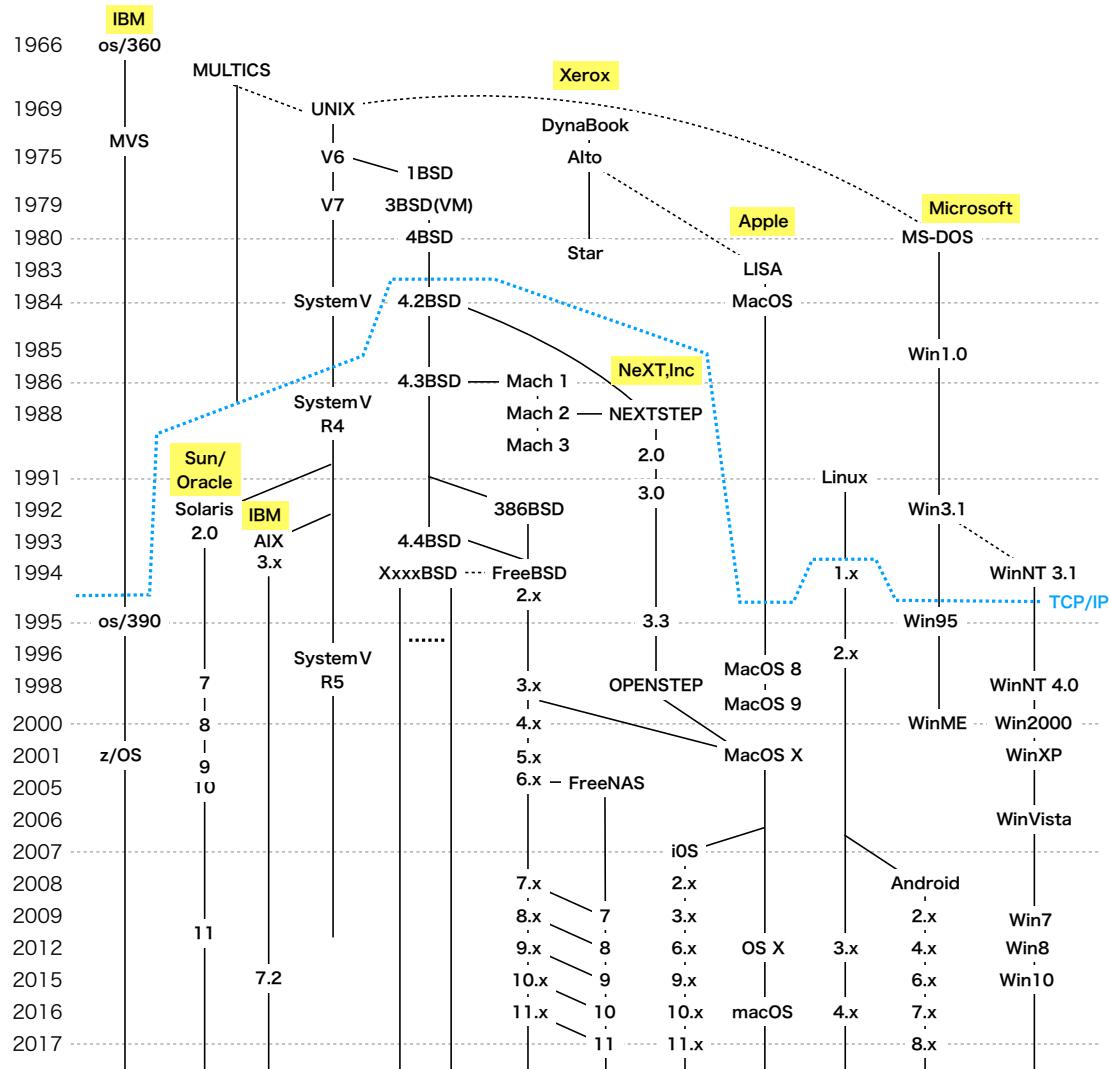
現在のオペレーティングシステムは TCP/IP 機構が組込まれインターネットに接続することができる。今ではパーソナルコンピュータやスマートフォンの使用をインターネット抜きに考えることができない。オペレーティングシステムにとってインターネットに接続できることは重要なことである。

TCP/IP を実装した 4.2BSD が 1984 年に公開された [10]。以来、4.2BSD の子孫はインターネットに対応している。1988 年に公開された System V R4 は BSD 起原の TCP/IP の実装を含んでいた [24]。この子孫もインターネットに対応している。Linux も 1.0 の頃には TCP/IP の実装を含んでいた [26]。Windows は Windows 95 から TCP/IP を標準装備している [22]。MacOS は MacOS 8 が発表されるまでにはインターネット対応がされていた [15]。メインフレームの世界でも OS/390 はインターネットに対応した [3]。

このようにして 1990 年代の後半には多くのオペレーティングシステムがインターネット対応を完了させた。インターネット対応を完了させたオペレーティングシステムを「インターネット世代のオペレーティングシステム」と言うことができる。

1.3 まとめ

狭義のオペレーティングシステムはカーネルのことを指す。本書は狭義のオペレーティングシステムについて述べている。



系統図は [1, 2, 3, 4, 5, 6, 7, 8, 10, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22] の内容を総合して作成した。

図 1.13 オペレーティングシステムの系統図

オペレーティングシステムの重要な役割りは、コンピュータの資源を抽象化することと仮想化することである。オペレーティングのユーザは、使いやすい抽象化されたインターフェースを通して資源を利用できる。また、ユーザは仮想化された資源を必要なだけ独占して使用することができる。

オペレーティングシステムは、1950 年代に出現したバッチモニタから進化してきた。現在では、スーパーコンピュータから組み込み用コンピュータまで、非常に広い範囲のコンピュータが本格的なオペレーティングシステムを搭載している。

練習問題

- 1.1 抽象化について説明しなさい。
- 1.2 抽象化の例をいくつか挙げなさい。

- 1.3 仮想化について説明しなさい.
- 1.4 仮想化の例をいくつか挙げなさい.
- 1.5 自分がいつも使用しているコンピュータやスマートフォンのオペレーティングシステムの種類を調べなさい.

第2章

前提知識

以下では、本書で想定しているコンピュータのハードウェアやソフトウェアの構成について解説する。

2.1 コンピュータのハードウェア構成

本書は、コンピュータのハードウェア構成が図 2.1 のようになっていることを前提にしている。複数の CPU (Central Processing Unit) がメモリを共有し、また、全ての CPU は同じ機能を持ち優劣がない。このような方式を **SMP** (対称型マルチプロセッシング : Symmetric Multiprocessing) と呼ぶ。メモリは CPU だけでなく、I/O コントローラ (図 2.1 ではアダプタやコントローラ) にも共有される。

1. CPU

CPU はコンピュータの頭脳である。図は CPU が二つの構成になっているが、実際は一つの場合も、もっと多い場合もある。

2. メモリ（主記憶装置）

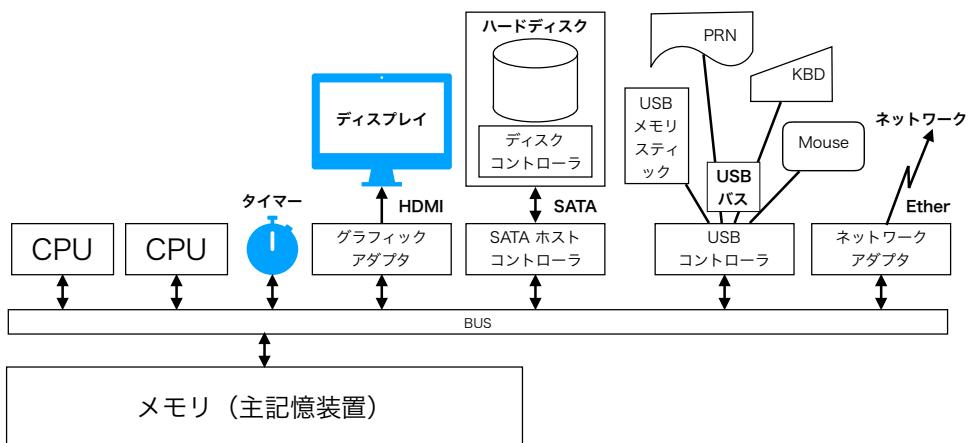


図 2.1 ハードウェア構成

プログラムやデータを記憶し、プログラム実行する際に CPU が直接使用する記憶装置である。

3. タイマー

一定間隔で繰り返し CPU に割り込みを発生するインターバルタイマーである。

4. グラフィックアダプタ

ディスプレイを接続するためのアダプタである。表示内容を記憶するメモリを独自に持つ場合と、主記憶装置を使用する場合がある。最近のパーソナルコンピュータでは、グラフィックアダプタに GPU(Graphics Processing Unit) が組込まれている。

5. SATA ホストコントローラ

SATA (Serial Advanced Technology Attachment) は、パーソナルコンピュータと二次記憶装置（ハードディスクや CD-ROM）を接続するためのインターフェース規格である。SATA ホストコントローラは次のような動作をする。

- CPU が SATA ホストコントローラにコマンドを書き込む。コマンドは、「読み／書き」、「セクタアドレス」、「セクタ数」、「メモリアドレス」を含んだものである。
- SATA ホストコントローラは、ディスクコントローラと通信しハードディスクにコマンドを渡す。
- ハードディスクの読み・書きが可能になったら、ホストコントローラはハードディスクとメモリの間でデータ転送を行う。このような CPU を介さないデータ転送のことを、DMA (Direct Memory Access) と呼ぶ。
- SATA ホストコントローラは CPU に割り込み信号を送り、データの転送が完了したことを知らせる。（I/O 完了割り込み）

CPU は、SATA ホストコントローラにコマンドを送ってから割り込みが発生するまでの間、他の仕事をすることができる。ハードディスクの操作（I/O 操作）と CPU の計算は並列実行される。

6. USB コントローラ

USB (Universal Serial Bus) は、パーソナルコンピュータと周辺装置を手軽に接続できるインターフェースである。USB メモリスティックやプリンタ、キーボード、マウス等、多くの周辺装置が USB を通して接続できる。USB コントローラも SATA ホストコントローラのように DMA 機能を備えている。

7. ネットワークアダプタ

パーソナルコンピュータのネットワークアダプタは、GbE (Gigabit Ethernet) 規格のものが普及している。これも SATA ホストコントローラのように DMA 機能を備えている。

8. BUS (バス)

パーソナルコンピュータのハードウェアを構成する装置の間でデータをやり取りするための配線である。CPU だけでなく DMA を使用するコントローラやアダプタが大量のデータ転送を行うので、バスのデータ転送能力がパーソナルコンピュータの性能向上のボトルネックになる。

そのため後で説明するように、実際の物理的な接続は図 2.1 とはかなり異なった構成になっている。しかし、オペレーティングシステムが意識しなければならない論理的な接続は図 2.1 のようなものである。

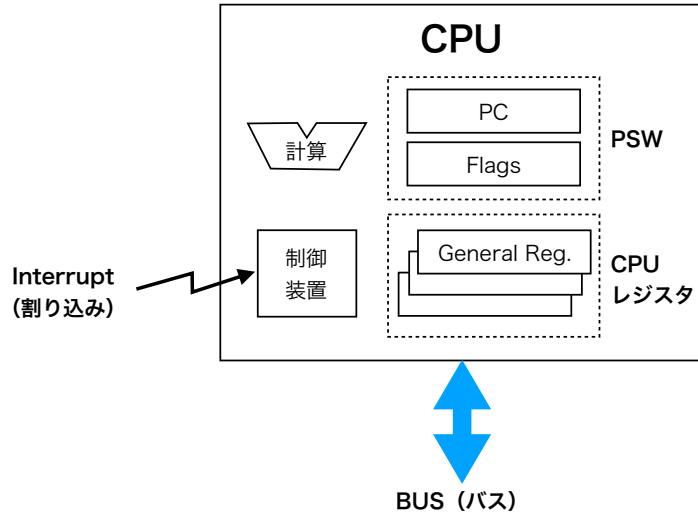


図 2.2 CPU の構成

2.2 CPU の構成

本書では、CPU は図 2.2 のような部品で構成されると考える。図 2.1 に示したように、CPU は BUS を通して他の装置と接続される。CPU は、一つの機械語命令の実行が終わり次の命令の実行を開始する前に、他の装置から割り込みを受け付けることができる^{*1}。

1. PSW (Program Status Word)

PSW は、PC (Program Counter) と Flags (フラグ) から構成されるものとする^{*2}。PC は CPU が実行中のプログラムの命令アドレスを保持するカウンタである。Flags は計算の結果によって変化するフラグの他に、割り込み許可／不許可を表現するビット、実行モード（ユーザモード／カーネルモード）を表現するビット等が含まれる。

2. CPU レジスタ

計算に使用する CPU の汎用レジスタのことである。TeC では G0, G1, G2, SP のこと、情報処理技術者試験の COMET では GR0, GR1, GR2, GR3, GR4 のことである。

PSW と CPU レジスタは、機械語命令を実行する毎に値が変化・確定しプログラムが意識している^{*3}ので、CPU を仮想化し実行するプロセスを切換える際に保存・復旧の対象となる。

2.3 最近のコンピュータの実際の構成

Intel 社の CPU を使用したデスクトップ・パーソナルコンピュータとサーバコンピュータの構成を説明する。バスがボトルネックにならないように、CPU にメモリを直接接続してある。

^{*1} 例外的に、メモリ管理に関する一部の割込は機械語命令の途中で発生する。

^{*2} 教科書によつては、フラグだけを PSW と呼ぶ場合もある。

^{*3} 一方で CPU 内部にはプログラムから見えないレジスタもある。

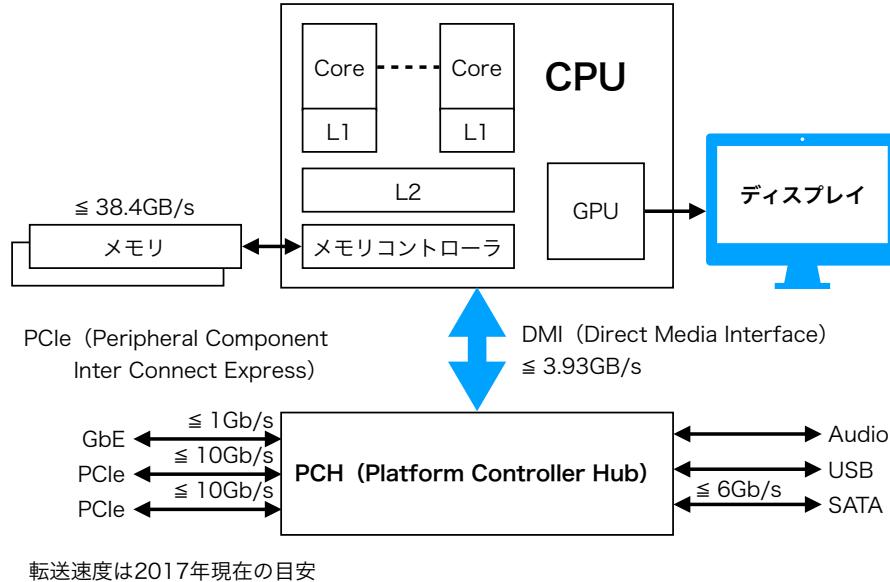


図 2.3 デスクトップ PC の構成

2.3.1 デスクトップ・パーソナルコンピュータ

図 2.3 は Intel 社の CPU を使用した近年のデスクトップ・パーソナルコンピュータの構成を表している。Intel 社の用語では、これまで「CPU」と読んでいたものが「Core (コア)」と呼ばれる。「CPU」は複数のコアを含んだ LSI のことを指している。デスクトップ用の CPU には 1 ~ 4 個のコアが集積されている。

コアに隣接している L1 はレベル 1 キャッシュ (Level 1 cache) を表している。L2 は複数のコアにシェアされるレベル 2 キャッシュ (Level 2 cache) を表している。メモリとのデータ転送量が多い Core と GPU が CPU に集積され、I/O 装置のコントローラやアダプタは PCH に集積されている。CPU と PCH は DMI と呼ばれる専用のインターフェースを用いて接続される。

2.3.2 サーバコンピュータ

より強力な処理能力が必要なサーバ用コンピュータでは、図 2.4 のように多くのコアを内蔵する CPU を複数個使用する。現在（2017 年秋）最新の Intel Xeon Processor Scalable Family の場合、CPU 同士は UPI と呼ばれる高速な専用インターフェースで接続される。最大の構成は、28 コアの CPU を 8 個使用し合計 224 コアのものである。PCH もサーバ用のものでは、より多くのストレージやネットワークを接続できる。

2.4 オペレーティングシステムの構造

図 2.5 にオペレーティングシステムの構造を示す。オペレーティングシステムのカーネルは図 2.5 中央部分のソフトウェアである。ユーザプロセスはユーザモードで、カーネルはカーネルモードで実行される。

2.4.1 カーネルの構成

図 2.5 に示すように、カーネルは以下のようなモジュールから構成される。

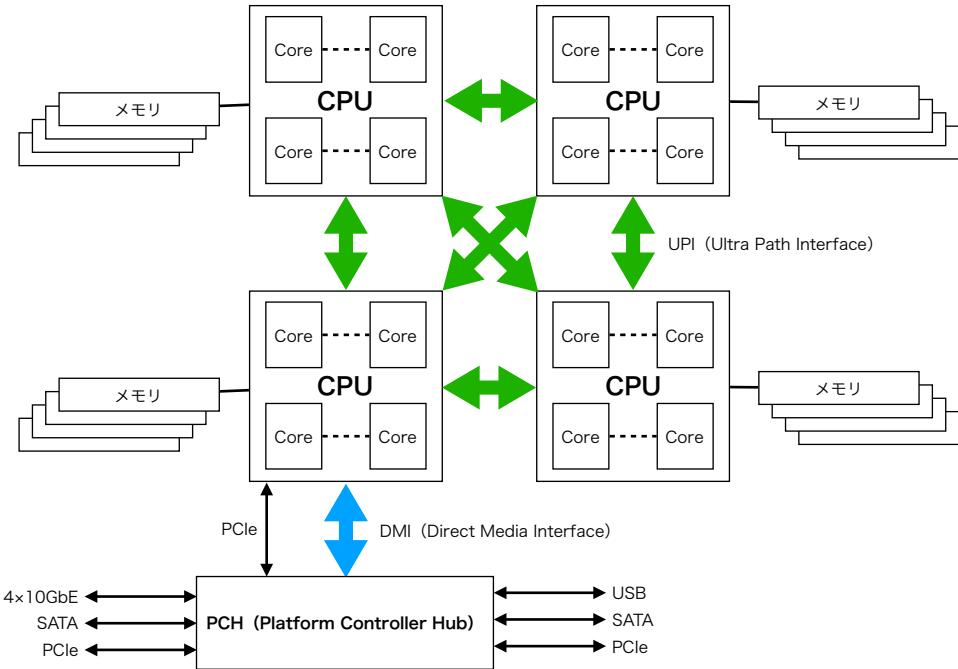


図 2.4 サーバ PC の構成

1. 割り込みハンドラ

割込みが発生した時に自動的に実行される割込み処理ルーチンである。割込みが発生した原因を判断し、必要なモジュールを呼出す。例えば、タイマーからの割込みならタイマーのデバイスドライバを呼出す。

2. ディスパッチャ

カーネルの処理が終了した時、実行可能なプロセスの中から一つを選んで実行を再開させる。

3. コア

割込みハンドラとディスパッチャを含むコアは、資源の仮想化を行うために必ずカーネルモードで実行される必要がある部分である。

4. サービスマジュール

サービスモジュールは、ハードウェアを抽象化した便利なコンピュータをユーザ・プロセスに提供するためのプログラムである。

2.4.2 カーネルの動作概要

通常、コンピュータはユーザ・プロセスを実行し目的の仕事をしている。何かイベントが発生すると割込みにより CPU に通知される。CPU はカーネルモードに切り替わり割込みハンドラに制御を移す。CPU がユーザ・プロセスの実行からカーネルの実行に移行するのは、割込みが発生した時だけである。割込み原因

カーネルへ実行を移すには割込みを発生する以外に方法がない。割込みが発生する原因には以下のようなものがある。システムコール以外はユーザ・プロセスが意図しない間に発生する。

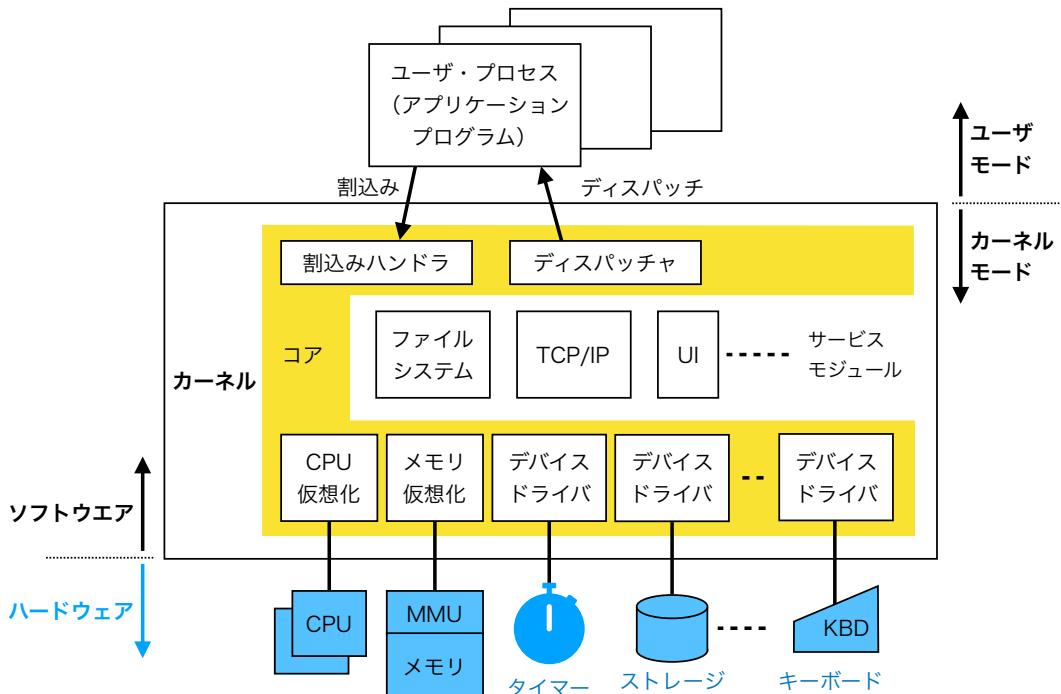


図 2.5 オペレーティングシステムの構造

1. I/O 完了・タイマー

ホストコントローラやネットワークアダプタ、タイマーのようなハードウェアが、コマンドの実行完了等を CPU に知らせるために発生する。

2. システムコール

ユーザ・プロセスは、割込みを発生する特殊な機械語命令である **SVC (Supervisor Call)** 命令^{*4} を用いてシステムコールを発行する。カーネルは SVC 命令実行時の CPU レジスタの値などからシステムコールの種類やパラメータを知ることができる。

3. 保護違反

ユーザ・プロセスが、ユーザ・モードでは実行が許可されない命令を実行したり、アクセスが許可されないメモリ領域をアクセスした場合に発生する。

4. ソフトウェアのエラー

ユーザ・プロセス実行中に計算でオーバーフローが発生したような時に発生する。

5. ハードウェアのエラー

ハードウェアの故障や電源の異常を検知した時に発生する。

割込み発生時のカーネルの動作

割込みが発生するとカーネル・モードに切り換わり割込みハンドラに制御が移る。その後、カーネル内では以下の手順で処理がされる。

^{*4} CPU によっては TRAP 命令、INT 命令と呼ばれることがある。

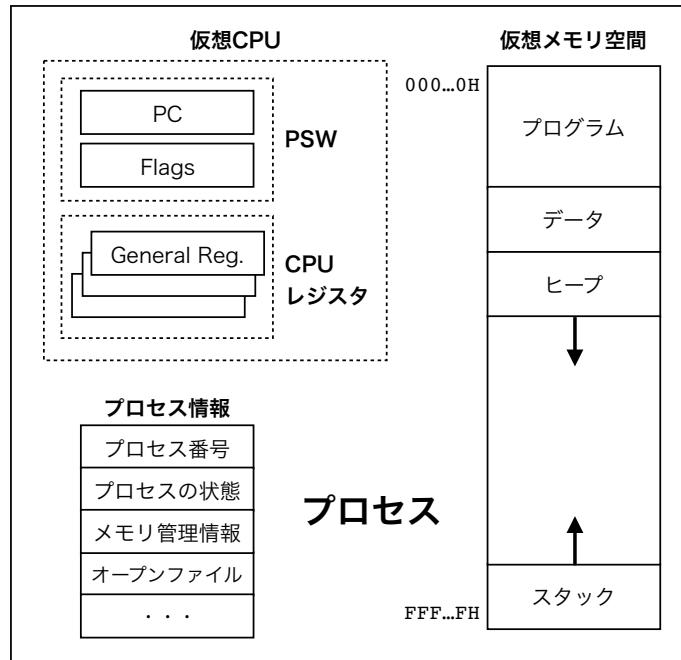


図 2.6 プロセスの構造

- 割込みハンドラは後でプロセスの実行を再開できるように、プロセスの CPU の状態（コンテキスト : PSW, CPU レジスタ）を保存する。
- 割込みハンドラは割込み原因を調べ、原因に応じたカーネル内のサービスモジュールやデバイスドライバに制御を渡す。例えばファイル操作のシステムコールならファイルシステムへ制御を渡す。
- サービスモジュールやデバイスドライバの処理が終了したらディスパッチャに制御が渡される。ディスパッチャは実行可能なプロセスの一つを選び、コンテキストを復旧しプロセスの実行を再開させる。

2.4.3 プロセスの構造

図 2.5 のユーザ・プロセス部分を詳しく描いたものを図 2.6 に示す。プロセスを構成する各部を以下で説明する。

1. 仮想 CPU

CPU を仮想化し、プロセス毎に CPU が存在するように見せることで、マルチプログラミングを可能にする。プロセスが CPU を使用する時間を区切り、次々に切替える時分割多重により CPU の仮想化は達成される。

他のプロセスが CPU を使用している間に、プロセスのコンテキストを保存する領域を仮想 CPU と呼ぶこととする。ハードウェアの実 CPU に対応して PSW と CPU レジスタの保存先が必要である。前の節で説明したように、プロセスからカーネルに制御が移る時にプロセスのコンテキストを保存する。プロセス実行時にはコンテキストが実 CPU にロードされる。

2. 仮想メモリ空間

メモリを仮想化しプロセス毎に専用のメモリ空間が存在するように見せかける。実現方法は第7章の「メモリ管理」で詳しく学ぶ。仮想メモリ空間は次の部分から構成される。

(a) プログラム

機械語プログラムがここに配置される。C言語で記述されたプログラムの場合、関数の実行文(式文、if文、for文、while文など)が翻訳された機械語が該当する。

(b) データ

プログラムの変数部分がここに配置される。C言語ではグローバル変数が該当する。

(c) ヒープ

プログラム実行時に動的に拡大される領域である。C言語の `malloc()` 関数はヒープに新しい領域を確保する。`malloc()` 関数が使用される度にヒープ領域は後ろに向かって拡大していく。

(d) スタック

プログラム実行時にメモリ空間の最後から前に向かって伸びて行く領域である。サブルーチン・コール時に戻りアドレスを保存したり、C言語のローカル変数や関数引数を置いたりするために使用される。

3. プロセス情報

名前にあたる「プロセス番号」、実行中／実行可能／待ちのどの状態なのか表す「プロセスの状態」、使用しているメモリの大きさ等を表す「メモリ管理情報」、CPUを使用した時間を表す「CPU時間」等の情報のことである^{*5}。その他に、プロセスが現在オープンしているファイルに関する情報や、親プロセス、子プロセス、シグナルハンドラの登録状況、プロセスの優先度など、様々な情報がここに記録される。

2.5 カーネルの構成方式

カーネルが動作不良を起こすと実行中の全てのユーザ・プロセスを巻き込んでシステムが停止するので、カーネルには非常に高い信頼性が要求される。しかし、カーネルは非常に大きなプログラムになりがちであり^{*6}、高い信頼性を確保するにはカーネルの構成方法に工夫が必要である。

2.5.1 単層カーネル（モノリシック・カーネル）

最も一般的な構成方法である。図2.5のカーネルは単層カーネルの例になっている。カーネル内の全てのモジュールがリンクされ、一つのプログラムになる。カーネル内でモジュールの呼出しはCALL機械語命令を用いて行うので効率が良い。しかし、モジュール同士が密にリンクされているので、モジュール間で情報の隠蔽がし難くバグが入りやすい。また、全てのモジュールがカーネル・モードで実行されるので、一つのモジュールのバグが致命的な結果を引き起こす。LinuxやFreeBSDは、この方式のカーネルを持つ。

^{*5} これらはUNIXのpsコマンドで表示することができる。

^{*6} LinuxやWindowsのカーネルのソースコードは500万行にもなる[34]。

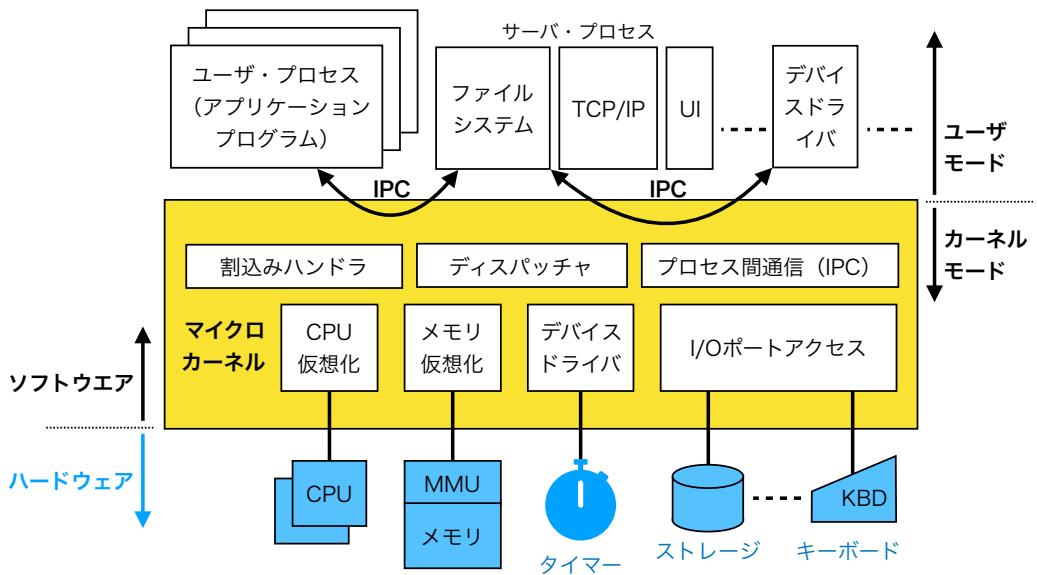


図 2.7 マイクロカーネル方式

2.5.2 マイクロカーネル (micro-kernel)

図 2.5 の「コア」からデバイスドライバを取り除き^{*7}、カーネル（マイクロカーネル）とし構成する方式である。図 2.7 にマイクロカーネル方式の概要を示す。カーネル・モードで実行されるのはマイクロカーネルだけである。

サービスモジュールはカーネルから独立したサーバ・プロセスとし、権限の低いユーザ・モードで実行される。ユーザ・プロセスは、マイクロカーネルが提供する IPC (プロセス間通信 : Inter-Process Communication) を用いて、サーバ・プロセスにサービスを要求する。サーバ・プロセス同士、サーバ・プロセスとデバイスドライバ・プロセスも IPC を用いて通信する。

デバイスドライバは I/O ポートにアクセスするのでカーネル・モードで実行される必要があると考えられるが、I/O ポートへのアクセスをマイクロカーネルのシステムコールに置換えることで、デバイスドライバもユーザ・プロセスとして実装することが可能である。この場合は、デバイスドライバがアクセスしても良い I/O アドレスの範囲内かどうか、マイクロカーネルがチェックすることが可能である。

マイクロカーネル方式は、サービスモジュールやデバイスドライバが権限の低いプロセスとして実行されるので、これらのバグでシステム全体が停止する危険性が低い。また、サービスモジュールやデバイスドライバ毎に独立したプログラムになりモジュール化が徹底しやすいので、巨大な単一プログラムであるモノリシックカーネルと比較してバグが発生しにくい。信頼性の高いオペレーティングシステムを構成するために有利である。しかし、IPC とプロセス切り替えのオーバヘッドが大きいため性能が低くなる。多くの場合、信頼性と性能はトレードオフの関係にある。

^{*7} タイマーのデバイスドライバは CPU の仮想化に必要なので、マイクロカーネルに残す。

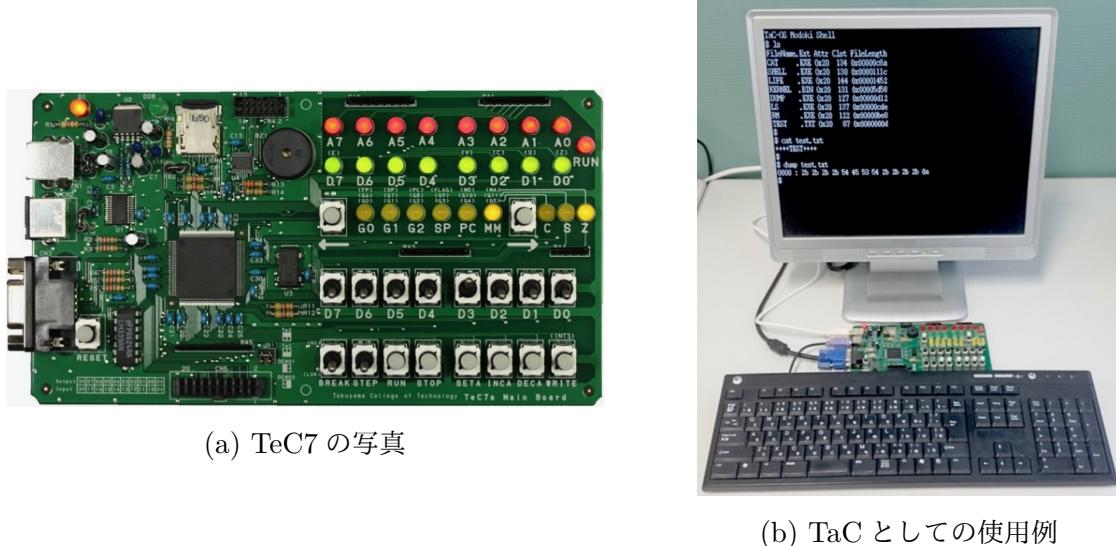


図 2.8 TeC7 と TaC

2.6 TaC

TaC (Tokuyama Advaced educational Computer) は、TeC7 (Tokuyama Educational Computer Ver.7)*8に内蔵された 16bit のコンピュータである。TeC7 基板上のジャンパ設定により TaC モードに切り換える。図 2.8 に写真を示す。TaC は、ディスプレイ、キーボード、マイクロ SD カードを接続することで、1980 年代前半の 8bit パソコン程度の能力を発揮する。コンピュータサイエンスを学ぶ大学や高専の学生が、実際に動作する PC の例として使用したり、設計を解析する目的で設計してある。

TaC 上では C--言語*9で記述された TacOS*10 が動作する。本書では TacOS をオペレーティングシステムの実装例として参照する。

2.6.1 ハードウェア構成

図 2.9 に TaC のハードウェア構成を示す。16 ビットのシングルプロセッサ (CPU が一つ), 主記憶 64KiB の非常に単純なシステムである。単純なのでオペレーティングシステムの構築も容易である。TaC に関する資料を付録 A にまとめる。

- コンソールパネル

図 2.8 「(a) TeC7 の写真」で、TeC7 本体右半分のランプやスイッチで構成される部分をコンソールパネルと呼ぶ。コンソールパネルは CPU や主記憶と直接接続されており、CPU を停止した状態で、CPU や主記憶の内容を操作したり観察したりすることができる。また、機械語命令を一命令毎に実行するステップ実行機能や、ある番地の命令を実行した時点でプログラムを停止するブレーク機能ポイントが利用できる。コンソールパネルの機能はハードウェアで実現されているの

*8 詳細は <https://github.com/tctsigemura/TeC7> を参照のこと。

*9 C 言語に似た言語、詳細は <https://github.com/tctsigemura/C--/blob/master/doc/cmm.pdf> を参照のこと。

*10 詳細は <https://github.com/tctsigemura/TacOS> を参照のこと。

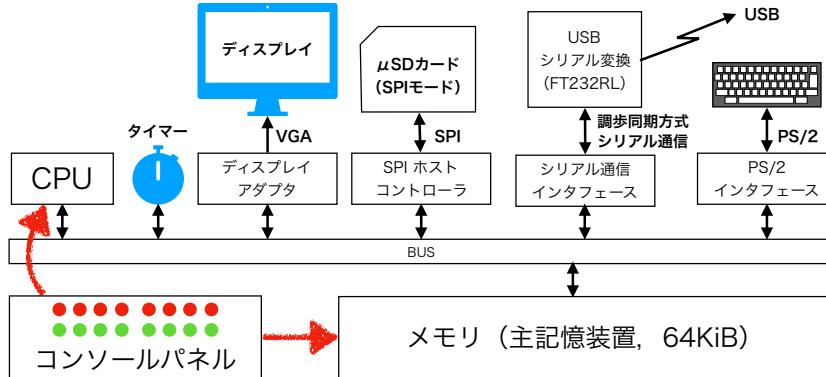


図 2.9 TaC のハードウェア構成

で、オペレーティングシステムの内部をステップ実行することも可能である。TacOS の開発では、コンソールパネルがデバッグに活用された。

- **CPU**

図 A.2 に示すような CPU レジスタと PSW を持つ 16 ビット CPU である。PSW のフラグに実行モードを表す P ビットを持ち、カーネルモードとユーザモードを切り換えることができる。機械語命令は、図 A.3 に示す 46 種類が準備されている。機械語命令のアドレッシングモードは 8 種類ある。

- **メモリ**

メモリは図 A.4 に示す構成である。メモリ空間全体で 64KiB、自由に使用できるメモリが 56KiB、2KiB の VRAM と 4KiB の IPL、32B の割込みベクタからなる。メモリは 8 ビット単位、または、16 ビット単位で読み書きできる。16 ビット単位の場合は偶数アドレスを用いる。

- **タイマー**

1 ミリ秒から $2^{16} - 1$ ミリ秒までの間隔で割込みを発生するインターバルタイマーが二つ利用可能である。

- **ディスプレイアダプタ**

80 文字 × 24 行の文字を VGA ディスプレイに表示する。メモリ空間の E000h から配置される VRAM に書き込んだ ASCII コードと対応する文字をディスプレイに表示する。E000h 番地がディスプレイの左上隅に対応する、E001h 番地が一行目の 2 文字の位置、E04Fh 番地が一行目の 80 文字の位置、E050h 番地が二行目の 1 文字の位置に対応する。

- **SPI ホストコントローラ**

スロットに挿入された μ SD カードを SPI モードに切換え読み書きを行う。SPI ホストコントローラに初期化コマンドを発行すると、μ SD カードを SPI モードに切換える。ブロックアドレスとメモリアドレスを設定して読み出しこマンドを発行すると、μ SD カードの指定したブロックから 512 バイトのデータを CPU を介さずに (DMA : Direct Memory Access を用いて) メモリに読み出す。書き込みコマンドを発行すると、メモリから指定ブロックにデータを書き込む。

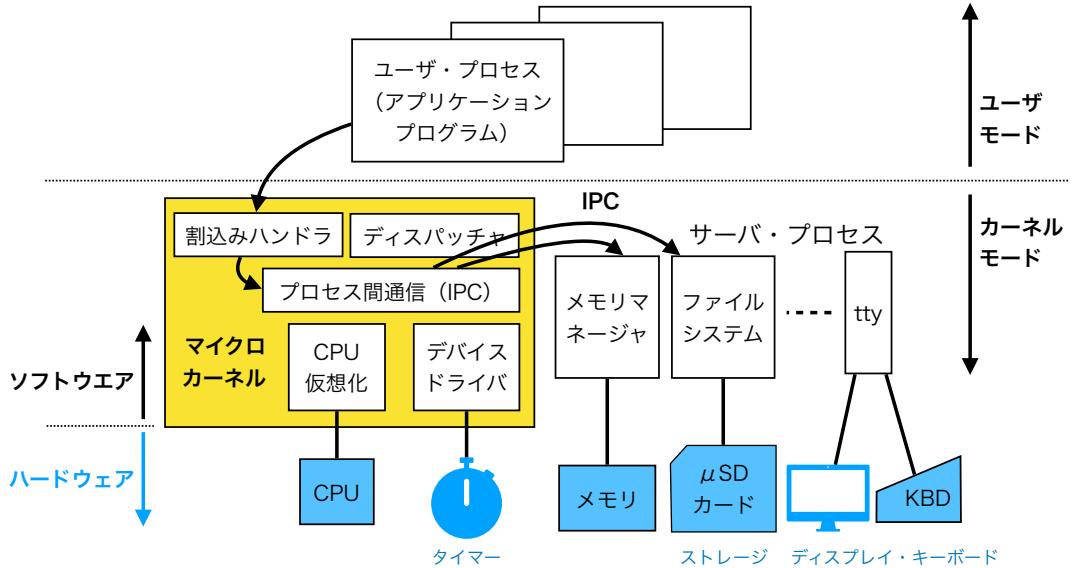


図 2.10 TacOS の構成

- シリアル通信インターフェース

調歩同期方式, 9,600Baud の通信インターフェースである。USB シリアル変換 IC を通して PC 等のシリアルターミナルと通信できる。1 バイト転送する毎に割込みを発生する。

2.6.2 TacOS

図 2.10 に TaC 用の OS である TacOS の構造を示す。マイクロカーネルがプロセス間通信 (IPC) 機能を提供し、サーバプロセスがメモリ管理やファイルシステム機能を提供する。図 2.7 の一般的なマイクロカーネル方式と異なり、サーバプロセスがカーネルモードで動作しハードウェアに直接アクセスする。また、サーバプロセスはマイクロカーネルと同じアドレス空間で動作するので、カーネル内ルーチンを CALL 機械語命令で直接に呼び出すことができる。

割込みや SVC 命令の実行が原因で、ユーザプロセスはカーネルモードに切り替わりマイクロカーネル内の割込みハンドラが呼び出される。割り込みハンドラで割込み原因を判断し、マイクロカーネル内のルーチンを呼び出したり、サーバプロセスの機能を IPC を用いて呼び出したりする。

2.7 もう一つの仮想マシン

1.1 で述べたように、オペレーティングシステムは抽象化され便利な拡張マシン（仮想マシン）を、必要な数だけ提供する。ここで述べた仮想マシンは、単一ユーザ・プロセスの実行環境のことである。同じ「仮想マシン」という用語が、オペレーティングシステムを実行することが可能な、よりハードウェアを忠実に再現した仮想マシンを指す場合もある。ここでは、一台のコンピュータ上で複数のオペレーティングシステムを実行可能な、もう一つの仮想マシンについて紹介する。

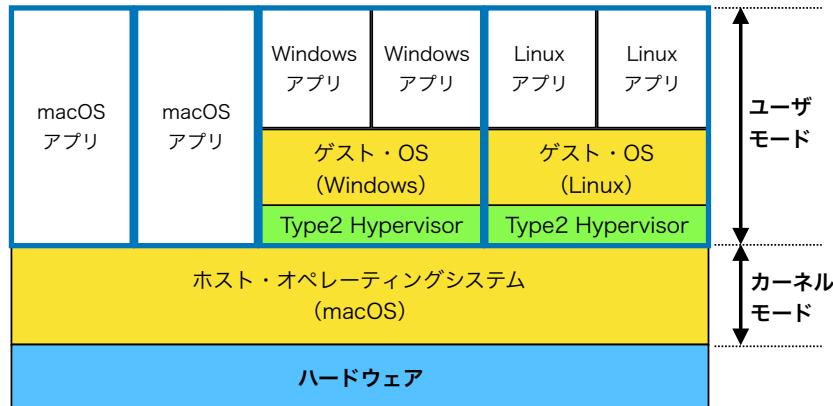


図 2.11 Type 2 ハイパーバイザ

2.7.1 Type 2 ハイパーバイザ

例えば、Mac を使用している人が Windows でしか動作しないアプリケーションを使用する場合を想像してしてみる^{*11}。予め Mac のハードディスクに macOS とは別に Windows もインストールしておき、電源投入時に macOS と Windows を選んでブートする方法もあるが、オペレーティングシステムを切換える度にコンピュータを再起動するのは不便である。また、macOS のアプリケーションと Windows のアプリケーションを同時に実行したい場合もある。

そこで、図 2.11 に示すような「Type 2 ハイパーバイザ (Type 2 Hypervisor)」を用いた仮想化が用いられる。ハイパーバイザはホスト・オペレーティングシステムの一つのユーザプロセスとして実行され、コンピュータ一台の機能をエミュレーションする。ハイパーバイザがエミュレーションするコンピュータの中で、ゲスト・オペレーティングシステムが稼働する。エミュレーションはソフトウェアだけで完全に行うのではなく^{*12}、ハードウェアの支援を受けて行うので高速に行うことができる^[35]。Type 2 ハイパーバイザとして有名な製品は、VMware Workstation, VMware Fusion, VirtualBox^{*13} 等である。

2.7.2 Type 1 ハイパーバイザ

メインフレーム上で 1960 年代から使用されている方式である。現在では PC サーバの仮想化にも使用されている。Type 1 ハイパーバイザはホスト・オペレーティングシステム無しにハードウェア上で直接実行される。Type 1 ハイパーバイザとして有名な製品は、IBM z/VM, VMware vSphere, Xen, Hyper-V 等である。

サーバ向けの製品が主流であり、例えば VMware vSphere は実行中のゲストを他の物理サーバに移動する等、非常に高度な機能を持っており^[36]、一台のサーバ上に効率よく多数の仮想マシンを動かすことができる。徳山高専情報電子工学科のパソコン室でも、2 台のサーバ上に 50 台の仮想デスクトップマシンを動かしていたことがある。

*11 徳山高専情報電子工学科のパソコン室では、Windows や Linux でしか動作しない Xilinx ISE WebPACK を Mac で使用している。

*12 完全にソフトウェアで行う場合もある。

*13 徳山高専情報電子工学科のパソコン室では macOS 上の VirtualBox で Windows を動作させている。

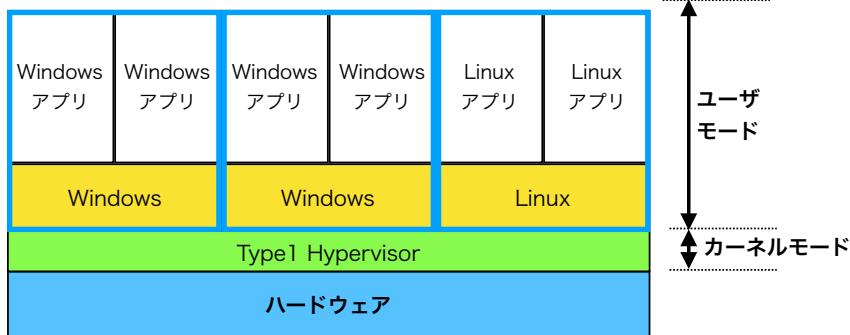


図 2.12 Type 1 ハイパーテーバイザ

2.7.3 仮想アプライアンス

ゲスト・オペレーティングシステムとアプリケーションまでインストールし、すぐに使用できる状態で配布される仮想マシンである。例えば、メールフィルタソフトをインストールした仮想マシンを入手しハイパーテーバイザで実行するだけですぐにメールフィルタリングが開始できる。

同じ手法で、すぐに使用できるパーソナルコンピュータ用のデスクトップ・オペレーティングシステムが配布されている場合もある。Linux の一種である Ubuntu の場合、VirtualBox ですぐに実行できるディスクイメージがダウンロードできる [37]。仮想アプライアンスは、ソフトウェアの新しい流通手法である。

2.8 まとめ

本書は SMP (対称型マルチプロセッシング : Symmetric Multiprocessing) のコンピュータを前提にしている。CPU は PSW (Program Status Word) と CPU レジスタを含んでいる。最近の Intel 社の CPU では、従来の CPU を Core (コア)、複数のコアを含んだ LSI のことを CPU と呼ぶ。

オペレーティングシステムのカーネルは、割込みハンドラ、ディスパッチャ、サービスモジュール、デバイスドライバ等から構成される。ユーザ・プロセスからカーネルへの切換え原因は割込みだけである。ユーザ・プロセス毎に仮想 CPU、仮想メモリ空間、管理情報等を持っている。

カーネルの構成方式には、単層カーネル (モノリシック・カーネル) 方式とマイクロカーネル (micro-kernel) 方式の二種類があった。マイクロカーネル方式ではサービスモジュールをサーバ・プロセスとし、IPC (プロセス間通信) を用いてサービスを要求する。サービスモジュール間の独立性が高くなり高信頼性のシステムを構成可能であるが、IPC はオーバーヘッドが大きい。信頼性と性能はトレードオフの関係にある。

TaC は、本書でオペレーティングシステムの実装例として使用する TacOS を稼働させるコンピュータである。コンソールパネルを持ち、TacOS のカーネル内までステップ実行によるトレースが可能である。TacOS はマイクロカーネル方式の簡単なオペレーティングシステムである。本書では、しばしば TacOS のソースコードを実装例として参照する。

第 III 部

CPU 管理

第3章

CPU の仮想化

オペレーティングシステムは、ハードウェアを抽象化した使いやすい拡張マシン（仮想マシン）を必要な数だけ提供する。数に限りがある資源が必要な数あるように見せるために仮想化が行われる。CPU 資源も仮想化し、各プロセスが自分専用の CPU を持っているように見せかける。

3.1 時分割多重

CPU を仮想化するためには時分割多重が用いられる。ハードウェアである実 CPU の数は限られているので、時間を区切って実 CPU を使用するプロセスを次々に切換えていく。図 3.1 に CPU 仮想化の原理を示す。

実 CPU は図 2.2 のような構造をもつハードウェアである。プロセスの構造は図 2.6 に示した通りであり、仮想 CPU を含んでいる。実 CPU が短時間（例えば 10ms）に次々と実行するプロセスを切換えていくことで、複数のプロセスが夫々に専用の CPU を持ち並行して実行されているように見せかける。

まず、現在のプロセス実行中の実 CPU のコンテキストを、プロセスの仮想 CPU 領域に保存する。次に、新しく実行するプロセスの仮想 CPU 領域から実 CPU にコンテキストを読み込み、新しいプロセスの実行を再開する。一つのプロセスから別のプロセスに切換える処理をコンテキストスイッチと呼ぶ。また、実 CPU にコンテキストを読み込んで実行を再開することをディスパッチ、ディスパッチを行うプログラムをディスパッチャと呼ぶ。図 2.5 にもディスパッチャは描かれていた。

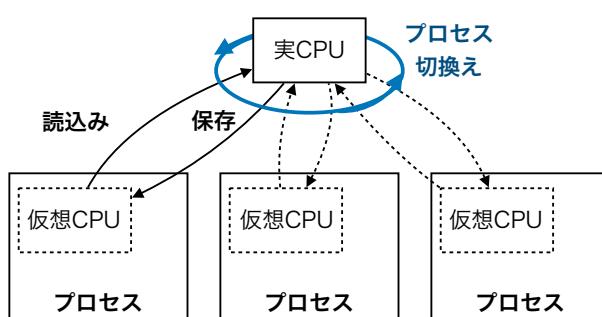


図 3.1 時分割多重による CPU の仮想化

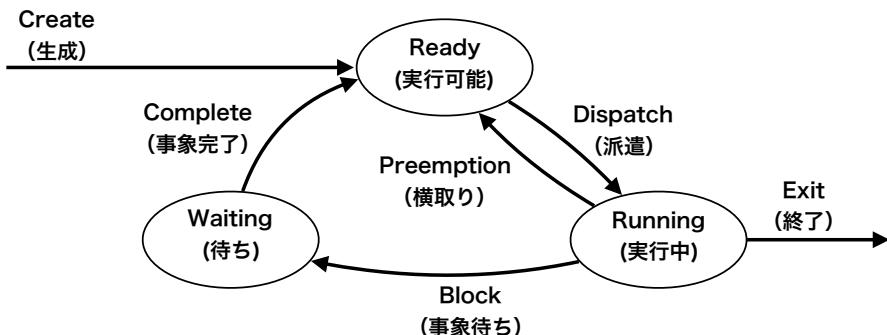


図 3.2 プロセスの状態遷移

3.2 プロセスの状態

プロセスは、キーボード等の入出力装置からの入力を待つ状態になったり、時間が経過するのを待つ状態になったりする。待ち (Waiting) 状態のプロセスには CPU を割当てる必要がない。このようにプロセスは幾つかの状態を持っている。プロセスの状態は UNIX では ps コマンドで確認できる。プロセスを模式的に示した図 2.6 では、「プロセス情報」の「プロセスの状態」のことである。

3.2.1 基本的な三つの状態

図 3.2 にプロセスの状態遷移図を示す。この図は最も簡単なものであり、実際のオペレーティングシステムでは、もっと状態数が多くなる^{*1}。図に示された三つの状態を説明する。

- **Ready (実行可能)**

CPU を割当てれば実行を開始できる状態のことである。プロセスは CPU が割当てられるのを待っている。

- **Running (実行中)**

CPU が割当てられ実行している状態のことである。CPU の数より多くのプロセスが同時に Running になることはできない。

- **Waiting (待ち)**

シグナルの到着や入出力の完了等の事象を待っている状態である。プロセスは実行することができない。

3.2.2 状態遷移

図 3.2 に示された六つの状態遷移の意味は以下の通りである。

1. **Create (生成)**

新しいプロセスが生成されると Ready 状態になる。親プロセスが `fork()` システムコール (UNIX の場合) や `CreateProcess()` システムコール (Windows の場合) を実行すると、新しい子プロ

^{*1} macOS の ps コマンドのオンラインマニュアルで確認すると、macOS ではプロセスの状態が、I (Idle), R (Runnable), S (Sleep), T (sTopped), U (Uninterruptible wait), Z (Zombi) の六つであることが分かる。

セスが生成される。

2. Dispatch（派遣）

Ready 状態のプロセスは、自分の順番が来たら CPU が割当てられ Running 状態に遷移し実行を開始する。

3. Preemption（横取り）

Running 状態のプロセスは、決められた時間（クォンタムタイム）を使い切ったとき、より優先度の高いプロセスが Ready 状態になったとき等に、CPU を取り上げられて Ready 状態に遷移する。

4. Block（事象待ち）

Running 状態のプロセスが、システムコールを発行して自ら Waiting 状態に遷移することがある。例えば入出力システムコール (`open()`, `read()`, `write()`, `close()` 等) や、シグナル待ちシステムコール (`pause()`, `wait()`, `sleep()` 等) を発行した場合である。また、他のプロセスからシグナルを受信した場合も、Waiting 状態に遷移することがある。更に、仮想記憶の機能を持つオペレーティングシステムでは、プロセスが読み書きしようとした領域がメモリ上に存在しない時もこの遷移が起こり、メモリ領域を確保するための処理がカーネル内部で始まる。

5. Complete（事象完了）

Waiting 状態のプロセスは、入出力の完了やシグナルの発生等の事象（イベント）が発生すると Ready 状態に遷移する。Waiting 状態のプロセスは停止しているのでプロセスが事象を発生することはない。事象はプロセスの外部からもたらされる。

6. Exit（終了）

プロセスが自ら `exit()` システムコール（UNIX の場合）や `ExitProcess()` システムコール（Windows の場合）を用いて終了する場合、プロセスがシグナルを受ける等して終了させられる場合に、この遷移が起こる。シグナルはプロセス（他プロセス、自プロセス）から明示的に送信される場合と、自プロセスが保護違反などのエラーを起こして発信される場合がある。

3.3 プロセスの切換え

Running 状態のプロセスが Block 遷移または Preemption 遷移し CPU を取り上げられると、他の Ready 状態のプロセスが CPU を割付けられ Dispatch 遷移し実行される。

3.3.1 切換えの原因

Running 状態のプロセスが状態遷移を起こす原因を以下にまとめ直す。

1. イベント

Running 状態のプロセスは、自ら「システムコールを発行」することで Block 遷移をすることがある。また、他のプロセスからの「干渉^{*2}を受け」Block 遷移することができる。

2. タイムスライシング

Running 状態のプロセスが長時間の実行を続けると Preemption 遷移をする。一度に実行しても良い時間（クォンタムタイム）を使い切ったためである。Ready 状態のプロセスが他にあれば、そ

^{*2} 干渉には、より優先順位の高いプロセスが実行可能になった、別のプロセスからシグナル等を受取った等がある。

のプロセスに実行が切換わる。他に実行すべきプロセスが無い場合は、再度、同じプロセスが実行される。

3.3.2 切換え手順

図 3.3 に二つのプロセス間で実行が切り換わる様子を示す。図では時間に従って上から下へ処理が進んでいく。左側はプロセス A の実行を、右側はプロセス B に実行を、図の中央はカーネルの実行を表している。プロセスの実行が切り替わっていく手順を以下で説明する。

1. 実行

日頃は CPU がユーザ・プロセスを実行している。

2. 割込み

割込みが発生し処理がプロセス A からカーネル内の割込みハンドラに移る。割込みの原因は 2.4.2 で述べた様々な原因が考えられる。割込みが発生すると以下の処理が CPU のハードウェアにより自動的にされる。

- CPU の (PC を含む) PSW がスタックに保存される。
- CPU の実行モードがカーネルモードに切り換わる。
- 割込みハンドラにジャンプする。

3. 割込みハンドラ

PSW (スタック上にある) と CPU レジスタ (図 2.2 参照) からなるプロセスのコンテキストをプロセスの仮想 CPU 領域 (図 2.6 参照) に保存する。次に割込み原因を調べ、割込み原因に応じた処理 (サービスモジュール等) にジャンプする。例えば、割込み原因が `open()` システムコールなら、`open` システムコールの処理を行うファイルシステムのサービスモジュールにジャンプする。割込み原因が I/O 完了なら、完了した I/O に対応するデバイスドライバにジャンプする。

4. サービスマジュール等

割込み原因に応じた処理を行う。この過程でプロセスの状態が変化せることがある。例えば、プロセスが発行したシステムコールが原因で Block 遷移する場合や、タイマーや I/O の完了割込により Waiting 状態だった別のプロセスが Complete 遷移する場合、タイマーの完了割込により現在のプロセスが Preemption 遷移する場合等が考えられる。サービスモジュールの処理が完了するとディスパッチャにジャンプする。

5. ディスパッチャ

実行可能なプロセスの中から適切な一つを選び、選んだプロセスの仮想 CPU 領域の内容を CPU レジスタにロードする。最後に PSW を復旧する機械語命令を実行しコンテキストを完全に CPU にロードし、プロセスの実行に戻る。CPU の実行モードを表すフラグは PSW に含まれているので、PSW が復旧されることで実行モードがカーネルモードからユーザモードに切り換わる。PSW を復旧する機械語命令として割込復帰用の **RETI (RETurn from Interrupt)** 命令を用いる。RETI 命令は単一の命令で PSW (PC とフラグ) を一度にスタックから復旧する。

6. 実行

新しく選択されたユーザ・プロセスが実行される。

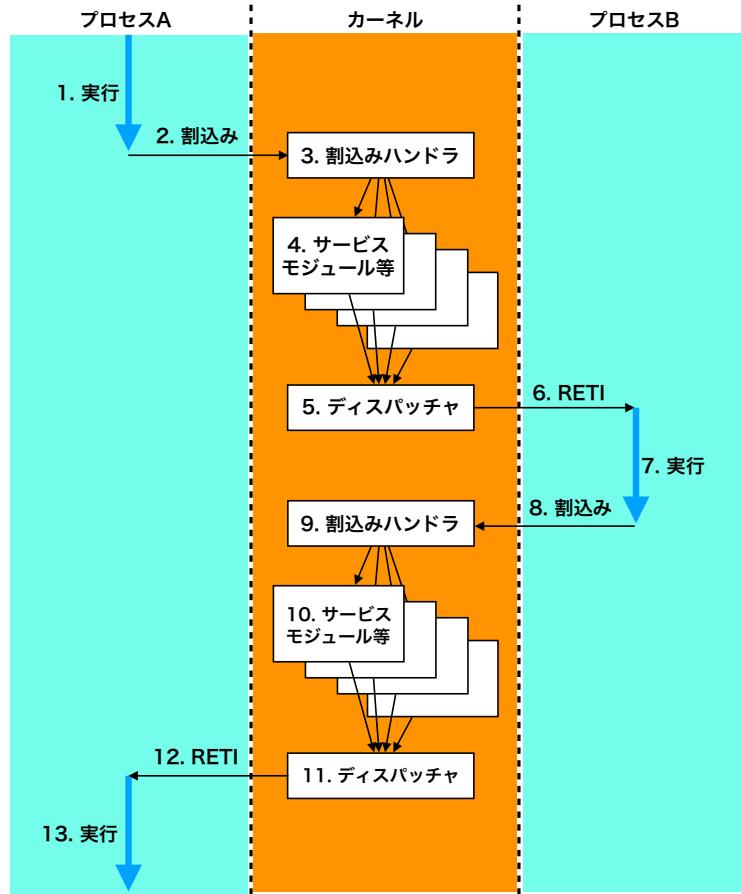


図 3.3 プロセスの切換え

図 3.3 の下半分、プロセス B からプロセス A へ実行が移る手順も上と同様である。

3.3.3 切換えの例

計算に長い時間を要する二つのプロセスだけがある時、クオンタムタイムを使い切ってもう一方のプロセスに切り換わり、交互に実行される様子を図 3.4 に示す。以下に手順を説明する。

1. 実行

プロセス A は計算処理を続けている。長い時間に渡ってシステムコールを発行することは無い。

2. タイマー割込み

タイマーは一定間隔で割込みを発生する。割込が発生すると CPU のハードウェアが自動的に PSW を保存し、割り込みハンドラにジャンプする。オペレーティングシステムは、主に、この割込みを基準に時間の経過を認識する。

3. 割込みハンドラ

プロセスのコンテキストをプロセスの仮想 CPU に保存する。その後、割込原因を調べタイマーからの割込みなので、「タイマーに関する処理」を行うカーネル内のモジュールへジャンプする。

4. タイマーに関する処理

一定間隔で発生するタイマーからの割込みを利用して、システムの時計を進めたり、リソース

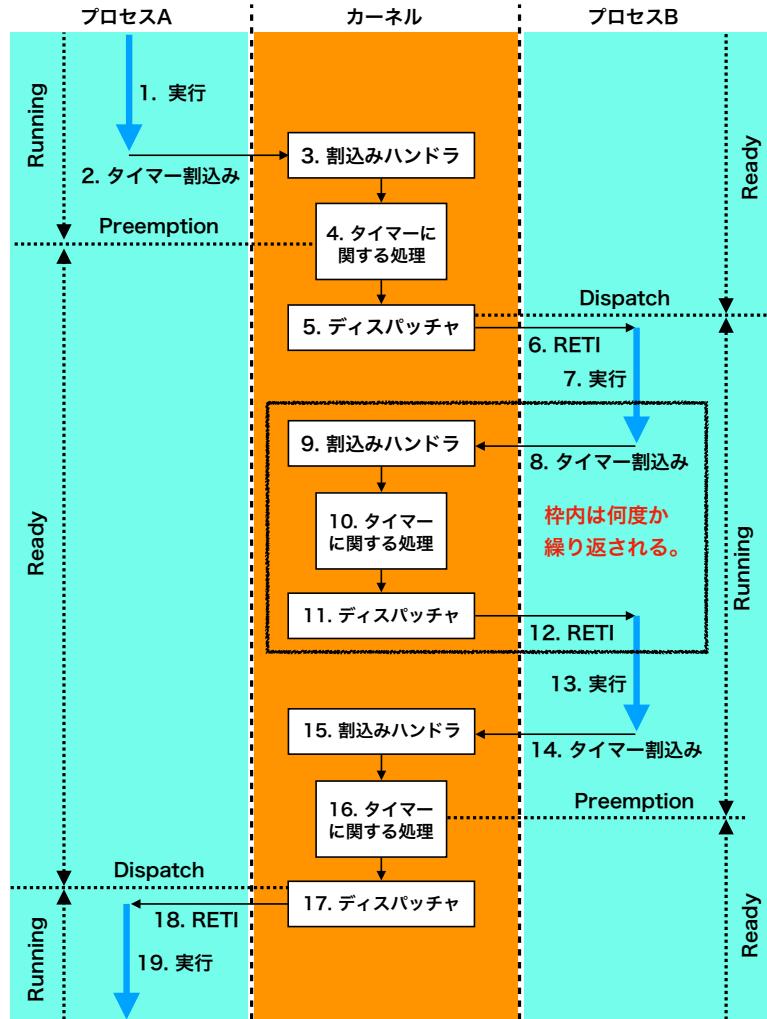


図 3.4 プロセスの切換えの例

(CPU やメモリ等) の利用統計データを更新したりする。その間にプロセス A がクォンタムタイムを使い切ったことが判明すると、プロセス A を Preemption 遷移させる。この時点でプロセス A の状態が Ready に変化する。

5. ディスパッチャ

Ready 状態のプロセスの中から適切な一つを選び Dispatch 遷移させる。図 3.4 はプロセス B が選択された場合である。ディスパッチャはプロセス B の CPU レジスタを復旧する。

6. RETI

プロセス B の PSW を復旧し、プロセス B の実行を再開する。

7. 実行

プロセス B は計算処理を再開する。プロセス B も長い時間計算を続けるプロセスとする。

8. タイマー割込み

計算を続けるうちにタイマーからの割込みが発生する。

9. 割込みハンドラ
プロセス B のコンテキストを保存する.
10. タイマーに関する処理
プロセス B は、まだ、クオントムタイムを使い切っていないので、Preemption は発生しない.
11. ディスパッチャ
Preemption は発生しないので、プロセス B のコンテキストを復旧する.
12. RETI
プロセス B に戻る.
13. 実行
プロセス B は計算処理を再開する.
14. タイマー割込み
8.~13. を何度か繰り返し、クオントムタイムを使い切った時のタイマー割込みである.
15. 割込みハンドラ
プロセス B のコンテキストを保存する.
16. タイマーに関する処理
クオントムタイムを使い切ったので Preemption が発生する.
17. ディスパッチャ
Ready 状態のプロセス A を選択し Dispatch 遷移させる。プロセス A のコンテキストを復旧する.
18. RETI
プロセス A に戻る.
19. 実行
プロセス A は計算処理を再開する.

3.4 PCB (Process Control Block)

PCB はプロセス毎に用意される最も重要なカーネルのデータ構造である。PCB はカーネル内のプロセステーブルに格納される。

3.4.1 PCB の内容

PCB は、図 2.6 に示した模式的なプロセスの構造図の「仮想 CPU」と「プロセス情報」を合わせたものに相当する。PCB には以下のような情報が格納される。

- 仮想 CPU
- プロセス番号
- 状態 (Running, Waiting, Ready 等)
- 優先度
- 統計情報 (CPU 利用時間等)
- 次回のアラーム時刻
- 親プロセス
- 子プロセス一覧

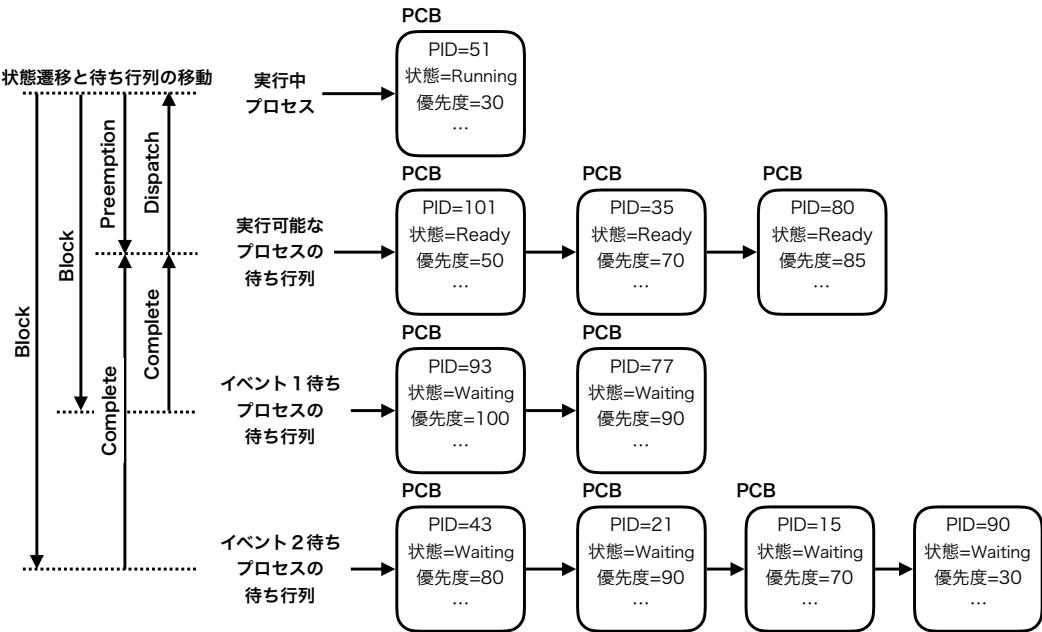


図 3.5 PCB のリスト

- シグナルハンドリング
- 使用中のメモリ
- オープン中のファイル
- カレントディレクトリ
- プロセス所有者のユーザ番号
- PCB のリストを作るためのポインタ

3.4.2 PCB リスト

PCB はプロセスを表現するデータ構造である。例えば、Ready 状態のプロセスは優先度順にソートされ、優先順位が最も高いものから順に CPU が割当てられる。ソートされた Ready 状態のプロセスのリストは、優先度をキーにソートされた PCB の線形リスト（待ち行列）として表現される。その様子を図 3.5 に示す。図は、数値が小さいほど優先度が高い意味になっている。

Ready 状態のプロセスだけでなく、Running 状態のプロセスや、Waiting 状態のプロセスも待ち行列として表現される。Waiting 状態のプロセスは、待ち合わせているイベント毎に待ち行列を作っている。イベント待ちの待ち行列のソート順はイベント毎にルールが決められる。

プロセスの状態遷移に合わせて PCB が待ち行列の間を移動する。図 3.5 の左側の「状態遷移と待ち行列の移動」が「どの待ち行列から、どの待ち行列に移動可能か」を表している。例えば、Running 状態（実行中）のプロセスが Preemption 遷移をすると、状態が Ready に変わるだけでなく、PCB が「実行可能なプロセスの待ち行列」に移動する。この移動ルールは図 3.2 の状態遷移と一致している。

3.5 TacOS の CPU 仮想化

実例として TacOS^{*3}の例を紹介する。 TacOS はマルチプロセスのオペレーティングシステムである。以下では CPU の時分割多重に必要なプロセス切換え機構を紹介する。

3.5.1 PCB

PCB はプロセス切換え機構にとって最も重要なデータ構造である。 TacOS の PCB は図 3.6 に示す PCB 構造体として定義されている^{*4}。 PCB 構造体の内容を順に説明する。

- 仮想 CPU(sp)

TacOS はプロセスのコンテキストのほとんどをカーネルスタック上に保存する。 そして、保存位置を表すスタックポインタ (SP) だけを PCB に保存する。 PCB に保存されるのは仮想 CPU の一部だけである。

- プロセス番号 (pid)
- 状態 (stat)

TacOS のプロセスの状態は以下の三つである。

1. P_RUN

Running と Ready の二つを兼用している。 プロセスは実行可能プロセスの待ち行列（実行可能列）に挿入される際に P_RUN 状態になる。 実行中も P_RUN 状態のまま変更しない。

2. P_WAIT

Waiting 状態のことである。

3. P_ZOMBIE

プロセスが終了したが、 終了ステータスを親プロセスに渡していない状態である。 終了処理の途中状態と考えるとよい。

- 優先度 (nice, enice)

ゼロが最も高い優先度を表す。 優先度には、 本来の優先度 (nice) と、 実質の優先度 (enice) の二つがある。 現在の実装ではこの二つは同じ値を持つ。 将来、 動的に変化する優先度を採用する場合に、 enice の値が変化するようにする。

- プロセステーブルのインデクス (idx)

この PCB が登録されているプロセステーブル内の位置である。 プロセスが消滅する際にプロセステーブルから PCB を削除するために使用する。

- イベント用カウンタとセマフォ (evtCnt, evtSem)

セマフォはプロセス間の同期に使用する基本的な機構である^{*5}。 タイマー待ち、 子プロセスの終了待ち等で、 このセマフォを使用してプロセスを待ち状態にする。 カウンタはタイマーの待ち時間を計るため等に使用される。

- プロセスのアドレス空間 (memBase, memLen)

^{*3} TacOS の詳細は <https://github.com/tctsigemura/TacOS> を参照のこと。

^{*4} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/process.hmm> の一部である。

^{*5} 詳しくは後の章で解説する。

```
1 #define P_RUN    1      // プロセスは実行可能または実行中
2 #define P_WAIT   2      // プロセスは待ち状態
3 #define P_ZOMBIE 3      // プロセスは実行終了
4
5 // プロセスコントロールブロック (PCB)
6 // 優先度は値が小さいほど優先度が高い
7 struct PCB {           // PCB を表す構造体
8     int sp;             // コンテキスト (他の CPU レジスタと PSW は
9                     // プロセスのカーネルスタックに置く)
10    int pid;            // プロセス番号
11    int stat;            // プロセスの状態
12    int nice;            // プロセスの本来優先度
13    int enice;           // プロセスの実質優先度 (将来用)
14    int idx;             // この PCB のプロセステーブル上のインデクス
15
16 // プロセスのイベント用セマフォ
17    int evtCnt;          // カウンタ (>0:sleep 中, ==-1:wait 中, ==0:未使用)
18    int evtSem;           // イベント用セマフォの番号
19
20 // プロセスのアドレス空間 (text, data, bss, ...)
21    char[] memBase;       // プロセスのメモリ領域のアドレス
22    int memLen;           // プロセスのメモリ領域の長さ
23
24 // プロセスの親子関係の情報
25    PCB parent;          // 親プロセスへのポインタ
26    int exitStat;         // プロセスの終了ステータス
27
28 // オープン中のファイル一覧
29    int[] fds;            // オープン中のファイル一覧
30
31 // プロセスは重連結環状リストで管理
32    PCB prev;            // PCB リスト (前へのポインタ)
33    PCB next;            // PCB リスト (次へのポインタ)
34    int magic;            // スタックオーバーフローを検知
35};
```

図 3.6 TacOS の PCB 宣言ソースプログラム

TacOS には仮想記憶のような高度な機構は無い。各プロセスは、物理メモリの領域をオペレーティングシステムによって割付けられる。`memBase` はオペレーティングシステムがプロセスに割当てたメモリ領域の開始アドレス、`memLen` はメモリ領域のバイト数である。

- プロセスの親子関係の情報 (`parent`, `exitStat`)

TacOS のプロセスは親プロセスだけ記憶している。`parent` は親プロセスの PCB を指すポインタである。`exitStat` は P_ZOMBIE 状態になった時、親に渡すべき終了ステータスを保存する領域である。

- オープン中のファイル一覧 (`fds`)

プロセスがオープンしたファイルのファイルディスクリプタ（番号）の一覧を記憶する配列である。TacOS ではシステム全体で一意なファイルディスクリプタ（番号）が用いられる^{*6}。`close()` システムコールは、クローズするファイルディスクリプタが正当なものか調べるために、この配列を使用する。`exit()` システムコールは、プロセスを終了する前にプロセスの全オープンファイルをクローズするために、この配列を使用する。

- PCB リストの管理 (`prev`, `next`)

TacOS はプロセスのリストを PCB のリストとして表現する。TacOS の PCB リストは番兵付きの重連結環状リストである（図 3.8 参照）。`prev`, `next` はリスト上で前後のプロセスの PCB を指すポインタである。

- スタックオーバーフローの検知 (`magic`)

TacOS は PCB の直後にプロセスのカーネルスタックを配置する。万一、カーネルスタックがオーバーフローすると PCB が後ろから破壊される。`magic` はそれを検知するために使用される。

TacOS は PCB を初期化する際に `magic` に 0xabcd を格納する。カーネルスタックがオーバーフローすると、まず、`magic` 領域が破壊される。`magic` の値が変化していないかチェックすることで、カーネルスタックのオーバーフローを検知することができる。

3.5.2 メモリ配置

図 3.7 に TacOS 実行時のメモリマップを示す。図は二つのプロセスが実行中の例である。まず「物理メモリ空間」の配置について、次に「PCB とカーネルスタック」について説明する。

(a) 物理メモリ空間

- カーネル

カーネルのプログラムとデータ（変数）がこの領域に配置される。

- プロセス #1 の PCB とカーネルスタック

プロセス 1 の PCB とカーネルスタックが隣接して配置される。詳細は図 3.7(b) に示してある。

- プロセス #2 の PCB とカーネルスタック

プロセス毎に PCB とカーネルスタックが準備される。

- プロセス #1 のメモリ空間

^{*6} UNIX のファイルディスクリプタ（番号）はプロセス毎に 0 番から割振られる。

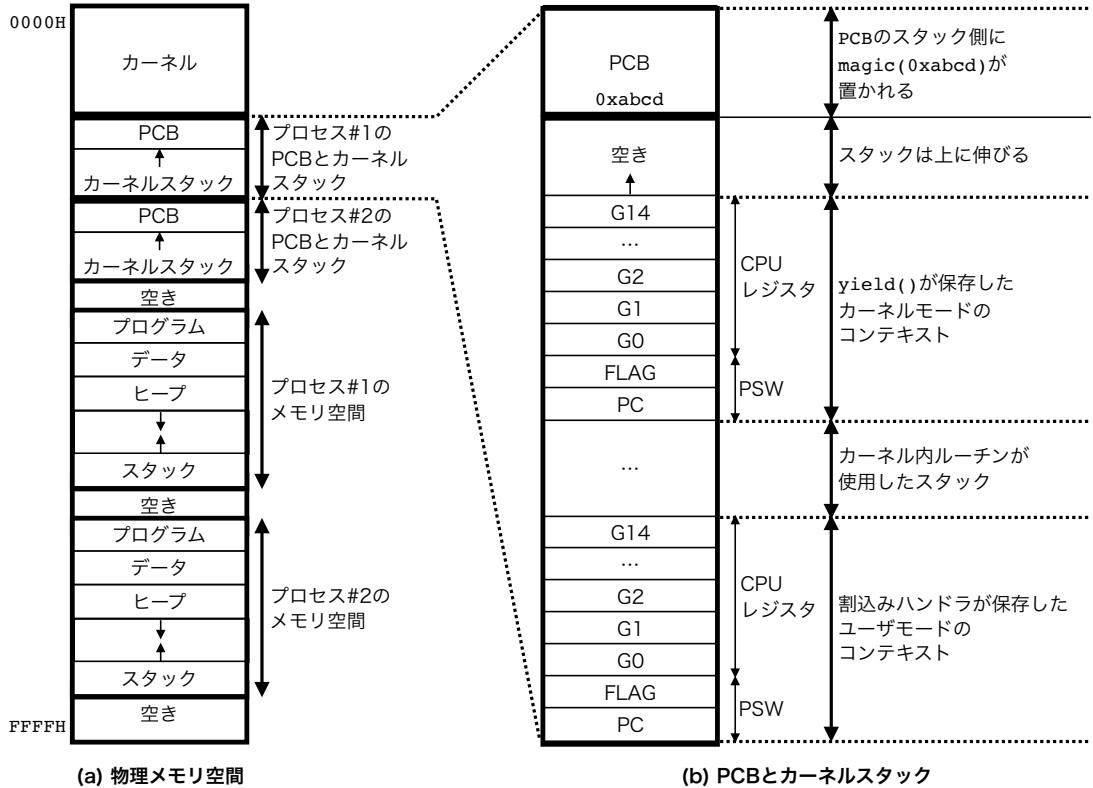


図 3.7 TacOS のメモリ配置

プロセス1のプログラム、データ、ヒープ、スタック領域が配置される。ユーザモードのプロセスは、この範囲以外のメモリをアクセスできないように保護すべきである。しかし、TaCはメモリ保護機構を持っていない。

- プロセス#2のメモリ空間

プロセス毎にメモリ空間が準備される。

(b) PCBとカーネルスタック

PCBとカーネルスタックはプロセスの生成時に隣接して領域が確保される。ユーザプログラム実行中はスタックの内容が空になる（スタックポインタがスタック領域の最大アドレスを指す）。割込みが発生すると自動的にPSWがカーネルスタックに保存され、実行モードがカーネルモードに切換わる。次に割込みハンドラに制御が移りCPUレジスタをスタックに保存した後、カーネル内ルーチンの実行が始まる。カーネル内ルーチンは、PCBに向かって伸びるカーネルスタックを使用する。そこで、PCBの最もスタック寄りにマジックナンバー（0xabcd）を配置し、スタックが伸びすぎPCBを破壊したことを検知するために使用する。

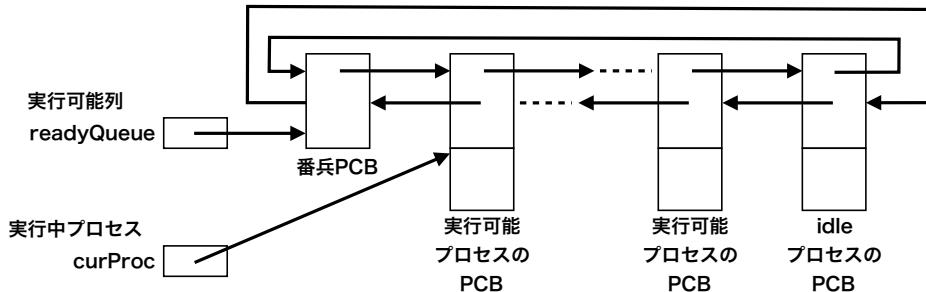


図 3.8 TacOS の実行可能列

3.5.3 プロセス切換えプログラム

図 3.9 に TacOS のプロセス切換えプログラムを示す^{*7}。切換えプログラムは、コンテキストを保存する `yield()`^{*8} と復旧する `dispatch()`^{*9} からなる。`dispatch()` は図 3.8 に示す実行可能列の先頭プロセスを実行する。TacOS は、PCB を実行可能列に置いたままプロセスを実行する。TacOS は、図 3.7(b) に示すように、プロセス毎にユーザモードとカーネルモードの二つのコンテキストを保存する。

割込が発生しカーネル内に実行が移ると、まず、割込みハンドラがユーザモードのコンテキストをカーネルスタックに保存する。次に、カーネルモードでカーネル内のプログラムが実行される。この時点では、割込み前のプロセスの一部として実行されている。プロセスを切換えるためには `yield()` を呼出す。`yield()` はカーネルモードのコンテキストをカーネルスタックに追加保存し、新しいプロセスの実行に切換える。次回、プロセスの実行が再開されるのは、カーネル内の `yield()` を呼出した位置になる。次に図 3.9 の内容を解説する。

2 行 プロセスを切換える時にカーネル内で呼出される `yield()` 関数の入口である。`yield()` 関数は、現在プロセスのカーネルモード・コンテキストを保存し CPU を解放する。

3~16 行 プロセスのカーネルモードのコンテキストをスタックに保存する処理である。`yield()` が (CALL 命令で) 呼出された時点で PC はスタックに格納されている。後で RETI 命令で PC と FLAG を同時に復旧するので PC の次に FLAG を格納している (7 行、図 3.7(b) 参照))。

17~19 行 プロセスのカーネルモードのコンテキストを保存したスタックの位置を PCB に保存する。`_curProc` 変数には現在のプロセスの PCB を指すポインタが保存されている (図 3.8 参照)。PCB 先頭の `sp` 領域 (図 3.6 参照) にスタックポインタを保存する。ここまで処理でコンテキストの保存が完了した。

20~23 行 カーネルスタックがオーバーフローしていないか調べる。PCB の `magic` フィールドの値をチェックし `0xabcd` 以外の値になっていたら、カーネルスタックが隣接する PCB まで伸びた (オーバーフローした) と判断する。この場合、`.stkOverflow` ルーチンにジャンプしシステムを停止する。カーネルのエラーなので復旧は諦める。オーバーフローが検知されない

^{*7} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/dispatcher.s> の一部である。

^{*8} 高級言語から `yield()` 関数を呼出すと、アセンブリ言語の `_yield` ルーチンが実行される。

^{*9} 高級言語から `dispatch()` 関数を呼出すと、アセンブリ言語の `_dispatch` ルーチンが実行される。

場合は26行に進み、新しいプロセスにディスパッチする。

26行 プロセスにCPUを割り付けるディスパッチャ(`dispatch()`関数)の入口である。

27~31行 まず、実行可能列(`_readyQueue`)の先頭プロセスのPCBアドレスを`_curProc`変数にセットする。実行可能列は番兵付きの重連結環状リストなので番兵の次が先頭のPCBである(図3.8参照)。実行可能なプロセスが無い場合はidleプロセスが選択される。`_curProc`が更新されたので、新しいプロセスが現在のプロセスになった。次に、現在のプロセスの保存してあったスタックポインタ(`sp`)を復旧する。

31~42行 スタックポインタが復旧されたので、スタックからCPUレジスタを復旧する。

43~44行 RETI命令を用いてPSW(FLAGとPC)を復旧し、前回プロセスが`yield()`を呼出した位置に戻る。`yield()`を呼出した位置に戻るためにRET命令ではなくRETI命令を使用するのは、プロセス生成時は例外的に、このRETIで実行モードを切換えてユーザプログラムの実行を開始するからである。

3.6 スレッド(Thread)

ここまで、一つのプロセスが一つの仮想CPUを持つモデルを考えてきた。しかし、実際のコンピュータのハードウェアはCPUを複数持つ場合がある。これでは「ハードウェアの機能を抽象化した便利な拡張マシン」(1.1.1参照)であるはずのプロセスが、「CPUが一つしかない縮小マシン」になっている。そこで、プロセスが複数の仮想CPUを持つモデルを導入する。これにより、一つのプロセスに並列実行する複数の処理の流れ(スレッド)を持つことが可能になる。

3.6.1 スレッドの役割

複数のプロセス(ジョブ)を主記憶にロードしておくことでCPUの利用効率を高くできることは既に説明した(9ページ、マルチプログラミング参照)。マルチプログラミングの、もう一つのメリットは、プログラミングが簡単になる場合があることである。以下ではWebサーバを例に、マルチプログラミングによる改善を紹介する。

- マルチプログラミングなし

図3.10(a)に最も簡単なモデルを示す。Webサーバはリクエストを受信すると、それに対するレスポンスを返す。処理は1番目のクライアントから順に行われ、2番目のクライアントは1番目の処理が終了するまで待たれる。このモデルの問題点は、処理中にWebサーバプロセスがI/O待ち等でブロック(Block)する可能性があり、その間、他のクライアントへのサービスがされないことである。

2番目以降のクライアントが長時間待たされないように、複数のクライアントの処理を並行してできるように改良したモデルが図3.10(b)である。「I/O完了の監視」は通信を含む複数の入出力を同時に監視し、どれかが読み書き可能になるのを待つ機能である。UNIXでは`select()`システムコールがこの機能を持つ。読み書き可能になったことを確認後に読み書きを行うのでプロセスがブロックすることが無くなり、複数のクライアントに対して同時にサービスを行うことができる。

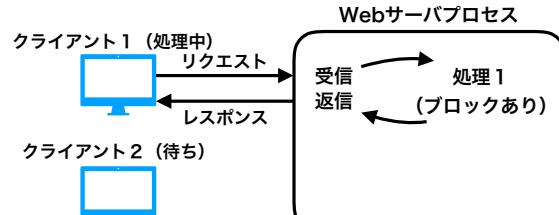
しかし、Webサーバのプログラミングは難しくなる。一方のクライアントの処理が終わらないうちに、別のクライアントの処理を開始する必要があるからである。クライアント毎に処理がどこま

```

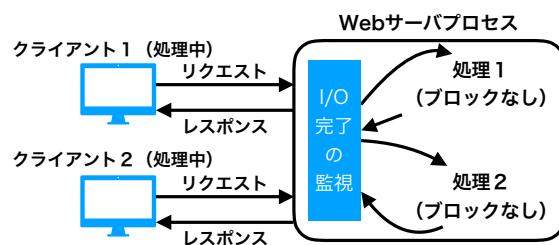
1 ; 現在のプロセス (curProc) が CPU を解放する。その後、新プロセスへディスパッチする。
2 _yield                                ; 高級言語からは yield() 関数として呼出す。
3     ;--- G13(SP) 以外の CPU レジスタと FLAG をカーネルスタックに退避 ---
4     push    g0                      ; FLAG の保存場所を準備する
5     push    g0                      ; G0 を保存
6     ld      g0,flag                ; FLAG を上で準備した位置に保存
7     st      g0,2,sp                ;
8     push    g1                      ; G1 を保存
9     push    g2                      ; G2 を保存
10    push   g3                      ; G3 を保存
11    ...
12    ...                            ; G4 から G10 も同様に保存する
13    ...
14    push    g11                     ; G11 を保存
15    push    fp                      ; フレームポインタ (G12) を保存
16    push    usp                     ; ユーザモードスタックポインタ (G14) を保存
17    ;----- G13(SP) を PCB に保存 -----
18    ld      g1,_curProc            ; G1 <- curProc
19    st      sp,0,g1                ; [G1+0] は PCB の sp フィールド
20    ;----- [curProc の magic フィールド] をチェック -----
21    ld      g0,30,g1                ; [G1+30] は PCB の magic フィールド
22    cmp    g0,#0xabcd             ; P_MAGIC と比較、一致しなければ
23    jnz   .stkOverflow           ; カーネルスタックがオーバーフローしている
24 ;
25 ; 最優先のプロセス(readyQueue の先頭プロセス)へディスパッチする。
26 _dispatch                             ; 高級言語からは dispatch() 関数として呼出す。
27     ;----- 次に実行するプロセスの G13(SP) を復元 -----
28     ld      g0,_readyQueue        ; 実行可能列の番兵のアドレス
29     ld      g0,28,g0                ; [G0+28] は PCB の next フィールド (先頭の PCB)
30     st      g0,_curProc            ; 現在のプロセス (curProc) に設定する
31     ld      sp,0,g0                ; PCB から SP を取り出す
32     ;----- G13(SP) 以外の CPU レジスタを復元 -----
33     pop   usp                     ; ユーザモードスタックポインタ (G14) を復元
34     pop   fp                      ; フレームポインタ (G12) を復元
35     pop   g11                     ; G11 を復元
36     ...
37     ...                            ; G10 から G4 も同様に復元する
38     ...
39     pop   g3                      ; G3 を復元
40     pop   g2                      ; G2 を復元
41     pop   g1                      ; G1 を復元
42     pop   g0                      ; G0 を復元
43     ;----- PSW(FLAG と PC) を復元 -----
44     reti                           ; RETI 命令で一度に POP して復元する

```

図 3.9 TacOS のプロセス切換えプログラム



(a) 最も基本的なWebサーバのモデル



(b) 改善したWebサーバのモデル

図 3.10 マルチプログラミングを用いない Web サーバ

で進んでいるのかを表す状態を持つ必要がある。また、CPUが複数存在する場合でも、同時に一つのCPUしか動かないことも問題である。

- マルチプロセス

マルチプログラミングを用いることで前記の問題を解決したモデルを図 3.11(a) に示す。Web サーバプロセスは、まず、接続要求を待ちクライアント 1 からの接続を受け入れる。次に、クライアント 1 専用のサーバプロセスを生成し処理を任せる。Web サーバプロセスは、生成したプロセスの終了を待たずに、次の接続要求待ちになる。クライアント 2 からの接続要求があったらクライアント 2 専用のサーバプロセスを生成し、接続要求待ちに戻る。

このモデルなら、各クライアントの処理を別々のプロセスが行っているので、プロセスがブロックしても構わない。そのため、プログラミングは簡単になる。また、CPUが複数あればプロセスが真に並列に実行される。しかし、プロセスの生成はメモリ空間の確保や初期化を含み重い処理である。また、プロセスはメモリを共有していないのでプロセス間の情報共有には効率が悪い。

- マルチスレッド

複数のスレッドを使用したモデルを図 3.11(b) に示す。マルチプロセスの場合と良く似たプログラムであるが、クライアント毎に専用のプロセスを作る代わりに、クライアント毎に専用のスレッドを作る。スレッドの生成はプロセス生成より 10~100 倍速いと言われている [38]。また、スレッドはメモリを共有しているので情報共有には都合が良い。例えば、Web サーバが頻繁に参照されるページをメモリ上にキャッシュする場合、キャッシュをスレッドで共有できる。

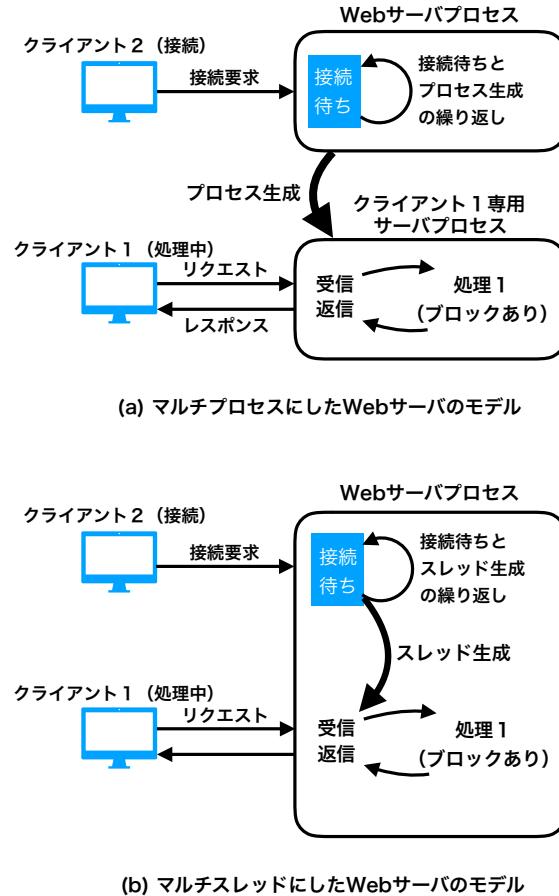


図 3.11 マルチプログラミングを用いる Web サーバ

3.6.2 スレッドの形式

読者は、「スレッドはカーネルが実現する」と暗黙のうちに考えていたかも知れない。しかし、ユーザプログラム（ライブラリ）内でスレッドを実現することもある。カーネルが実現するスレッドをカーネルスレッド、ユーザプログラム内で実現するスレッドをユーザスレッドと呼ぶ。

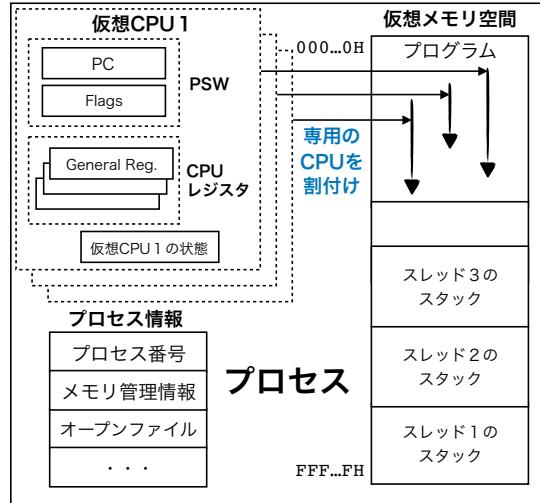
- カーネルスレッド

カーネルスレッドの模式図を図 3.12(a) に示す。カーネルスレッドはプロセスの仮想 CPU を複数にし、仮想 CPU がプログラムを並行して実行する。「プロセス情報」から「プロセスの状態」は無くなり、代わりに仮想 CPU 毎に「仮想 CPU の状態」を管理するようになる。CPU が複数ある時、カーネルスレッドであれば、プロセス内を真に並列実行することが可能である。

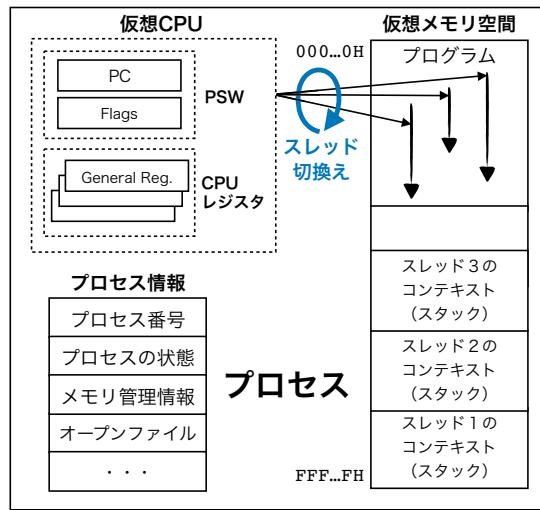
- ユーザスレッド

ユーザスレッドの模式図を図 3.12(b) に示す。プロセスには単一の仮想 CPU しかない。ユーザスレッドは仮想 CPU を時分割多重して実現される。カーネルを経由しないでスレッドの生成や切換えをすることができるので、オーバーヘッドが非常に小さい。

以下に述べるように、両者を組合せた三つのスレッドモデルが使用される。



(a) カーネルスレッド



(b) ユーザスレッド

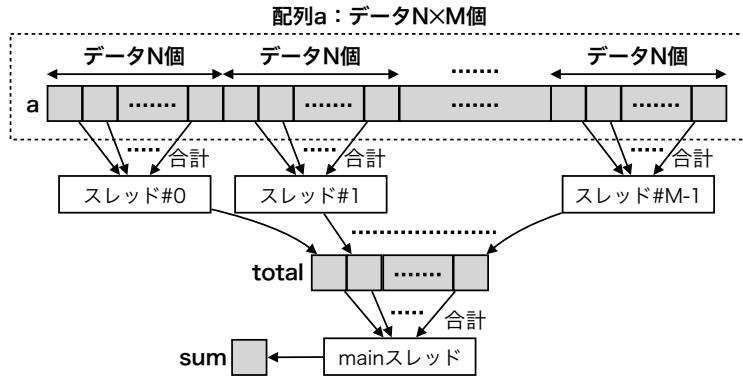
図 3.12 ユーザスレッドとカーネルスレッド

1. Many-to-One Model

複数 (Many) のユーザスレッドを一つ (One) のカーネルスレッドで実行するモデルである。図 3.12(b) に相当する。プロセス内にカーネルスレッドは一つしか存在しない。ユーザスレッドはユーザプログラム (ライブラリ) の工夫で单一のカーネルスレッドを複数に見せかけているだけなので、眞の並列実行にはならない。また、何れかのスレッドがシステムコールでブロックすると、全てのスレッドが停止してしまう問題がある。

2. One-to-One Model

全てのスレッドがカーネルスレッドのモデルである。図 3.12(a) に相当する。プロセス内にカーネルが管理する仮想 CPU が複数あるので、複数プロセスと同等な並列実行が可能である。しかし、スレッドの生成や切換えにカーネルが介入するので、処理は重くなる。また、システムによっては

図 3.13 M 個のスレッドで手分けして合計を計算する様子

生成できるスレッド数に制限がある。

3. Many-to-Many Model

複数の (Many) のユーザスレッドを複数の (Many) のカーネルスレッドで実行するモデルである。カーネルスレッドの数をユーザスレッドの数より多くすることはない。前記二つのモデルの折衷案である。

3.6.3 スレッドプログラミング

配列データの合計を求める処理をスレッドを用いて高速化する例を考えよう。図 3.13 に原理を示す。配列 a を M 分割し個別スレッドで (CPU が複数あれば) 同時に小計を計算する。小計は配列 total に格納する。最後に main スレッドが total の合計を求めると全体の合計 sum が計算できる。

POSIX スレッドによる実装

このアイデアを POSIX スレッド^{*10}を用いた C 言語プログラムにしたものを見図 3.14^{*11}に示す。

12 行の `thread()` 関数は M 個のスレッドで同時に並列実行される。配列 a の担当範囲等は引数 arg により指示される。関数の引数 (arg) やローカル変数 (args , sum , i) は、スレッドのスタック (図 3.12 参照) に割り付けられるので、スレッド毎に別の実体を持つ。グローバル変数 a や total 等は全てのスレッドで共有される。

33 行の `pthread_attr_init()` は引数の `pthread_attr_t` 型変数をデフォルトのアトリビュート値で初期化する。34 行の `pthread_create()` がスレッドを生成する関数である。新しいスレッドの実行は引数で指定された `thread()` 関数から始まる。`pthread_create()` の引数 p は、`thread()` 関数が実行を開始する時に arg 引数に渡される。

39 行の `pthread_join()` はスレッドの終了を待つ関数である。スレッドの終了が確認できたら、40 行で小計を sum に足し込んでいる。

^{*10} POSIX スレッドは UNIX 系のオペレーティングシステムで使用できる。

^{*11} このプログラムは macOS High Sierra で動作確認をした。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #define N 1000           // 1スレッドの担当データ数
5 #define M 10            // スレッド数
6 pthread_t tid[M];      // M個のスレッドのスレッド ID
7 pthread_attr_t attr[M]; // M個のスレッドの属性
8 int a[M*N];           // このデータの合計を求める
9 int total[M];          // 各スレッドの求めた部分和
10 typedef struct { int no, min, max; } Args; // スレッドに渡す引数の型定義
11
12 void *thread(void *arg) {           // 自スレッドの担当部分のデータの合計を求める
13     Args *args = arg;              // m番目のスレッド
14     int sum = 0;                  // 合計を求める変数
15     for (int i=args->min; i<args->max; i++) { // a[N*m ... (N+1)*m] の
16         sum += a[i];             // 合計を sum に求める.
17     }
18     total[args->no]=sum;          // 担当部分の合計を記録
19     return NULL;                 // スレッドを正常終了する
20 }
21
22 int main() {                      // mainスレッドの実行はここから始まる
23     // 擬似的なデータを生成する
24     for (int i=0; i<M*N; i++) {    // 配列 a を初期化
25         a[i] = i+1;
26     }
27     // M個のスレッドを起動する
28     for (int m=0; m<M; m++) {     // 各スレッドについて
29         Args *p = malloc(sizeof(Args)); // 引数領域を確保
30         p->no = m;                // m番目のスレッド
31         p->min = N*m;             // 担当範囲下限
32         p->max = N*(m+1);        // 担当範囲上限
33         pthread_attr_init(&attr[m]); // アトリビュート初期化
34         pthread_create(&tid[m], &attr[m], thread, p); // スレッドを生成しスタート
35     }
36     // 各スレッドの終了を待ち、求めた小計を合算する
37     int sum = 0;
38     for (int m=0; m<M; m++) {    // 各スレッドについて
39         pthread_join(tid[m], NULL); // 終了を待ち
40         sum += total[m];          // 小計を合算する
41     }
42     printf("1+2+ ... +%d=%d\n", N*M, sum);
43     return 0;
44 }
```

図 3.14 M 個のスレッドで分担して配列データの合計を求めるプログラム

表 3.1 スレッド数による実行時間比較

M N M*N	スレッド数 (M)・データ件数 (M*N)									
	1 10,000	2 5,000	3 3,333	4 2,500	5 2,000	6 1,666	7 1,428	8 1,250	9 1,111	10 1,000
経過時間 (s)	1.881	0.980	0.657	0.493	0.406	0.339	0.335	0.332	0.319	0.312
ユーザ CPU 時間 (s)	1.879	1.953	1.959	1.958	2.009	2.011	2.244	2.462	2.679	2.846
システム CPU 時間 (s)	0.002	0.002	0.002	0.001	0.001	0.002	0.003	0.003	0.003	0.002

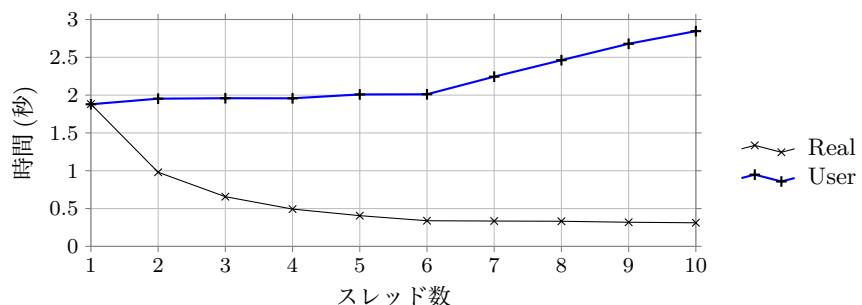


図 3.15 スレッド数による実行時間の変化

実行時間の計測結果

図 3.14 のプログラムの実行時間の計測結果を表 3.1 に、グラフにしたもの図 3.15 に示す^{*12 *13 *14}。

スレッド数が 1 の時は、経過時間 (Real) とユーザ CPU 時間 (User) が、ほぼ、同じになる。一つのコア^{*15}が全力で合計を計算した結果である。

スレッド数が 1~6 の間は、経過時間がスレッド数に反比例して短くなる。合計の計算時間に対応するユーザ CPU 時間は、ほぼ一定である。使用したコンピュータが持つ六つのコアが、最大で六つのスレッドに割当てられ、真に並列実行された結果である。

スレッドの数が 6~10 に増加する間、経過時間は、ほぼ一定である。しかしユーザ CPU 時間が増加している。必要な計算量は一定のはずなのに長い CPU 時間を必要とするので、コアの性能が悪化したように見える。

コアの性能が悪くなったように見えるのは、ハイパースレッディング・テクノロジー [39] により、コアの数が倍（12 個）あるように見せかけているためである。ハイパースレッディング・テクノロジーは、単一スレッドを実行する場合は遊んでしまうコア内のユニットを、二つのスレッドを同時に実行することで効率よく使用する技術である。見かけ上コアの数が二倍になるが、合計の性能は二倍には達しないので、コアあたりの性能が下がったように見える。

*12 実行時間の計測には OS X の `time` コマンドを用いた。

*13 実行時間が短すぎて比較し難いので、プログラムの 14 行から 17 行を 10 万回繰り返すように改造した上で計測した。

*14 計測に使用したコンピュータは OS X Yosemite をインストールした Mac Pro (Late 2013, 3.5GHz 6-Core Intel Xeon E5) である。C 言語コンパイラは Apple LLVM version 7.0.0 (clang-700.0.72) を使用した。

*15 従来の CPU のこと。

第4章

CPU スケジューリング

プロセス（スレッド）の実行順序を決めるこことをスケジューリングと呼ぶ^{*1}。システム内で最も貴重な資源であるCPUの割当てを決める重要な機能である。

4.1 評価基準

スケジューリングの良し悪しを判断する評価基準には次のようなものがある。

- **スループット (Throughput)**

単位時間あたりに処理できるジョブ数のことである。大きい方が良い。

- **ターンアラウンド時間 (Turnaround time)**

プロセスが実行できるようになってから終了するまでの時間のことである。短いほうが良い。バッチ処理で、ユーザがジョブを提出してから実行結果の印刷物が届くまでの時間をイメージすると分かりやすい。

- **レスポンス時間 (Response time)**

対話的なシステム（TSS やデスクトップパソコン）において、ユーザが操作した影響で出力が変化し始めるまでの時間である。例えば、エンターキーを入力したあと画面が変化を始めるまでの時間である。対話的なアプリケーションの操作性に大いに影響がある。当然、短いほうが良い。

- **締め切り (Deadline)**

制御用に用いられるリアルタイムシステム（Real-time system）では、決められた時刻（締め切り）までに結果を出すことが求められる。必ず時間を守らなければならない場合をハードリアルタイム（Hard real time）、できる限り時間を守らなければならない場合をソフトリアルタイム（Soft real time）と呼ぶ。オペレーティングシステムは、制御用プロセスが締め切りを守ることができるスケジューリングを行う必要がある。

- **その他**

システムの使用方法などにより様々な評価基準が考えられる。例えば、モバイルデバイスではバッテリーのために省エネルギーが評価基準になり得る。

^{*1} プロセスとスレッドの両方にあてはまることが多いので、この章ではプロセスのスケジューリングを前提に議論する。

4.2 システムごとの目標

システムの種類によって、スケジューリングの目標は異なる。表 4.1 に概略をまとめる。

- **メインフレーム**

バッチ処理を行う場合はユーザとの対話的な処理ではないので、スループットを優先する。例えば、コンテキストスイッチにも処理時間が必要なので、プリエンプションを行わないスケジューリング方式を採用し、コンテキストスイッチの回数を少なくすること等が考えられる。また、ユーザが結果を早く受取ることができるように、ターンアラウンド時間にも気を使う必要がある。

- **ネットワークサーバ**

ネットワークに接続され、複数のクライアントから同時に多数の要求を受付けて処理する。この場合は、クライアントを操作しているユーザの操作性を損なわないレスポンス時間と、多数の要求を処理するためのスループットが両立することが望まれる。両者のバランスが良いスケジューリングが求められる。

- **デスクトップパソコン**

一人のユーザが独占して使用するコンピュータである。ユーザは、複数の処理を同時にすることは少ない。ユーザの操作に素早く反応するためにレスポンス時間が重要である。例えば、ユーザがワードプロセッサを操作している間にバックグラウンドでメールの着信チェックを行うプロセスが動く場合、ワードプロセッサが軽快に動くことを重視し、メールの着信チェックプロセスの性能が落ちても構わない。ユーザが直接操作するプロセスを優先するスケジューリングが求められる。

- **モバイルデバイス**

ノートパソコンやスマートフォンのようなシステムでは、基本的にはデスクトップパソコンと同じようにレスポンス時間が重視される。しかし、バッテリーで駆動される場合は消費電力が少なくなるような工夫も必要である。例えば、プロセスの切換え頻度を少なくすることで、エネルギーの消費を小さくするスケジューリングを採用することが考えられる。

- **組込み制御用のコンピュータ**

締め切りまでに処理を完了することが重要である。例えば、時速 50km で走行するエレベータ^{*2}の制御コンピュータが、1 秒遅刻してブレーキを掛けたらどうなるだろうか。時速 50km は秒速 13m なので、エレベータは 13m 行き過ぎて停まることになる。最上階、または、最下階を目指しているとき 13m 行き過ぎるとエレベータは天井か床に激突してしまう。エレベータのブレーキ制御プロセスはハードリアルタイムに分類できる。同じエレベータでも、現在階数の表示はタイミングが少し遅れても大きな影響はない。エレベータの階数表示プロセスはソフトリアルタイムに分類できる。

^{*2} 高層ビルのエレベータの中にはもっと高速なものもある。

表 4.1 スケジューリングの目標

コンピュータの種類	重視する性能
メインフレーム (バッチ処理)	スループット, ターンアラウンド時間
ネットワークサーバ	レスポンス時間, スループット
デスクトップパソコン	レスポンス時間
モバイルデバイス	レスポンス時間, 省エネルギー
組込み制御	締め切り

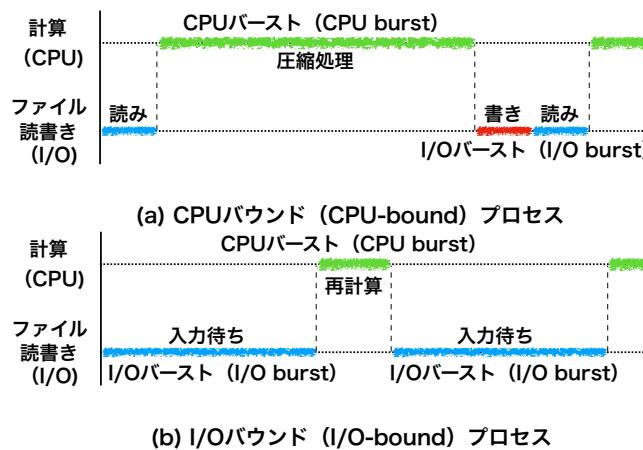


図 4.1 CPU バウンドと I/O バウンドプロセス

4.3 プロセスの振舞

一般に、プロセスは計算と入出力を繰り返す。計算と入出力にかかる時間の割合に応じて、二種類のプロセスに分類できる。

4.3.1 CPU バウンドプロセス

例として、動画を圧縮するビデオエンコーディング・プロセスを考えてみよう。プロセスは、図 4.1(a)に示すように、次の三つの処理を繰り返す。

1. 未圧縮の動画ファイルを少し読む。
2. 圧縮処理を行う。
3. 結果を圧縮済み動画ファイル書込む。

ビデオエンコーディング・プロセスは CPU が行う圧縮処理に長い時間がかかり、入出力にかかる時間が短い。このように CPU 処理にかかる時間が相対的に長いプロセスのことを **CPU バウンド (CPU-bound) プロセス** と呼ぶ。また、CPU が使用される期間を **CPU バースト (CPU burst)**、I/O が使用される期間を **I/O バースト (I/O burst)** と呼ぶ。CPU バウンドプロセスは長い CPU バーストと短い I/O バーストを持つ。

4.3.2 I/Oバウンドプロセス

二つ目の例としてスプレッドシート・プロセスを考えてみよう。スプレッドシート・プロセスは、まず、ユーザが何れかのセルにデータを入力するのを待つ。次に、入力されたデータを用いてスプレッドシートの再計算を行い結果を表示する。ユーザがセルにデータを入力するたびに同様な処理を繰り返す。このプロセスは図 4.1(b) に示すように、ユーザ操作を待つ長い入力待ちと、再計算と表示を行う短い CPU 処理を行う。このような、長い I/O バーストと短い CPU バーストを持つプロセスを **I/O バウンド (I/O-bound)** プロセスと呼ぶ。

4.4 スケジューリング方式

いくつかの代表的なスケジューリング方式を紹介する。

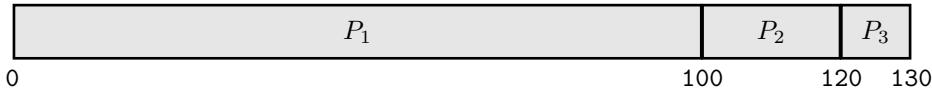
4.4.1 First-Come, First-Served (FCFS) スケジューリング

Ready 状態になった順（到着順）に実行する方式である。Running 状態になったらブロックするまで実行を継続する。プリエンプションはしない。以下の例では CPU バースト一回分の期間しか示さないが、実際は、図 4.1 に示すように CPU バーストが繰り返し発生する。

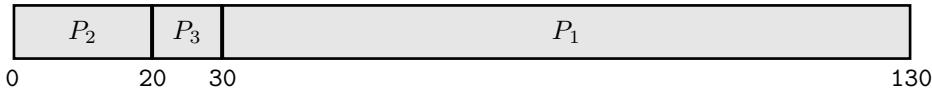
FCFS 方式は実行可能列を FIFO にするだけで実現できるが性能は良くない。例えば次の三つのプロセスが時刻 0 で、 P_1 , P_2 , P_3 の順に Ready 状態になったとする。

プロセス	到着時刻	CPU バースト時間 (ms)
P_1	0	100
P_2	0	20
P_3	0	10

この時、三つのプロセスの実行開始・終了の時刻を図で表すと次のようになる。



平均ターンアラウンド時間を計算すると、 $(100 + 120 + 130)/3 = 117 \text{ ms}$ となる。もしも、プロセスの到着順が P_2 , P_3 , P_1 の順だったとすると、三つのプロセスの実行開始・終了の時刻は図のようになる。



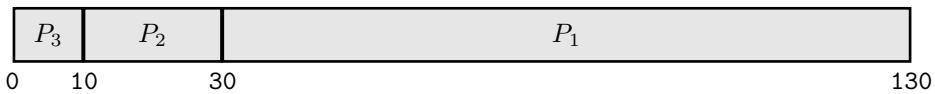
この場合の平均ターンアラウンド時間を計算すると、 $(20 + 30 + 130)/3 = 60 \text{ ms}$ となる。このように、FCFS では最悪な平均ターンアラウンド時間を選択することもある。プリエンプションをしないので、一旦、CPU バウンドなプロセスが実行を開始すると、他のプロセスは長い時間待たされる。

4.4.2 Shortest-Job-First (SJF) スケジューリング

SJF 方式^{*3}は、平均ターンアラウンド時間を最小にするスケジューリング方式である。SJF 方式では CPU バースト時間が短いものを先に実行するようにスケジューリングする。実行可能列は CPU バースト時間が短い順にソートされている。

三つのプロセスがあった時、実行順に各プロセスの実行時間が T_1, T_2, T_3 とすると、平均ターンアラウンド時間は、 $(T_1 + (T_1 + T_2) + (T_1 + T_2 + T_3))/3 = T_1 + T_2 * 2/3 + T_3/3$ となるり、先に実行したプロセスの実行時間ほど結果に及ぼす影響が大きいことが分かる。実行時間が短いプロセスを先に実行するスケジューリング方式は、平均ターンアラウンド時間を最小にする。

前出の三つのプロセスを SJF 方式でスケジューリングした時の、実行開始・終了時刻は次の図のようになる。



この図より平均ターンアラウンド時間を求めると $(10 + 30 + 130)/3 = 57 \text{ ms}$ となり、これまでで最短である。しかし、次回の CPU バースト時間を知ることは一般には不可能なので、SJF 方式は現実的な方式ではない。次回の CPU バースト時間を予測することで擬似的な SJF 方式を実現する。

次回の CPU バースト時間を予測する方法として、指数平滑平均 (exponential average) を用いる例を紹介する。次回の予測時間を T_{n+1} 、前回の予測時間を T_n 、前回の実際の CPU バースト時間を t_n とすると、 $0 \leq \alpha \leq 1$ の時、指数平滑平均は次の式で表すことができる。

$$T_{n+1} = \alpha t_n + (1 - \alpha)T_n$$

この式から

$$T_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} T_0$$

を得る。 $\alpha = 0.5$ の場合は、

$$T_{n+1} = 0.5t_n + 0.5^2 t_{n-1} + \cdots + 0.5^{j+1} t_{n-j} + \cdots + 0.5^{n+1} T_0$$

となる。この式は、過去の CPU バースト時間を、最近のものほど大きな重みを付けて平均したものになっている。つまり、次回の CPU バースト時間は、過去の CPU バースト時間と同程度であろうとの仮定に基づいた予測値を計算している。

4.4.3 Shortest-Remaining-Time-First (SRTF) スケジューリング

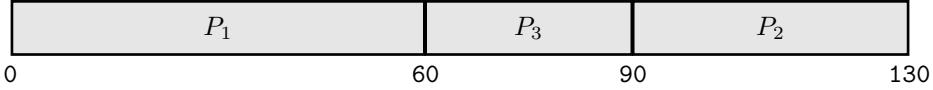
SRTF 方式^{*4}は、プリエンプション付きの SJF 方式である。プロセスが Ready 状態になると、このプロセスの CPU バースト時間と実行中のプロセスの残り CPU バースト時間とを比較し、残り CPU バースト時間の方が長いときプリエンプションをおこす。次の例で SJT と SRFT を比較してみよう。

^{*3} 皆さんの教科書では SPT のこと。

^{*4} 皆さんの教科書では SRPT のこと。

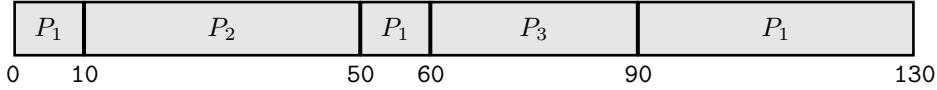
プロセス	到着時刻	CPUバースト時間(ms)
P_1	0	60
P_2	10	40
P_3	60	30

三つのプロセスを SJF でスケジューリングした場合は次の図のようになる。



平均ターンアラウンド時間を計算すると、 $((60 - 0) + (90 - 10) + (130 - 60)) / 3 = 70 \text{ ms}$ となる。

三つのプロセスを SRTF でスケジューリングした場合は次の図のようになる。



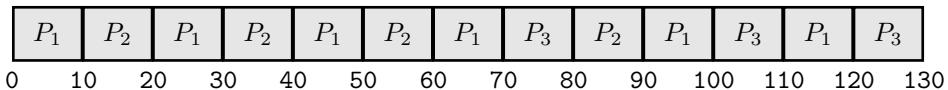
P_2 が到着した時、 P_2 の CPU バースト時間(40 ms)の方が P_1 の残り CPU バースト時間($60 - 10 = 50 \text{ ms}$) より短いので、 P_1 はプリエンプションし P_2 が先に実行される。 P_3 が到着した時も同様である。平均ターンアラウンド時間を計算すると、 $((130 - 0) + (50 - 10) + (90 - 60)) / 3 = 67 \text{ ms}$ となり、SJF よりも改善されている。

4.4.4 Round-Robin (RR) スケジューリング

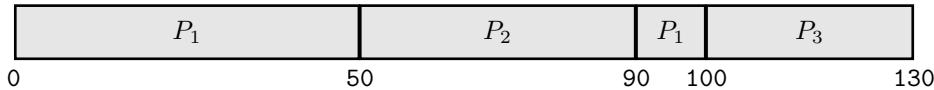
タイムシェアリングシステム (TSS) で使用された方式である。クォンタム時間 (time quantum), または、タイムスライス (time slice) と呼ばれる 10 ms ~ 100 ms 程度の一定の時間が予め決められている。実行可能列は FIFO になっている。実行可能列の先頭のプロセスに CPU が割り付けられて Running 状態になる。プロセスの実行がクォンタム時間連続するとプリエンプションが発生し、プロセスは実行可能列の最後尾に付け加えられる。

クォンタム時間 (q) が短いとレスポンス時間が短くなり、対話的な処理が円滑に行える。例えば、10 個のプロセスが CPU を奪い合うような状況でも、 $q = 10 \text{ ms}$ なら 100 ms に一度は全てのプロセスに CPU が割り付けられる。しかし、 q を小さくしすぎるとコンテキストスイッチの回数が多くなり、オーバーヘッドが大きくなる。逆に q が長いと FCFS と同じ結果になる。

前出の三つのプロセスを RR 方式 ($q = 10 \text{ ms}$) でスケジューリングした例を次の図に示す。なお、新規プロセスと、クォンタム時間を使い切りプリエンプションしたプロセスが、同時に実行可能列に追加される場合は、新規プロセスを優先することにする。



平均ターンアラウンド時間を計算すると、 $((120 - 0) + (90 - 10) + (130 - 60)) / 3 = 90 \text{ ms}$ となる。次に $q = 50 \text{ ms}$ でスケジューリングした例を示す。



平均ターンアラウンド時間を計算すると, $((100 - 0) + (90 - 10) + (130 - 60))/3 = 83 \text{ ms}$ となる. $q = 50 \text{ ms}$ でスケジューリングした方が, 平均ターンアラウンド時間が短くなった上に, コンテキストスイッチの回数が少ない. このようなプロセスの集合に対しては, $q = 10 \text{ ms}$ はクォンタム時間が短すぎると言える.

4.4.5 Priority (優先度順) スケジューリング

プロセス毎に決められた優先度を基に行うスケジューリング方式である. 実行中に優先度が変化する動的優先度を用いる方法と, プロセス生成時に決められ変化しない静的優先度を用いる方法がある. TacOS は静的優先度を用いる優先度順スケジューリング方式を用いる. SRTF 方式は, 次回 CPU バースト時間が短い順の動的優先度方式と考えられる.

優先度順スケジューリング方式の問題点は, 優先度の低いプロセスが全く実行されないスタベーション (starvation) が発生することである. この対策として, 実行可能列に留まるプロセスの優先度を徐々に高めしていくエージング (aging) が用いられる. 実行可能列に長く留まるプロセスは優先度が高くなり, やがて実行される.

4.4.6 Multilevel Feedback Queue (FB) スケジューリング

Windows, macOS, UNIX 等で広く使用されているスケジューリング方式である. 図 4.2 に示すように実行可能列を優先度別に複数設ける. 優先度が近いプロセスが同じ実行可能列に登録される. 同じ実行可能列では RR 方式でスケジューリングするので^{*5}, 列内でプロセスの順番は優先度とは関係がない. CPU を割り付ける際は, 優先度の高い実行可能列から順に調べ, 最初に見つかった空ではない実行可能列を使用する.

プロセスの優先度は動的に変化する方式を用いる. CPU バウンドなプロセスの優先度は急激に引き下げられ, プロセスは下位の実行可能列に移動する. 長く実行可能列に留まっているプロセスはエージングにより優先度が引き上げられ, 上位の実行可能列に移動する. 実行中のプロセスより上位の実行可能列にプロセスが登録されるとプリエンプションが発生し, 実行中のプロセスは CPU を取り上げられる.

4.5 TacOS のスケジューラ

実行可能になったプロセスをスケジューリングするプログラムをスケジューラと呼ぶ. スケジューラの例として, TacOS のスケジューラのソースプログラムを図 4.3 に示す^{*6}. TacOS の実行可能列は, PCB の番兵付き重連結環状リストとして表現する (図 3.8 参照).

1~7 行 関数 `insProc()` は, 実行可能列に PCB を登録するために使用される. スケジューラ以外からも呼出される汎用的なものである.

9~19 行 関数 `schProc()` がスケジューラである. `enice` がプロセスの優先度である. `enice` は

^{*5} 実行可能列ごとに, 異なるスケジューリング方式を採用することも可能である.

^{*6} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm> の一部である.

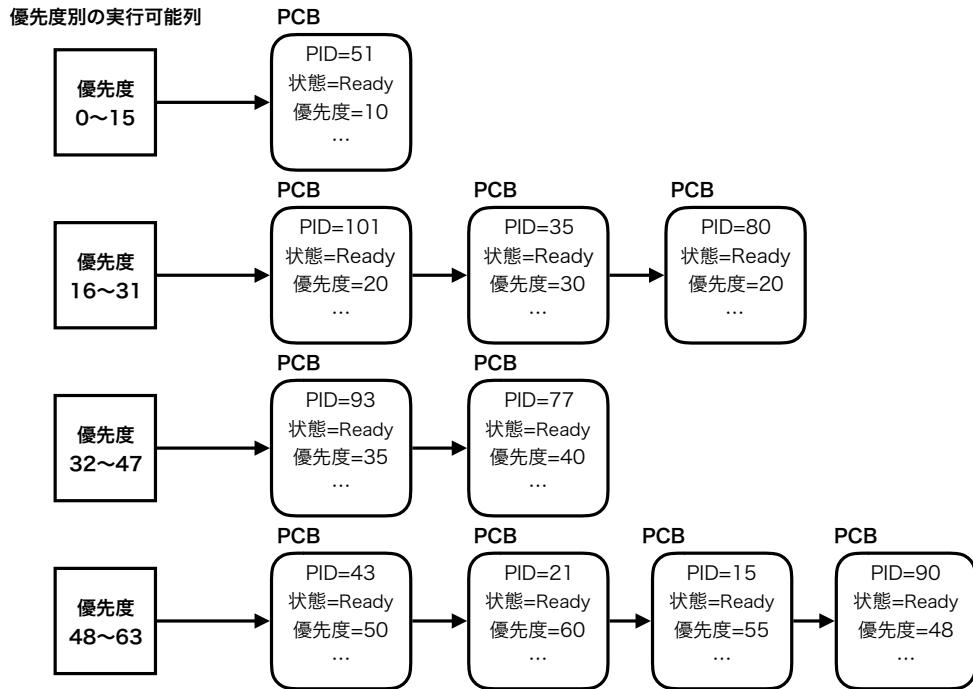


図 4.2 Multilevel Feedback Queue

```

1 // プロセスキーで p1 の前に p2 を挿入する p2 -> p1
2 void insProc(PCB p1, PCB p2) {
3     p2.next=p1;
4     p2.prev=p1.prev;
5     p1.prev=p2;
6     p2.prev.next=p2;
7 }
8
9 // プロセススケジューラ : プロセスを優先度順で readyQueue に登録する
10 // (カーネル外部からも呼び出されるのでここで割込み禁止にする)
11 public void schProc(PCB proc) {
12     int r = setPri(DI|KERN);           // 割り込み禁止、カーネル
13     int enice = proc.enice;
14     PCB head = readyQueue.next;       // 実行可能列から
15     while (head.enice<=enice)         // 優先度がより低い
16         head = head.next;             // プロセスを探す
17     insProc(head,proc);              // 見つけたプロセスの
18     setPri(r);                     // 直前に挿入する
19 }                                // 割り込み状態を復元する

```

図 4.3 TacOS のスケジューラ・ソースプログラム

値が小さい方が優先度が高い。スケジューラは、実行可能列（`readyQueue`）を番兵 PCB の次の PCB から開始して（14 行）、挿入するプロセスの `enice` より大きいものを探す（15, 16 行）。大きいものを見つけたら `insProc()` を使用して、見つけた PCB の直前に新しいプロセスの PCB を挿入する（17 行）。実行可能列の最後には、常時 Idle プロセスの PCB が置かれている（図 3.8 参照）。Idle の `enice` は最大値に設定されているので 15 行のループは必ず正常に終了する。

現在の実装では、`enice` はプロセス生成時に `nice` と同じ値に設定され、その後は変化しない。TacOS は静的優先度を用いる優先度順スケジューリング方式を用いていることになる。将来、`enice` の値を動的に変化させるように変更すれば、動的優先度方式になる。

`setPri()` 関数は PSW の割込み許可フラグを操作するために使用している。詳しくは「[5.5.6 setPri\(\) 関数](#)」で説明する。

第 5 章

プロセス同期

これまで見てきたように、複数のプロセス（スレッド）が並行して実行される。複数の並行して実行されるプロセス（スレッド）が、決して競合することなく、必要に応じて協調して動作するために、プロセス（スレッド）間で同期をとる必要がある。この章ではプロセス（スレッド）間の同期について勉強する。

5.1 競合 (Race Condition, Competition)

複数のプロセス（スレッド）が資源を共有して処理を進めることがある。ここで言う資源とは「スレッド間で共有する変数」、「プロセス間で共有するメモリ」、「カーネル内部のデータ構造」、「ファイル」、「入出力装置」等が考えられる。共有する資源をプロセス（スレッド）がアクセスする時、きちんとした取り決めが無いと誤った結果になる場合がある。

例えば、銀行口座を管理する架空の例を考えよう。一つのプロセス内で、入金を処理するスレッドと、引き落としを処理するスレッドが並行して実行されているとする。図 5.1 にこのプロセスの処理内容の一部を TeC 風のアセンブリ言語で示す。

ほぼ同時に、プロセスが給料 3 万円の振込と、カード料金 2 万円の引き落としを受信した場合を考えてみよう。二つのスレッドが競って `account` 変数の更新をすることになる。処理前は `account` 変数に口座の残高 10 万円が記録されていたとする。

1. (1) → (2) → (3) → (a) → (b) → (c) の順で実行された場合
`account` 変数の値は 11 万円になり正しい結果になる。
2. (a) → (b) → (c) → (1) → (2) → (3) の順で実行された場合
`account` 変数の値は 11 万円になり正しい結果になる。
3. (1) → (2) → (a) → (b) → (c) → (3) の順で実行された場合
 入金管理スレッドが途中で preemption し、引き落としスレッドが実行された後、入金管理スレッドが再開された場合である。`account` 変数の値は 13 万円になる。
4. (1) → (a) → (2) → (b) → (3) → (c) の順で実行された場合
 二つの CPU が並列にスレッドを実行した場合である。`account` 変数の値は 8 万円になる。

以上のように、スレッドの実行順序等により計算結果が間違ってしまうことがある。これは資源の利

```

// スレッド間の共有変数
receipt DS      1    // 入金(3万円)
payment DS      1    // 引き落とし(2万円)
account DS      1    // 残高(10万円)

// 入金管理スレッド                                // 引き落とし管理スレッド

// 会社から給料(3万円)を受領し                      // カード会社から引き落としを
// receipt に金額を格納した.                         // 受信し payment に金額を格納した.

// 口座 account に足し込む                          // 口座 account から差し引く
(1) LD      G0,account                           (a) LD      G0,account
(2) ADD     G0,receipt                           (b) SUB     G0,payment
(3) ST      G0,account                           (c) ST      G0,account

// 次の処理に進む                                // 次の処理に進む

```

図 5.1 共有変数をアクセスする二つのスレッド

用について、競合 (Race Condition または Competition) が発生しているからである。

5.2 クリティカルセクション (CriticalSection)

競合が発生するのは、一方のスレッドが自分の CPU レジスタにコピーした account の値を変更し書き戻すまでの間（変更中）に、もう一方のスレッドが account の値を自分の CPU レジスタにコピーすることが原因である。「変更中」の共有変数に他のスレッドがアクセスすることを禁止する必要がある。他のスレッドが共有変数にアクセスすることが許されないプログラムの区間をクリティカルセクション (CriticalSection)，または、クリティカルリージョン (Critical Region) と呼ぶ。

図 5.1 の例で「(1) から (3)」と「(a) から (c)」は account 変数のクリティカルセクションであり、この区間をどれかのスレッドが実行している間は、他のスレッドが account 変数にアクセスしてはならない。クリティカルセクションの競合問題を効率よく解決するためには、次の三つの条件を満たす必要がある。

1. 二つ以上のプロセス（スレッド）が同時にクリティカルセクションに入らない。
2. クリティカルセクションに入っているプロセス（スレッド）がない時は、待たされることなくクリティカルセクションに入ることができる。
3. クリティカルセクションに入るために永遠に待たされることがない。

5.3 相互排除 (Mutual Exclusion)

複数のプロセス（スレッド）が同時にクリティカルセクションに入らないように制御することである。排他制御または相互排他とも呼ばれる。相互排除を達成するために、プロセス（スレッド）は、ク

// 口座 account に足し込む		// 口座 account から差し引く	
DI // Entry Section		DI // Entry Section	
(1) LD G0,account	(a) LD G0,account		
(2) ADD G0,receipt	(b) SUB G0,payment		
(3) ST G0,account	(c) ST G0,account		
EI // Exit Section	EI // Exit Section		

図 5.2 割込み禁止による相互排除

クリティカルセクションに入る際に権利を得る手続きを行う。これを行うプログラムの部分をエントリーセクション (**Entry Section**) と呼ぶ。クリティカルセクションを出る際に権利を返却する手続きを行う。これを行うプログラムの部分をエグジットセクション (**Exit Section**) と呼ぶ。

5.3.1 割込み禁止

シングルプロセッサ (CPU が一つしかない) システムでは、クリティカルセクションを実行するとき割込みを禁止することで目的を達成できる。図 5.2 に図 5.1 を改良したプログラムを示す。

エントリーセクションで DI (Disable Interrupt) 命令を実行し割込みを禁止する。エグジットセクションで EI (Enable Interrupt) 命令を実行し割込みを許可する。クリティカルセクションでは、CPU が割込みを受けない^{*1}のでプリエンプションは発生しない。クリティカルセクションの終わりまで CPU は連続して命令を実行する。また、CPU が一つしかないので他の CPU が account 変数をアクセスすることもない。よって、account 変数の変更中に他のプロセス (スレッド) が account 変数をアクセスすることはない。

この方法は簡単に相互排除を行うことができるが、割込み禁止時間が長くならないように注意する必要がある。割込み禁止が長くなると、タイマーからの割込みを取りこぼし時計が正確に進まなくなったり、入出力装置の制御が間に合わなくなるなどの弊害が生じる^{*2}。また、DI 命令、EI 命令は特権命令なので、カーネル内だけで使用できる手法である。

5.3.2 専用命令を用いる方式

マルチプロセッサ (CPU が複数ある) システムでは、割込み禁止による方法では目的を達成することができない。クリティカルセクションでプリエンプションが発生しなくとも、他の CPU によって実行されるプロセス (スレッド) がクリティカルセクションに入る可能性があるからである。

マルチプロセッサシステムとは、図 2.1 に示したメモリを共有する SMP システムのことである。複数の CPU によるメモリのアクセスはハードウェアにより順序付けされる。同じメモリアドレスへのアクセスが競合し、どちらの CPU が書き込んだ値とも異なる値になることはない。順序付けの結果、後になった書き込みの結果がメモリに残る。また機械語命令は、一部の例外を除いて、途中で割込まれることはない。このようなシステムでは、以下の機械語命令を相互排除の目的に使用できる。

- TS (Test and Set) 命令

^{*1} 再度、割込みが許可されるまで保留になる。プリエンプションはクリティカルセクションを出るまで遅延する。

^{*2} 割込み禁止期間に同じ割込みが複数回発生した場合、割込み許可になったとき割込みの種類につき一度だけ割込みが発生する。ハードウェアに、保留になった割込みのカウンタはない。

```

// エントリーセクション
L1    DI          // クリティカルセクションでプリエンプションしないように
      TS      GO, FLG // ゼロを取得できるプロセス(スレッド)は一時には一つだけ
      JZ      L2      // ゼロを取得できた場合だけクリティカルセクションに入れる
      EI          // ビジーウェイティングの間はプリエンプションのチャンスを作る
      JMP     L1      // クリティカルセクションに入れないとビジーウェイティング

// クリティカルセクション
L2    ...

// エグジットセクション
LD      GO, #0
ST      GO, FLG // フラグのクリアは普通の ST 命令で OK
EI          // クリティカルセクション終了, プリエンプションしても良い

// 非クリティカルセクション
...
...

// メモリ上に置いたフラグ(CPU のフラグと混同しないこと)
FLG    DC      0      // 初期値ゼロ(TS 命令により 1 に書き換えられる)

```

図 5.3 TS 命令の使用例

TS 命令は「(1) メモリの値を CPU レジスタにロード」し、「(2) 1 を同じメモリアドレスに書き込む」命令である。この二つを他の CPU のメモリアクセスに割込まれることなく、アトミック (**atomic**) に実行する。TS 命令 (TS R,M) の動作は、例えば次のようになる。

1. バスをロックする
2. $R \leftarrow [M]$
3. $\text{if } (R==0) \quad Zero \leftarrow 1; \text{ else } Zero \leftarrow 0;$
4. $[M] \leftarrow 1$
5. バスのロックを解除する

TS 命令は、他の CPU がメモリをアクセスしないように、まずバスをロックする。次に、メモリの指定番地から値を CPU レジスタにロードする。また、レジスタの値によって CPU の Zero フラグの値を決める。更に、メモリの指定番地に「1」をストアする。最後にバスのロックを解除する。ロードとストアで合計二回のメモリアクセスがあるが、バスがロックされているので、TS 命令の実行途中に他の CPU がメモリをアクセスすることはない。図 5.3 に TS 命令の使用例を示す。JZ 命令は Zero フラグが「1」の場合のみジャンプする。この例のように、クリティカルセクションに入れるとなるまでループで待つ方式をビジーウェイティング (**Busy Waiting**) と呼ぶ。

メモリのクリアは通常の ST 命令ができる^{*3}。TS 命令を用いる場合もクリティカルセクションは

^{*3} 通常の命令もメモリアクセスする度にバスをロックしている。

```

// エントリーセクション
L1    DI          // クリティカルセクションでプリエンプションしないように
      LD  GO, #1 // フラグに書き込む値
      SW  GO, FLG // ゼロを取得できるプロセス(スレッド)は一時には一つだけ
      CMP GO, #0 // ゼロを取得できたかテスト
      JZ   L2        // ゼロを取得できた場合だけクリティカルセクションに入る
      EI           // ビジーウェイティングの間はプリエンプションのチャンスを作る
      JMP  L1        // クリティカルセクションに入れないとビジーウェイティング

// クリティカルセクション
L2    ...

// エグジットセクション
      LD  GO, #0
      ST  GO, FLG // フラグのクリアは普通の ST 命令で OK
      EI           // クリティカルセクション終了, プリエンプションしても良い

// 非クリティカルセクション
      ...

// メモリ上に置いたフラグ(CPU のフラグと混同しないこと)
FLG  DC  0       // 初期値ゼロ(SW 命令により 1 に書き換えられる)

```

図 5.4 SW 命令の使用例

割込み禁止で実行する必要がある。クリティカルセクションでのプリエンプションを避けるためである。もしも、優先度の低いプロセス（スレッド）がクリティカルセクション内でプリエンプションすると、優先度の高いプロセス（スレッド）がエントリーセクションでビジーウェイティングを始めデッドロックに陥る可能性があるからである。この方式も、特権命令 DI, EI を使用するのでカーネル内でしか利用できない。

- **SW (Swap)** 命令

SW (Swap) 命令も SMP システムでの相互排除に使用できる。「SW R, M」は以下をアトミック (atomic) に実行する。

1. バスをロックする
2. $T \leftarrow [M]$
3. $[M] \leftarrow R$
4. $R \leftarrow T$
5. バスのロックを解除する

ここで T は CPU 内部の一時的なレジスタ (T レジスタの存在はプログラムから見えない) である。図 5.4 に SW 命令の使用例を示す。使用例は TS 命令のものと似ているので解説は省略する。

- **CAS (Compare And Swap)** 命令

<code>// 口座 account に足し込む</code>			
L1	LD	G0,account	LD
	LD	G1,G0	LD
	ADD	G1,receipt	SUB
	CAS	G0,G1,account	CAS
	JNZ	L1	JNZ
<code>// 口座 account から差し引く</code>			
		G0,account	G0,account
		G1,G0	G1,payment
			CAS
			G0,G1,account
			L2

図 5.5 CAS 命令を用いた口座管理プログラムの例

CAS (Compare And Swap) 命令も SMP システムでの相互排除に使用できる。例えば「CAS R0, R1, M」は、以下をアトミック (**atomic**) に実行する。

1. バスをロックする
2. $T \leftarrow [M]$
3. $\text{if } (T == R0) \{ [M] \leftarrow R1; Zero \leftarrow 1; \} \text{ else } \{ R0 \leftarrow T; Zero \leftarrow 0; \}$
4. バスのロックを解除する

CAS 命令を用いたエントリーセクション、エグジットセクションの作り方も、TS 命令と同様なのでここでは使用例を省略する。CAS 命令を用いると共有資源にロックを掛けない、ロックフリー (**Lock-free**) なアルゴリズムを実現できる。前出の銀行口座を管理する架空のプロセス（図 5.1）を CAS 命令を用いて書き換えた例を図 5.5 に示す。

処理開始時の `account` の値を G1 に保存しておく。計算結果を格納する際に、処理開始から `account` の値が変化していないことを確認してから書き込む。以前の例では、他のプロセスが共有資源にアクセスしないように、何らかのロックをかけていた。この方式はロックを掛けずに「結果を書き込む時点で判断」している。

5.3.3 フラグを用いる方式

アルゴリズムを工夫しソフトウェアだけで相互排他を実現する方式である。中でも 1981 年に G. L. Peterson が発表した **Peterson のアルゴリズム (Peterson's solution)** が有名なので紹介する。図 5.6 に Java 風の言語で書いた例を示す。

このアルゴリズムの特徴は次の通りである。

1. マルチプロセッサシステムでも使用できる。
2. 2 プロセス (スレッド) 以上に拡張可能だが複雑になる。
3. 最近のプロセッサと相性が悪い。 (out-of-order 実行)

5.4 セマフォ (Semaphore)

これまでに紹介してきた相互排除は、主にビギュエイティングを用いるものであり、待っている間も CPU を使用し続ける。また、割込み禁止にする必要があるのでカーネル内でしか使用できない。これらは、カーネル内で短時間で終わる相互排除のために適しているが、長時間に渡る場合やユーザプログラムが直接使用する場合には適さない。

```

// スレッド間の共有変数
boolean flag[] = {false, false}; // クリティカルセクションに入りたい
int turn = 0; // 後でやってきたのはどちら

// スレッド 0 // スレッド 1
...
...

// エントリーセクション // エントリーセクション
flag[0] = true;
turn = 0;
while (turn==0 && flag[1]==true)
    ; // ビジーウェイティング

// クリティカルセクション // クリティカルセクション
...
...

// エグジットセクション // エグジットセクション
flag[0] = false;
flag[1] = false;

// 非クリティカルセクション // 非クリティカルセクション
...
...

```

図 5.6 Peterson のアルゴリズム

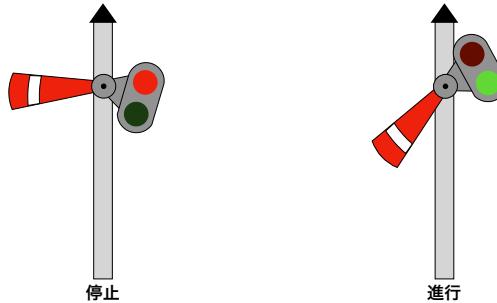


図 5.7 腕木式信号機

そこで、オペレーティングシステムが提供するより洗練されたプロセス同期機構であるセマフォを紹介する。なお、これまでに紹介してきた相互排除は、セマフォを実現するためにも使用される。

5.4.1 概要

セマフォ (Semaphore : 腕木式信号機) は、1965 年に E. W. Dijkstra が提案したデータ型^{*4}である。語源となった腕木式信号機は、鉄道で使用される図 5.7 のような信号機である。

セマフォ型の変数は内部にカウンタ^{*5}を持ち、また、プロセスの待ち行列を作ることができる。セマ

^{*4} C 言語なら構造体を用いてセマフォ型を宣言する。typedef struct { ... } Semaphore;

^{*5} 腕木信号機の進行・停止のように二つの状態しか取らないものをバイナリセマフォと呼ぶ。ここで取り上げるカウンタを

```
void P(Semaphore S) {
    if (S > 0) {
        S--;
    } else {
        プロセスを待ち (Waiting) 状態にする;
        プロセスを S の待ち行列に追加する;
    }
}
```

(a) P 操作

```
void V(Semaphore S) {
    if (S の待ち行列は空) {
        S++;
    } else {
        一つのプロセスを待ち行列から取り出す;
        そのプロセスを実行可能 (Ready) 状態にする;
    }
}
```

(b) V 操作

図 5.8 セマフォのアルゴリズム

フォ型 (Semaphore) の変数には、**P** 操作 (*Proberen:try*) と **V** 操作 (*Verhogen:raise*) を行うことができる。カーネルは P 操作と V 操作を、ユーザプロセスにシステムコールとして提供したり、カーネル内部のサービスモジュールやデバイスドライバにサブルーチンとして提供したりする。セマフォはプロセス（スレッド）の状態を待ち (Waiting) 状態に変える。ビージュエイティングでは無いので CPU を無駄遣いすることはない。

P 操作 (P(S)) セマフォ (S) の値が 1 以上ならセマフォの値を 1 減らす。値が 0 ならプロセス（スレッド）を待ち (Waiting) 状態にし、セマフォの待ち行列に追加する。アルゴリズムを C 言語風に記述したものを図 5.8(a) に示す。

V 操作 (V(S)) セマフォ (S) の待ち行列にプロセス（スレッド）がある場合は、それらの一つを起床させる。待っているプロセス（スレッド）が無い場合は、セマフォ (S) の値を 1 増やす。アルゴリズムを C 言語風に記述したものを図 5.8(b) に示す。

5.4.2 相互排除問題の解

初期値が 1 のセマフォを用いて相互排除問題の解を示すことができる。前出の架空の銀行口座管理プロセスの例を、セマフォを用いて解決したものを図 5.9 に示す。

1 行の `account` は相互排除が必要なスレッド間の共有変数である。2 行の `Semaphore` 型の変数 `accSem` が排他制御に使用するセマフォである。`accSem` は 1 で初期化される。クリティカルセクションに入るスレッドは、まず、6 行か 14 行で `accSem` に P 操作を行う。どちらか先にやって来たスレッドが P 操作を行った時点で `accSem` の値が 0 になる。

遅れてやって来たスレッドは `accSem` の値が 0 の間はクリティカルセクションに入ることができない。先のスレッドがクリティカルセクションを出て 8 行か 16 行で `accSem` に V 操作を行ったら、後のスレッドがクリティカルセクションに入ることができる。

5.4.3 生産者と消費者問題 (Producer-Consumer Problem) の解

生産者プログラム（スレッド）はデータを生産し有限な長さのリングバッファ（ring buffer）に書き込む。消費者プログラム（スレッド）はリングバッファからデータを読み出し消費する。この時、満

持つものはカウンティングセマフォと呼ぶ。カウンタの値は 0 以上の整数値である。

```

1 int account; // スレッド間の共有変数(残高)
2 Semaphore accSem = 1; // 初期値 1 のセマフォ accSem (account のロック用)
3 void receiveThread() { // 入金管理スレッド
4   for ( ; ; ) { // 入金管理スレッドは以下を繰り返す
5     int receipt = receiveMoney(); // ネットワークから入金を受信する
6     P( &accSem ); // account 変数をロックするための P 操作
7     account = account + receipt; // account 変数を変更する (クリティカルセクション)
8     V( &accSem ); // account 変数をロック解除するための V 操作
9   }
10 }
11 void payThread() { // 引落し管理スレッド
12   for ( ; ; ) { // 引落し管理スレッドは以下を繰り返す
13     int payment = payMoney(); // ネットワークから支払い額を受信する
14     P( &accSem ); // account 変数をロックするための P 操作
15     account = account - payment; // account 変数を変更する (クリティカルセクション)
16     V( &accSem ); // account 変数をロック解除するための V 操作
17   }
18 }
```

図 5.9 セマフォを用いた相互排除問題の解

杯のリングバッファに更に書き込んだり、空のリングバッファからデータを読み出したりしないように、プログラム（スレッド）間で歩調を合わせる（同期する）必要がある。セマフォを用いた解を図 5.10 に示す。

リングバッファとセマフォ 1 行の `buffer` は大きさ N のリングバッファである。型は応用によって決まるので、リングバッファの型は仮に `Data` 型としている。2 行の `emptySem` はリングバッファの空きスロット数を表すセマフォである。最初は全てのスロットが空きなので初期値は N である。3 行の `fullSem` はリングバッファの使用中スロット数を表すセマフォである。最初は全てのスロットが空きで、使用中のスロットは無いので、初期値は 0 している。

生産者スレッド 4 行から始まる `producerThread` が、データを生産しリングバッファに書き込むスレッドである。5 行の変数 `in` はリングバッファの次回書き込み位置を表すローカル変数である。 $0, 1, 2, \dots, N-1, 0, 1, 2, \dots$ の順に値が変化する。`in` はスレッドのローカル変数なので、相互排除をする必要がない。

`producerThread` は、7 行でデータを作り、8 行で空きスロット数が 1 以上なら `emptySem` の値を減らして、9 行でデータをリングバッファに書き込む。10 行で `in` の値を更新しているが、`in` はローカル変数なので 11 行より後でも良い。11 行で使用中スロット数 `fullSem` の値を増加させる。

消費者スレッド 14 行から始まる `consumerThread` は、データをリングバッファから読み出して消費するスレッドである。15 行の変数 `out` はリングバッファの次回読み出し位置を表すローカル変数である。`out` もスレッドのローカル変数なので、相互排除をする必要がない。

```

1 Data      buffer[N];           // スレッド間で共有するリングバッファ
2 Semaphore emptySem = N;       // リングバッファの空きスロット数を表すセマフォ
3 Semaphore fullSem = 0;         // リングバッファの使用中スロット数を表すセマフォ
4 void producerThread() {
5     int in = 0;                // リングバッファの次回格納位置
6     for ( ; ; ) {             // 生産者スレッドは以下を繰り返す
7         Data d = produce();   // 新しいデータを作る
8         P( &emptySem );        // リングバッファの空き数をデクリメント
9         buffer[ in ] = d;      // リングバッファにデータを格納
10        in = (in + 1) % N;    // 次回格納位置を更新
11        V( &fullSem );        // リングバッファのデータ数をインクリメント
12    }
13 }
14 void consumerThread() {        // 消費者スレッド
15     int out = 0;               // リングバッファの次回取り出し位置
16     for ( ; ; ) {             // 消費者スレッドは以下を繰り返す
17         P( &fullSem );        // リングバッファのデータ数をデクリメント
18         Data d = buffer[ out ]; // リングバッファからデータを取り出す
19         out = (out + 1) % N;   // 次回取り出し位置を更新
20         V( &emptySem );        // リングバッファの空き数をインクリメント
21         consume( d );        // データを使用する
22     }
23 }
```

図 5.10 セマフォを用いた生産者消費者問題の解

`consumerThread` は、17 行で空きスロット数が 1 以上なら `fullSem` の値を減らして、18 行でデータをリングバッファから読み出す。19 行で `out` の値を更新する。20 行で空きスロット数 `emptySem` の値を増加させる。21 行で読み出したデータを使用する。

5.4.4 複数生産者と複数消費者問題（Producer-Consumer Problem）の解

前の問題で、関数 `producerThread()`, `consumerThread()` それぞれについて、複数のスレッドが存在する場合を考える。バッファに関する同期の他に、書き込み位置 (`in`)、取り出し位置 (`out`) に関する排他制御が必要になる。解を図 5.11 に示す。

リングバッファとセマフォ 1 行から 3 行に変更はない。

生産者スレッド 次回書き込み位置を表す `in` 変数を複数の `producerThread` で共有する必要がある。`in` 変数の宣言を 5 行に移動し、スレッド間の共有変数に変更した。また、`in` 変数を `producerThread` 間で相互排除するためのセマフォ `inSem` を 6 行に追加した。

`producerThread` では、`in` 変数の参照や書き換えを行う 12 行と 13 行が `in` 変数に関するクリティカルセクションである。11 行と 14 行に `inSem` を用いた相互排除機構を追加した。

消費者スレッド 次回読み出し位置を表す `out` 変数について、生産者スレッドと同様な相互排除

```

1 Data      buffer[N];           // スレッド間で共有するリングバッファ
2 Semaphore emptySem = N;       // リングバッファの空きスロット数を表すセマフォ
3 Semaphore fullSem = 0;        // リングバッファの使用中スロット数を表すセマフォ
4
5 int       in = 0;             // リングバッファの次回格納位置
6 Semaphore inSem = 1;          // in の排他制御用セマフォ
7 void producerThread() {
8     for ( ; ; ) {            // 生産者スレッド(複数のスレッドで並列実行する)
9         Data d = produce();   // 生産者スレッドは以下を繰り返す
10        P( &emptySem );      // 新しいデータを作る
11        P( &inSem );          // リングバッファの空き数をデクリメント
12        buffer[ in ] = d;     // in にロックを掛ける
13        in = (in + 1) % N;    // 次回格納位置を更新
14        V( &inSem );          // in のロックを外す
15        V( &fullSem );        // リングバッファのデータ数をインクリメント
16    }
17 }
18
19 int out = 0;                // リングバッファの次回取り出し位置
20 Semaphore outSem = 1;         // out の排他制御用セマフォ
21 void consumerThread() {
22     for ( ; ; ) {            // 消費者スレッド(複数のスレッドで並列実行する)
23         P( &fullSem );        // 消費者スレッドは以下を繰り返す
24         P( &outSem );          // リングバッファのデータ数をデクリメント
25         Data d = buffer[ out ]; // リングバッファからデータを取り出す
26         out = (out + 1) % N;   // 次回取り出し位置を更新
27         V( &outSem );          // out のロックを外す
28         V( &emptySem );        // リングバッファの空き数をインクリメント
29         consume( d );         // データを使用する
30     }
31 }
```

図 5.11 セマフォを用いた複数生産者・複数消費者問題の解

機構を追加してある。

5.4.5 リーダ・ライタ問題 (Readers-Writers Problem) の解

共有データに対して、読み出しだけするリーダプロセス（スレッド）と、読み出し書き込みの両方を行うライタプロセス（スレッド）の二種類がある場合に、単に資源をロックするより並行性（**concurrency**）を高くすることができます。リーダプロセス（スレッド）は、値を読み出すだけなので、他のリーダプロセス（スレッド）と同時に共有データをアクセしても良い。ライタプロセス（スレッド）は、値を書換えるので、他のライタともリーダとも同時に共有データをアクセスすることは許されない。セマフォによる解を図 5.12 に示す。

```

1 Data      records;           // 共有するデータ
2 Semaphore rwSem = 1;        // リーダとライタの排他用セマフォ
3
4 void writerThread() {         // ライタスレッド(複数のスレッドで並列実行する)
5   for ( ; ; ) {             // ライタスレッドは以下を繰り返す
6     Data d = produce();     // 新しいデータを作る
7     P( &rwSem );            // 共有データにロックを掛ける
8     writeRecords( d );      // データを書き換える
9     V( &rwSem );            // 共有データのロックを外す
10 }
11 }
12
13 int      cnt = 0;           // リーダ間の共有変数(読み出し中のリーダ数)
14 Semaphore cntSem = 1;       // cnt の排他制御用セマフォ
15
16 void readerThread() {        // リーダスレッド(複数のスレッドで並列実行する)
17   for ( ; ; ) {             // リーダスレッドは以下を繰り返す
18     P( &cntSem );          // cnt にロックを掛ける
19     if ( cnt == 0 ) P( &rwSem ); // 自分が最初のリーダなら、代表してロックする
20     cnt = cnt + 1;           // cnt をインクリメント
21     V( &cntSem );          // cnt のロックを外す
22     Data d = readRecords(); // データを読みだす
23     P( &cntSem );          // cnt にロックを掛ける
24     cnt = cnt - 1;           // cnt をデクリメント
25     if ( cnt == 0 ) V( &rwSem ); // 自分が最後のリーダなら、代表してロックを外す
26     V( &cntSem );          // cnt のロックを外す
27     consume( d );          // データを使用する
28   }
29 }
```

図 5.12 セマフォを用いたリーダ・ライタ問題の解

共有データとセマフォ 1行の `records` が共有データである。2行の `rwSem` は共有データの相互排除用のセマフォである。これらは、全てのスレッドに関係がある。

ライタスレッド 4行の `writerThread` は共有データを書き換えることがあるスレッドである。書き換え途中に他のスレッドが共有データをアクセスすることを禁止するために、`writerThread()` は7行で `rwSem` にロックを掛ける。9行でロックを解除するまで、他のライタもリーダも同時に共有データにアクセスすることはできない。このようなロックを排他ロック（exclusive lock）と呼ぶ。

リーダスレッド 16行の `readerThread` は共有資源を読むことだけする。書き換え途中の不完全なデータを読み出さないように、`writerThread` と相互排除を行う必要がある。しかし、書き換え途中以外なら、他のリーダスレッドと同時にデータを読んでも構わない。

```
#define SEM_MAX 30          // セマフォは最大 30 個

struct Sem {              // セマフォを表す構造体
    int cnt;               // カウンタ
    PCB queue;             // 待ち行列
};
```

図 5.13 TacOS のセマフォ構造体

13 行の `cnt` 変数はリーダスレッド間で共有される。14 行の `cntSem` セマフォは `cnt` 変数の相互排除用である。リーダスレッドはこれらを使用し、`records` 共有データを読み出し中のリーダスレッドの数を管理する。19 行と 20 行、24 行と 25 行の二箇所が、`cnt` 変数に関するクリティカルセクションである。

19 行では最初に読み出しを始めるリーダを判断し、最初のリーダだけが代表して `rwSem` にロックを掛ける。二番目にやって来たリーダはロックを掛けないのでリーダ相互は排他されない。しかし、排他ロックを用いるライタとは相互排除される。このようなロックを共有ロック (**shred lock**) と呼ぶ。25 行で最後に読み出しを終えるリーダを判断し、最後のリーダだけが代表して `rwSem` のロックを解除する。

リーダ・ライタ問題は、共有ロックと排他ロックを使用する問題の例になっている。共有ロックと排他ロックの考え方は、ここに示したスレッド間の共有変数の管理だけでなく様々な場面で使用される。例えば UNIX のシステムコール `flock` は、引数に定数 `LOCK_SH` を渡すと共有ロックを、定数 `LOCK_EX` を渡すと排他ロックをファイルに掛ける。

また、UNIX の `open` システムコールは、引数に `O_SHLOCK` フラグを指定すると共有ロックを、引数に `O_EXLOCK` フラグを指定すると排他ロックを、ファイルのオープン時に自動的に掛ける。

5.5 TacOS のセマフォ

TacOS ではプロセス同期の基本機構としてセマフォを用いる。セマフォ機構は TacOS のマイクロカーネルが提供する。

5.5.1 データ構造

TacOS のセマフォは図 5.13 に示す構造体である^{*6}。

図 5.14 に TacOS のセマフォ関連データの構造を示す。`semTbl` はセマフォの一覧である。システム起動時に `SEM_MAX` 個（30 個）のセマフォを準備し `semTbl` に登録する。`semInUse` はセマフォが使用中かどうかを記録する論理型の配列である。セマフォが必要になった時に、一覧の中から空きセマフォを選んで使用する。セマフォは一覧のインデックス（セマフォ番号）で識別するので、P 操作や V 操作を行う関数の引数がセマフォ番号になる。

セマフォ構造体（`Sem` 構造体型）は、セマフォの値（`cnt`）とプロセスの待ち行列（`queue`）を持って

^{*6} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/process.hmm> の一部である。

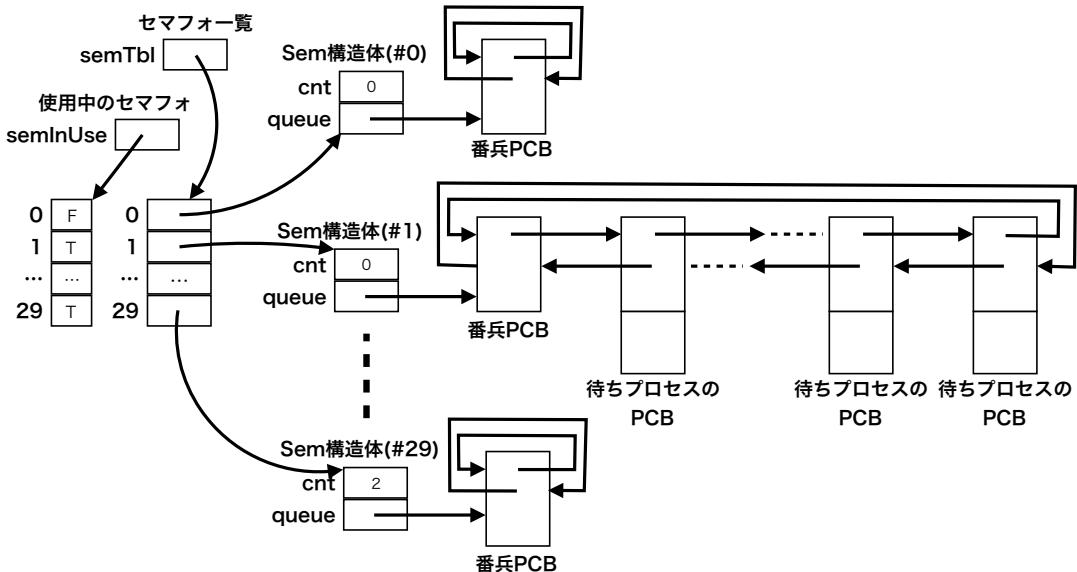


図 5.14 TacOS のセマフォ関連データ構造

いる。システム起動時に番兵 `PCB` を使用した空の重連結環状リストが登録される。プロセスの待ち行列の作り方は、図 3.8 に示した実行可能列と同様である。次に、図 5.14 で表している三つのセマフォについて説明する。

Sem 構造体 (#0) セマフォ一覧 (`semTbl`) の第 0 行に登録されている。Sem 構造体 (#0) は使用されていない Sem 構造体を表している。`semInUse` の対応する要素は `False` になっている。

Sem 構造体 (#1) 値が 0 の時に複数のプロセスが `P` 操作を行った状態である。使用中なので `semInUse` の対応する要素は `True` になっている。`P` 操作を行いブロックしたプロセスがセマフォの待ち行列に入っている。プロセスは待ち行列の最後（図では右）に追加され、待ち行列の先頭（図では左）から取り出される。同じセマフォについて、プロセスは FCFS のスケジューリングが適用される。

Sem 構造体 (#29) `V` 操作の結果、値が 2 になっている状態を表している。使用中なので `semInUse` の対応する要素は `True` になっている。値が 1 以上の時は、待ち行列が必ず空になる。

5.5.2 使用例

図 5.15 に TacOS でのセマフォの架空の使用例を示す。これは、図 5.9 の例を TacOS 用に書き換えたものである。

共有変数と相互排除用のセマフォ 以前の例ではセマフォを `Semaphore` 型の変数として扱っていた。今回の例では、セマフォはカーネル内部に存在し、使用者はセマフォを番号で指定するようになっている。そのため 3 行は、セマフォ変数の宣言から、番号を記憶する整数型変数の宣言に変更された。

使用するセマフォの割当て セマフォはカーネル内部で図 5.14 に示したように管理されている。4

```

1 #include <kernel.h>
2 int account; // スレッド間の共有変数(残高)
3 int accSem; // account のロック用セマフォの番号
4 void initProc() {
5   accSem = newSem(1); // 初期値 1 のセマフォを確保する
6 }
7 void receiveThread() { // 入金管理スレッド
8   for ( ; ; ) { // 入金管理スレッドは以下を繰り返す
9     int receipt = receiveMoney(); // ネットワークから入金を受信する
10    semP( accSem ); // account 変数をロックするための P 操作
11    account = account + receipt; // account 変数を変更する(クリティカルセクション)
12    semV( accSem ); // account 変数をロック解除するための V 操作
13  }
14 }
15 void payThread() { // 引落し管理スレッド
16   for ( ; ; ) { // 引落し管理スレッドは以下を繰り返す
17     int payment = payMoney(); // ネットワークから入金を受信する
18     semP( accSem ); // account 変数をロックするための P 操作
19     account = account - payment; // account 変数を変更する(クリティカルセクション)
20     semV( accSem ); // account 変数をロック解除するための V 操作
21   }
22 }
```

図 5.15 TacOS でのセマフォの架空の使用例

行のプロセスの初期化ルーチン `initProc()` 中で、カーネルが提供する関数 `newSem()` を用いてセマフォの割当てを受ける。`newSem()` 関数の引数はセマフォの初期値である。

P 操作と V 操作 TacOS で使用できる P 操作関数は `semP()`、V 操作関数は `semV()` である。10 行、12 行、18 行、20 行のようにセマフォ番号を引数に使用する。

5.5.3 割当

図 5.16 に TacOS カーネル内のセマフォ割当と解放ルーチンを示す^{*7}。

データ構造 1 行の `semTbl`、2 行の `semInUse` は、図 5.14 に描かれている「セマフォ一覧」と「使用中のセマフォ」のことである。`semTbl` は TacOS の起動時に「Sem 構造体」や「番兵 PCB」で初期化される。

割込み禁止による相互排除 5 行の `newSem()` 関数が `semTbl` から未使用のセマフォを探す。`newSem()` 関数や後述の `semP()`、`semV()` 関数は、複数のプロセスから並列に呼び出され `semTbl` や `semInUse` をアクセスする。これらのデータ構造はプロセス間の共有データである。`newSem()` 関数の内部はこれら共有データのクリティカルセクションに当たるので相互排他が

^{*7} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm> の一部である。

```

1 Sem[] semTbl=array(SEM_MAX); // セマフォの一覧表
2 boolean[] semInUse=array(SEM_MAX); // どれが使用中か(falseで初期化)
3
4 // セマフォの割当て
5 public int newSem(int init) {
6     int r = setPri(DI|KERN); // 割り込み禁止、カーネル
7     for (int i=0; i<SEM_MAX; i=i+1) { // 全てのセマフォについて
8         if (!semInUse[i]) { // 未使用のものを見つけたら
9             semInUse[i] = true; // 使用中に変更し
10            semTbl[i].cnt = init; // カウンタを初期化し
11            setPri(r); // 割込み状態を復元し
12            return i; // セマフォ番号を返す
13        }
14    }
15    panic("newSem"); // 未使用が見つからなかった
16    return -1; // ここは実行されない
17 }
18
19 // セマフォの解放
20 public void freeSem(int s) {
21     semInUse[s] = false; // 未使用に変更(アトミック)
22 }
```

図 5.16 TacOS のセマフォ割当て解放ルーチン

必要である。TaC はシングルプロセッサシステムなので、[5.3.1](#) で紹介した「割込み禁止による相互排除」を行う。

6 行では、現在のフラグ^{*8}の値を *r* に保存した後、「割込み禁止 (DI)」にしている。*setPri()* 関数はフラグの値を読み出し、同時に引数値をフラグにセットするアセンブリ言語ルーチンである^{*9}。*newSem()* 関数はカーネルモードで呼出すので、実行モードが変化しないように「カーネルモード (KERN)」も指定している。

7 行からのループで使用されていないセマフォを探す。割込み禁止で実行するので探索の途中でプリエンプションは発生しない。未使用のセマフォが見つかったら 12 行でその番号を返す。

クリティカルセクションが終わるので、通常は割込みを許可するが、*newSem()* 関数を呼出す前から割込み禁止だった場合もある。11 行では 6 行で保存した *r* を用いてフラグの状態を復旧している。もともと *newSem()* が割込み許可状態で呼出された場合だけ割込み許可状態に戻る。

エラー処理 未使用のセマフォが見つからなかった場合は、15 行で *panic()* 関数を呼出す。現在

^{*8} CPU の PSW のフラグのこと。

^{*9} *setPri()* 関数の詳細は「[5.5.6 setPri\(\) 関数](#)」を参照のこと

```

1 // プロセスキュ (実行可能列やセマフォの待ち行列) で p を削除する
2 void delProc(PCB p) {
3     p.prev.next=p.next;
4     p.next.prev=p.prev;
5 }
6 // セマフォの P 操作
7 public void semP(int sd) {
8     int r = setPri(DI|KERN);           // 割り込み禁止、カーネル
9     if (sd<0 || SEM_MAX<=sd || !semInUse[sd]) // 不正なセマフォ番号
10    panic("semP(%d)", sd);
11
12    Sem s = semTbl[sd];
13    if(s.cnt>0) {                     // カウンタから引けるなら
14        s.cnt = s.cnt - 1;            // カウンタから引く
15    } else {                          // カウンタから引けないなら
16        delProc(curProc);          // 実行可能列から外し
17        curProc.stat = P_WAIT;      // 待ち状態に変更する
18        insProc(s.queue,curProc);   // セマフォの行列に登録
19        yield();                   // CPU を解放し
20    }                                // 他プロセスに切換える
21    setPri(r);                      // 割り込み状態を復元する
22 }

```

図 5.17 TacOS の P 操作ルーチン

の TacOS ではセマフォを使用できるのはカーネルとサーバプロセスだけなので、セマフォが不足するようならオペレーティングシステムのバグである。`panic()` 関数はエラーメッセージを表示した後、CPU を停止する。`panic()` 関数は戻ってこないので 16 行は実行されない。
解放ルーチン 20 行の `freeSem()` は割当てられていたセマフォを解放する。共有変数 `semInUse` 配列の書き換えは、单一のストア機械語命令で終了するので割込み禁止にする必要はない^{*10}。

5.5.4 P 操作ルーチン

図 5.17 に TacOS の P 操作ルーチンを示す^{*11}。P 操作ルールーチンは `semP()` 関数のことである。

割込み禁止による相互排除 `semP()` 関数も、`semInUse` や、`semTbl` の配下の `Sem` 構造体、`PCB` 構造体等の共有データをアクセスするので相互排除を必要とする。`semP()` 関数の内部は 8 行と 21 行の `setPri()` 関数を用いて、割り込み禁止による相互排除を行っている。
セマフォ番号からセマフォ構造体への変換 9 行で引数のセマフォ番号が正当なものかチェックしている。不正なものが渡されるようならオペレーティングシステムのバグなので `panic()` 関数を用いてシステムを停止させる。セマフォ番号が正しい場合は、12 行で `semTbl` 配列から目

^{*10} CPU が機械語命令の途中で割込みを受け付けることはない。

^{*11} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm> の一部である。

的のセマフォを見つける。

セマフォ値のデクリメント 13行でセマフォの値を調べ、1以上なら14行で値を1減らす。この場合は21行で割り込み許可フラグを復元してsemP()関数を終了する。

Block(事象待ち) 13行でセマフォの値を調べ、1未満なら16行に進み現在のプロセスをブロック^{*12}する。ブロックの手順は次の通りである。

1. delProc()関数を用いて現在のプロセスを実行可能列から外す。
2. 現在のプロセスの状態を「待ち状態(P_WAIT)」に変更する。
3. 現在のプロセスをセマフォの待ち行列の最後に追加する^{*13}。
4. yield()関数を呼び出しCPUを解放する。後でセマフォがV操作されプロセスが実行可能になったら、yield()関数から実行が再開される。

なお、ここで使用しているdelProc()は図5.17の2行目で、insProc()は図4.3で定義されたプロセス行列の操作関数である。yield()関数は図3.9に示したプロセス切換えプログラムである。

5.5.5 V操作ルーチン

図5.18にTacOSのV操作ルーチンを示す^{*14}。TacOSのV操作ルーチンはiSemV()とsemV()の二種類がある。iSemV()関数はセマフォにV操作だけ行う。semV()関数はセマフォにV操作を行った後で、プロセス切換えを試みる。semV()関数を用いると、V操作によって実行可能になったプロセスの優先度が現在のプロセスの優先度より高い場合に、プロセスが切り換わる。iSemV()はカーネルや割込みハンドラ等でプリエンプションの発生を避けたい場合に使用する。

割込み禁止による相互排除 iSemV()関数やsemV()関数も相互排除を必要とする。semV()関数は22行と26行のsetPri()関数を用いて、割り込み禁止による相互排除を行っている。iSemV()関数は、呼び出し側で割り込み禁止にして使用する。

セマフォ番号からセマフォ構造体への変換 3行でセマフォ番号の妥当性をチェックしてから、7行でsemTbl配列から目的のセマフォを見つける。

セマフォ値のインクリメント 10行で待ち行列の状態を調べる。番兵PCB(q)と番兵直後のPCB(p)が同じなら待ち行列は空である^{*15}。待ち行列が空の場合は11行でセマフォの値を1増やしfalseを返り値としてiSemv()関数を終了する。

Complete(事象完了) 10行で待ち行列を調べ空でないなら13行に進み、待ち行列の先頭のプロセスを起床させる。先頭のプロセスはComplete(事象完了)^{*16}の状態遷移をする。13行でセマフォの待ち行列から先頭プロセスを外し、14行でプロセスの状態を実行可能(P_RUN)に変更し、15行でスケジューラ(schProc()関数)^{*17}に依頼し実行可能列の適切な位置に挿入する。この場合はtrueを返り値としてiSemv()関数を終了する。

^{*12} プロセスのブロック(Block:事象待ち)については、「3.2 プロセスの状態」を参照のこと。

^{*13} insProc()関数を用いて番兵PCBの直前に挿入する。環状リストで番兵PCBの直前は最後尾のことになる。

^{*14} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm> の一部である。

^{*15} 図5.14の「Sem構造体(#29)」を参照のこと。

^{*16} プロセスのComplete(事象完了)については、「3.2 プロセスの状態」を参照のこと。

^{*17} スケジューラ(schProc()関数)は図4.3で定義されている。

```

1 // ディスパッチを発生しないセマフォの V 操作
2 boolean iSemV(int sd) {
3     if (sd<0 || SEM_MAX<=sd || !semInUse[sd]) {           // 不正なセマフォ番号
4         panic("iSemV(%d)", sd);
5     }
6     boolean ret = false;                                // 起床するプロセスなし
7     Sem s = semTbl[sd];                                // 操作するセマフォ
8     PCB q = s.queue;                                  // 待ち行列の番兵
9     PCB p = q.next;                                   // 待ち行列の先頭プロセス
10    if(p==q) {                                       // 待ちプロセスが無いなら
11        s.cnt = s.cnt + 1;                            // カウンタを足す
12    } else {                                         // 待ちプロセスがあるなら
13        delProc(p);                                 // 待ち行列から外す
14        p.stat = P_RUN;                             // 実行可能に変更
15        schProc(p);                               // 実行可能列に登録
16        ret = true;                                // 起床するプロセスあり
17    }
18    return ret;                                    // 実行可能列に変化があった
19 }
20 // セマフォの V 操作
21 public void semV(int sd) {
22     int r = setPri(DI|KERN);                      // 割り込み禁止、カーネル
23     if (iSemV(sd)) {                            // V 操作し必要なら
24         yield();                                // プロセスを切り替える
25     }
26     setPri(r);                                // 割り込み状態を復元する
27 }
```

図 5.18 TacOS の V 操作ルーチン

プロセスの切換え `semV()` 関数は、V 操作により実行可能列に新しいプロセスが追加された場合 (`iSemv()` 関数が `true` で返った場合) に `yield()` 関数を呼出す。実行可能列に現在のプロセスより優先度の高いものがあった場合、プロセスの切換えが起こる。

TacOS のプロセス同期機構は全てセマフォに基づいて構成される。例えば、メッセージ通信機構もセマフォを利用して構築されている。

5.5.6 `setPri()` 関数

割り込み禁止による相互排除で使用した `setPri()` 関数のソースプログラムを図 5.19 に示す^{*18}。
`setPri()` 関数は CPU の PSW のフラグを参照・操作し、呼出し前の割込み許可状態を保存すると同時に、新しい値に変更する。CPU の PSW のフラグに割込許可ビットがある。

`setPri()` 関数は TaC のアセンブリ言語で記述してある。C--言語から `setPri` という名前で参照さ

*18 <https://github.com/tctsigemura/TacOS/blob/master/os/util/crt0.s> の一部である。

```

1 ;; CPU のフラグの値を返すと同時に新しい値に変更
2 _setPri
3     ld      g0,2,sp ; 引数の値を G0 に取り出す
4     push    g0        ; 新しい状態をスタックに積む
5     ld      g0,flag ; 古いフラグの値を返す準備をする
6     reti            ; reti は FLAG と PC を同時に pop する

```

図 5.19 TacOS のフラグ操作ルーチン

れるためには、アセンブリ言語では `_setPri` というラベルを宣言する必要がある。2行は `setPri()` 関数の入口になるラベルを宣言している。

C--言語プログラムは関数引数をスタックに積んで渡す^{*19}。3行では C--言語が `setPri()` 関数に渡した引数を G0 に読み出している。4行で読み出した値をスタックに積み直す。

5行では現在のフラグ値を G0 にコピーする。C--言語では関数の返り値を G0 レジスタに入れて返すので^{*20}、この値は `setPri()` 関数の返り値になる。6行の `reti` 機械語命令は、スタックからフラグと PC の値を取り出し、`setPri()` 関数を呼出した場所に制御を戻す。この時、4行でスタックに積んだ値がフラグに読み出される。

以上の仕組みで、`setPri()` 関数は引数の値を CPU のフラグにセットすると同時に、以前のフラグ値を呼び出し側に返している。

5.6 まとめ

この章ではプロセス間の同期に関する話題を取り上げた。競合が発生しないように、クリティカルセクションを実行する時は、プロセス間の相互排除をする必要がある。オペレーティングシステムのカーネル内部などで、短時間で終わるクリティカルセクションの相互排除を行う場合は、割込み禁止、専用命令とビジーウェイティングを用いる方法などが使用できる。専用命令として TS 命令、SW 命令、CAS 命令を紹介した。CAS 命令はロックフリーなアルゴリズムを実現するために使用できる。

クリティカルセクションの実行に長い時間がかかる場合は、セマフォなどプロセスの状態遷移を伴うオペレーティングシステムの機能を使用する。セマフォを用いた相互排除問題の解、生産者と消費者問題の解、複数生産者と複数消費者問題の解、リーダ・ライタ問題の解を学んだ。

また、TacOS でセマフォがどのように実現されているかを学んだ。セマフォ機構はマイクロカーネルによって提供される。セマフォはカウンタと PCB の待ち行列を保持する構造体として表現される。P 操作、V 操作などの内部では割込み禁止による相互排除が行われていた。

練習問題

5.1 競合とは何か？

5.2 クリティカルセクションとは何か？

^{*19} C 言語などの言語でも関数に引数を渡す仕組みは同様である

^{*20} C 言語などの言語でも関数値を返す仕組みは同様である

- 5.3 相互排除とは何か？
- 5.4 なぜ割り込みを禁止することで相互排除ができるか？
- 5.5 割り込み禁止による相互排除がマルチプロセッサシステムでは不十分な理由は？
- 5.6 割り込み禁止による相互排除はクリティカルセクションの三つの条件を満たしているか？
- 5.7 CPU が割り込み禁止になっている間に発生した割り込みはどのように扱われるか？
- 5.8 DI 命令, EI 命令が特権命令でなかったら, どのような不都合が生じるか？
- 5.9 シングルプロセッサシステムにおいて, 機械語命令はアトミック (atomic) と言えるか？
- 5.10 マルチプロセッサシステムにおいて, 機械語命令はアトミック (atomic) と言えるか？
- 5.11 TS 命令, SW 命令に共通な特長は何か？
- 5.12 図 5.3 のようなビギュエイティングはシングルプロセッサシステムでも使用できるか？
- 5.13 セマフォを相互排除に使用する手順を説明しなさい。
- 5.14 生産者と消費者の問題において, 二つのセマフォはどのような値に初期化されたか？
二つのセマフォは何の役割を持っていたか？
- 5.15 TaC をマルチプロセッサシステムに進化させた時, 「図 5.17 TacOS の P 操作ルーチン」をどのように改造する必要があるか？(Sem 構造体を変更しない場合)
- 5.16 TaC をマルチプロセッサシステムに進化させた時, 「図 5.17 TacOS の P 操作ルーチン」をどのように改造する必要があるか？(Sem 構造体も変更して良い場合)

第 6 章

プロセス間通信

この章ではプロセス間通信（IPC : Inter-Process Communication）について学ぶ。5 章で学んだ、「生産者と消費者の問題」や「リーダ・ライタ問題」の具体的な解を得るために、プロセス間で情報を共有することが必要である。プロセス間で情報を共有する代表的な機構として、共有メモリとメッセージ通信がある。

複数のプロセスが情報を共有し協調して処理を進めることで、次のようなメリットが期待できる。

- 複数のプロセスが共通の情報へアクセスすることができる。
- 並列処理による処理の高速化ができる可能性がある。
- システムを見通しの良いモジュール化された構造で構築できる。

6.1 共有メモリ

共有メモリは図 6.1 に示すように、プロセス間で同じ物理メモリを共有する方式である。プロセス 1 とプロセス 2 は同じ物理メモリ（共有メモリ）を、それぞれの仮想メモリ空間に貼り付けている。

メモリ管理のハードウェア（Memory Management Unit : MMU）^{*1}を適切に設定することで、複数のプロセスの仮想メモリ空間に同じ物理メモリを貼り付ける。メモリを貼り付ける操作はシステムコールを用いて行う。貼り付けが完了した後は、システムコールを用いることなく情報の共有が可能である。

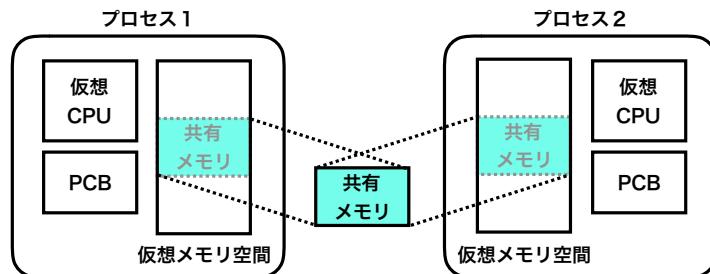


図 6.1 共有メモリ

^{*1} MMU については「メモリ管理」の章で解説する。

共有メモリに関するライブラリとシステムコール

共有メモリなどの識別に使用するキーを生成(ライブラリ)

```
key_t ftok(const char *path, int id);
```

返り値 : 引数から作成されるキー

path : 実際に存在するプロセスにアクセス可能なファイル

id : キーの作成に使用する追加の情報(同じ path から異なるキーを作る)

共有メモリセグメントの作成(システムコール)

```
int shmget(key_t key, size_t size, int flag);
```

返り値 : 共有メモリセグメント ID

key : キー

size : セグメントサイズ(バイト単位)

flag : 作成フラグとモード

共有メモリセグメントをプロセスの仮想アドレス空間に貼り付ける(システムコール)

```
void *shmat(int shmid, void *addr, int flag);
```

返り値 : 共有メモリセグメントを配置したアドレス

shmid : 共有メモリセグメント ID

addr : 貼り付けるアドレス(NULL(0) は, カーネルに任せる)

flag : 貼り付け方法等

共有メモリセグメントをプロセスの仮想アドレス空間から取り除く(システムコール)

```
int shmdt(void *addr);
```

返り値 : 0=正常, -1=エラー

addr : 取り除く共有メモリセグメントのアドレス

共有メモリセグメントの制御(システムコール)

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

返り値 : 0=正常, -1=エラー

shmid : 共有メモリセグメント ID

cmd : 削除(IPC_RMID) 等のコマンド

buf : コマンドのパラメータ

図 6.2 UNIX の共有メモリ関連システムコールとライブラリ関数

が、プロセス間の同期機構は別に準備する必要がある。

6.1.1 UNIX の共有メモリ関連システムコール等

UNIX の共有メモリ関連のシステムコールとライブラリ関数を図 6.2 に紹介する。

ftok() ライブラリ関数 **ftok()** は、**path** と **id** の組合せから、システム内で一意かつ唯一の **key** 値を生成する。

shmget システムコール **key** 値で識別される共有メモリセグメントの ID 返す。 **key** 値で識別される共有メモリセグメントが存在しない場合は、**size** バイトのものを新しく作ることも可能

である。flag の値は共有メモリのアクセス許可ビット（`rwxrwxrwx`）と、IPC_CREAT 等のフラグである。

`shmat` システムコール 共有メモリセグメントを ID (shmId) で指定し、プロセスの仮想アドレス空間に貼り付ける。

`shmdt` システムコール 共有メモリセグメントを ID (shmId) で指定し、プロセスの仮想アドレス空間から取り除く。

`shmctl` システムコール 共有メモリセグメントを ID (shmId) で指定し操作する。共有メモリセグメントの削除等の操作ができる。

6.1.2 UNIX の共有メモリ使用例

共有メモリセグメントを作成し、そこから定期的にデータを読み出し表示するサーバプログラムの例を図 6.3 に示す^{*2}。また、サーバプログラムが作成した共有メモリセグメントにデータを書き込むクライアントプログラムの例を図 6.4 に示す。

サーバプログラムでは、28 行の `printf()` が共有メモリ (data) から文字列を読み出し表示する。文字列が `end` ならプログラムを終了する。クライアントプログラムでは、27 行の `fgets()` が共有メモリ (data) に文字列を書き込む。これらのプログラムでは、共有メモリが普通の文字配列のように `printf()` や `fgets()` に渡されている。共有メモリなので、`fgets()` が書き込んだ内容を `printf()` が読み出すことになる。

実行例は図 6.5 のようになる。図は二つのターミナルを開いて操作した状態を示している。左半分が第一のターミナル、右半分が第二のターミナルである。まず、左のターミナルでサーバプログラム (`ipcUnixShearedMemoryServer`) を起動する。これで共有メモリセグメントが準備された。次に、右のターミナルでクライアントプログラム (`ipcUnixShearedMemoryClient`) を起動する。この状態で右のターミナルに入力した文字列が、クライアントプログラムにより共有メモリに書き込まれる。左のターミナルで実行中のサーバプログラムは、共有メモリの内容を定期的に表示する。

ここに紹介した簡単なプログラムでは、クライアントプロセスがデータを書き換え中に、サーバプロセスがデータを読み出す可能性がある。このようなプログラムを使用してはならない。実際に使用する場合は書き換え中のデータを読み出さないように、セマフォ等^{*3}を用いて相互排除を行う必要がある。原理の確認以外の目的にこのプログラムを使用してはならない。

6.2 メッセージ通信

メッセージ通信は図 6.6 に示すように、システムコールを用いてプロセス間で情報をコピーする方式である。プロセス 1 は `send()` システムコールを用いてプロセス 2 へメッセージを送る。プロセス 2 は `receive()` システムコールを用いてプロセス 1 からメッセージを受取る。

メッセージ通信は、データを送る度にシステムコールを使用するのでオーバーヘッドが大きいが、プロセス間の同期機構としても働く。

^{*2} ここで示すプログラムは macOS 10.13.2 で動作確認してあるが、他の UNIX でも動作するはずである。

^{*3} UNIX ではセマフォも使用できる。

```
1 // 共有メモリサーバ(ipcUnixShearedMemoryServer.c) :共有メモリからデータを読みだし表示する
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <sys/types.h>
7 #include <sys/ipc.h>
8 #include <sys/shm.h>
9 #define SHMSZ      512                         // 共有メモリのサイズ
10 int main() {
11     key_t key=ftok("shm.dat",'R');           // キーを作る
12     if (key==-1) {                           // エラーチェック
13         perror("shm.dat");
14         exit(1);
15     }
16     int shmid=shmget(key,SHMSZ,IPC_CREAT|0666); // 共有メモリを作る
17     if (shmid<0) {                           // エラーチェック
18         perror("shmget");
19         exit(1);
20     }
21     char *data=shmat(shmid,NULL,0);          // 共有メモリを貼り付ける
22     if (data==(char *)-1) {                  // エラーチェック
23         perror("shmat");
24         exit(1);
25     }
26     strcpy(data, "initialization...\n");      // 共有メモリに書き込む
27     while(1) {                                // 共有メモリの内容を
28         printf("sheared memory:%s",data);       // 5秒に1度メモリを表示
29         if (strcmp(data, "end\n") == 0) break;    // "end"なら終了
30         sleep(5);
31     }
32     if (shmctl(data) == -1){                  // 共有メモリをアドレス空間
33         perror("shmctl");                   // と切り離す
34         exit(1);
35     }
36     if (shmctl(shmid, IPC_RMID, 0) == -1){    // 共有メモリを廃棄する
37         perror("shmctl");
38         exit(1);
39     }
40     return 0;
41 }
```

図 6.3 UNIX の共有メモリサーバ例

```

1 // 共有メモリクライアント(ipcUnixShearedMemoryClient.c) :共有メモリにデータを書き込む
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <errno.h>
5 #include <sys/types.h>
6 #include <sys/ipc.h>
7 #include <sys/shm.h>
8 #define SHMSZ      512           // メモリのサイズ
9 int main() {
10     int      shmid;
11     key_t    key;
12     char    *data, *s;
13     if ((key=ftok("shm.dat",'R')) == -1) { // サーバ側と同じキーを作る
14         perror("shm.dat");
15         exit(1);
16     }
17     if ((shmid=shmget(key,SHMSZ,0666))<0) { // 共有メモリを取得する
18         perror("shmget");
19         exit(1);
20     }
21     data=shmat(shmid,NULL,0);           // 共有メモリを貼り付ける
22     if (data == (char *)-1) {           // エラーチェック
23         perror("shmat");
24         exit(1);
25     }
26     printf("Enter a string: ");
27     fgets(data,SHMSZ,stdin);          // 共有メモリに直接入力する
28     if (shmdt(data)==-1){            // 共有メモリをメモリ空間と
29         perror("shmdt");             // 切り離す
30         exit(1);
31     }
32     return 0;
33 }
```

図 6.4 UNIX の共有メモリクライアント例

6.2.1 通信相手の指定方式 (Naming)

システムコールでメッセージの通信相手を指定する方式が二つある。

直接指定方式 相手プロセスを直接指定する方式である。図 6.7 は直接指定方式を表している。

`send()`, `receive()` システムコールの引数は、相手プロセスとメッセージになる。相手プロセスとして ANY のような記述を許すことで、多対多の通信も可能である。また、受信したメッセージをいくつか貯めることができ、バッファ付きの通信方式もあり得る。

間接指定方式 リンク（ポート、ソケット、チャネルとも呼ばれる）を作成し、通信先としてリンク

<pre>[Terminal No.1] \$./ipcUnixShearedMemoryServer sheared memory:initialization... sheared memory:initialization... sheared memory:abcdefg sheared memory:abcdefg sheared memory:abcdefg sheared memory:1234567 sheared memory:1234567 sheared memory:end \$</pre>	<pre> [Terminal No.2] \$./ipcUnixShearedMemoryClient Enter a string: abcdefg \$./ipcUnixShearedMemoryClient Enter a string: 1234567 \$./ipcUnixShearedMemoryClient Enter a string: end \$</pre>
---	---

図 6.5 UNIX のメモリ共有プログラム実行例

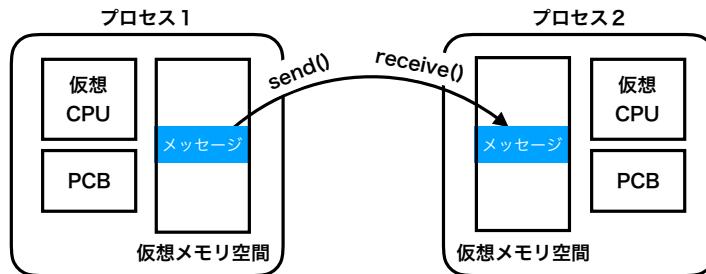


図 6.6 メッセージ通信

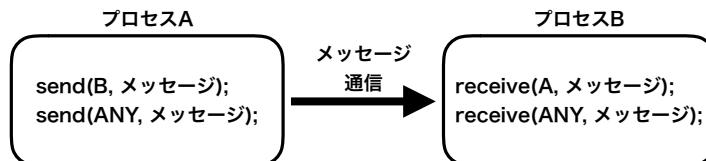


図 6.7 直接指定方式

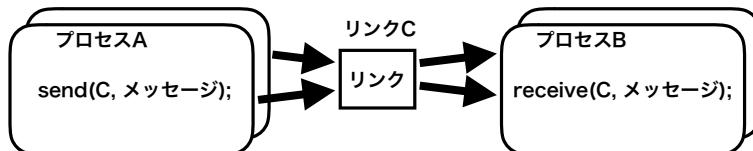


図 6.8 間接指定方式

クの名前を用いる方式である。図 6.8 は間接指定方式を表している。
`send()`, `receive()` システムコールの引数は、リンクとメッセージになる。同じリンクを共有する複数のプロセスが存在すると、自然に多対多の通信方式が実現できる。リンクにメッセージをいくつか貯めるバッファ機能を持たせる場合が多い。

6.2.2 バッファリング (Buffering)

直接指定方式か間接指定方式かに関わりなく、メッセージを格納するバッファを用意することができる。送信プロセスはバッファに空きがあれば待ち時間なしに `send()` システムコールを完了できる。受信プロセスはバッファにメッセージがあれば待ち時間なしに `receive()` システムコールを完了できる。

間接指定方式ではリンクがバッファを持つと考え、リンクを作成する時点でのバッファの大きさを指定する場合が多い。図 6.8 で「リンク」の位置にバッファがあると考えると分かりやすい。

6.2.3 メッセージの形式

通信に用いられるメッセージの形式には次のような選択肢がある。

メッセージ長 固定長方式または可変長方式

メッセージ形式 タグ付きまたはタグなし

タグは種類を表すためにメッセージに付加されるデータのことである。タグ付きのメッセージ通信機構では、送信側はメッセージにタグを付加する。受信側はタグを指定してメッセージを選択的に受信することができる。

6.2.4 同期方式 (Synchronization)

非同期方式（ノンブロッキング：Nonblocking）と同期方式（ブロッキング：Blocking）の二つがある。同期式の特別な場合としてバッファを用いないランデブー方式も考えられる。

非同期方式 `send()` はバッファに空きがない場合エラーで終了する。`receive()` はバッファにメッセージがない場合エラーで終了する。

同期方式 `send()` はバッファに空きがない場合ブロックし、空きができるのを待つ。`receive()` はバッファにメッセージがない場合ブロックし、メッセージが届くのを待つ。

ランデブー方式 `send()` は受信プロセスが `receive()` を実行するまでブロックする。`receive()` は送信プロセスが `send()` を実行するまでブロックする。両方のプロセスが `send()` と `receive()` を実行したらプロセス間でメッセージをコピーする。プロセス間で直にメッセージをコピーするので、バッファは不要である。

6.2.5 UNIX のメッセージ通信システムコール

UNIX では複数種類のメッセージ通信機構が利用可能である。ここでは、System V 系の UNIX を起原とする方式を紹介する。この方式は、間接指定方式、バッファリングあり、可変長、タグ付きの方式である。システムコールの引数によって、同期方式と非同期方式のどちらにも対応することができる。UNIX のメッセージ通信関連のシステムコール等を図 6.9 に示す。

`msgBuf` 構造体 ユーザが宣言する構造体である。必ず、`long` 型の `mtype` フィールドから始める必要がある。このフィールドがタグの役割を持つ。`mtext` はメッセージの本体を格納する領域であり、ユーザが自由に大きさや用途を決めることができる。

`msgget()` システムコール リンク（メッセージキューと呼ぶ）の ID を返す。`key` は、共有メモリの場合と同様に `ftok()` 関数を用いて生成した値である。メッセージキューを識別するために用いる。`msgflg` に `IPC_CREAT` を指定することで、メッセージキューを新規に作成することも

メッセージ通信に関するシステムコールとデータ構造
 メッセージ構造体(以下の構造体を自分で宣言して使用する)

```
struct msgbuf {
    long mtype;          // メッセージの型
    char mtext[N];       // メッセージの本体(N バイト)
};
```

メッセージキューの ID を返す。

```
int msgget(key_t key, int msgflg); (システムコール)
```

返り値 : メッセージキュー ID
 key : キー(ftok() で作成したもの)
 msgflg : IPC_CREAT 等のフラグとアクセス許可ビット

メッセージキューにメッセージを送信する(システムコール)

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

返り値 : 0=正常, -1=エラー
 msqid : メッセージキュー ID
 msgp : メッセージ構造体のポインタ
 msgsز : メッセージ本体のバイト数
 msgflg : IPC_NOWAIT 等のフラグ

メッセージキューからメッセージを受信する(システムコール)

```
int msgrcv(int msqid, const void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

返り値 : -1=エラー、受信したメッセージの本体バイト数
 msqid : メッセージキュー ID
 msgp : メッセージ構造体のポインタ
 msgsز : メッセージ本体の最大バイト数
 msgtyp : 受信するメッセージの型
 msgflg : IPC_NOWAIT 等のフラグ

メッセージキューの制御(システムコール)

```
int msgctl(int msqid, int cmd, struct msgid_ds *buf);
```

返り値 : -1=エラー、0<=コマンドにより異なる
 msqid : メッセージキュー ID
 cmd : 削除(IPC_RMID)等のコマンド
 buf : コマンドのパラメータ

図 6.9 UNIX のメッセージ通信関連システムコールとデータ構造

```

1 // ipcUnixMessage.h : メッセージ構造体の宣言
2 #define MAXMSG 100                                // メッセージ本体の長さ
3 struct msgBuf {                                    // メッセージ格納用構造体
4     long mtype;                                     // メッセージの型
5     char mtext[MAXMSG];                            // メッセージの本体
6 };

```

図 6.10 UNIX のメッセージ通信プログラム例（メッセージ構造体）

できる。

`msgsnd()` システムコール メッセージキューにメッセージを送信する。`msgp` に送信するメッセージを格納した `msgBuf` 構造体のポインタを渡す。メッセージは可変長方式なので `msgsz` で長さを指定する。`msgsz` は構造体全体ではなく、構造体の `mtext` 部分のバイト数である。`msgflg` に `IPC_NOWAIT` フラグを指定すると非同期方式になり、指定しないと同期方式になる。

`msgrcv()` システムコール メッセージキューからメッセージを受信する。`msgp` に受信したメッセージを格納する `msgBuf` 構造体のポインタを渡す。`msgsz` は受信可能な `mtext` の最大バイト数である。`msgtyp` に受信したいメッセージの `mtype` を指定し、タグが合致するメッセージを選択的に受信できる。`msgflg` に `IPC_NOWAIT` フラグを指定すると非同期方式になる。

`msgclt()` システムコール メッセージキューに対して操作を行う。`cmd` に操作の種類（コマンド）、`buf` にパラメータを渡す。`IPC_RMID` コマンドを指定するとメッセージキューの削除ができる。

6.2.6 UNIX のメッセージ通信プログラム例

メッセージを表現する構造体の例を図 6.10 に示す^{*4}。メッセージ本体の長さは `MAXMSG` に定義している。以下のプログラムは、メッセージ長をこの値に固定した例になっている。

図 6.11 にメッセージキューを作成しメッセージを書き込むプログラムの例を示す。このプログラムは入力した文字列をメッセージ本体に格納してメッセージキューに送信する。タグの役割を持つ `mtype` は常に 1 にしている。

図 6.12 に、メッセージキューからメッセージを読み込み内容を表示するプログラムの例を示す。22 行で `msgtyp` を 0 にして `msgrcv()` を実行している。`msgtyp` が 0 の場合は、メッセージの `mtype`（タグ）を無視してメッセージキューの先頭から順にメッセージを受信する。26 行で `mtype` と `mtext` の内容を表示している。送信側のプログラムがメッセージキューを削除すると 22 行でエラーが発生し 24 行で終了する。

6.2.7 UNIX のメッセージ通信プログラムの実行例

メッセージ通信プログラムの実行例を図 6.13 に示す。図は二つのターミナルを開いて操作した状態を示している。左半分が第一のターミナル、右半分が第二のターミナルである。まず、左のターミナル

^{*4} ここで紹介するプログラムは macOS 10.13.2 で動作確認した。macOS のオンラインマニュアルには、ここで紹介するメッセージ通信方式について記載がないが、試してみると使用できた。

```

1 // メッセージ送信プログラム(ipcUnixMessageWriter.c) : メッセージキューを作成し送信する
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/msg.h>
7 #include "ipcUnixMessage.h"                                // msgBuf 構造体の宣言
8 int main() {
9     struct msgBuf buf;                                     // メッセージ領域
10    int msqid;                                         // メッセージキュー ID
11    key_t key;                                          // メッセージキューの名前
12    if ((key=ftok("msgq.dat",'b'))== -1) {             // ftok はファイル名から
13        perror("ftok");                                 // 重複のない名前(キー)を
14        exit(1);                                       // 生成する
15    }
16    if ((msqid=msgget(key,0644|IPC_CREAT))== -1) { // メッセージキューを作る
17        perror("msgget");
18        exit(1);
19    }
20    printf("Enter lines of text, ^D to quit:\n");
21    buf.mtype = 1;                                       // メッセージの型
22    while (fgets(buf.mtext,MAXMSG,stdin)!=NULL) { // キーボードから1行入力
23        if (msgsnd(msqid,&buf,MAXMSG,0)== -1) {       // メッセージを送信
24            perror("msgsnd");
25            break;
26        }
27    }
28    if (msgctl(msqid,IPC_RMID,NULL) == -1) {           // メッセージキューを削除
29        perror("msgctl");
30        exit(1);
31    }
32    exit(0);
33 }
```

図 6.11 UNIX のメッセージ通信プログラム例（メッセージ送信側）

で送信プログラム（ipcUnixMessageWriter）を起動する。これでメッセージキューが準備された。次に、右のターミナルで受信プログラム（ipcUnixMessageReader）を起動する。この状態で左のターミナルに入力した文字列が、メッセージ通信を用いて右のターミナルで実行中のプログラムに送信される。右のターミナルには受信したメッセージの `mtype` と `mtext` の内容が表示される。

```

1 // メッセージ受信プログラム(ipcUnixMessageReader) : メッセージキューから受信する
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/msg.h>
7 #include "ipcUnixMessage.h"
8 int main() {
9     struct msgBuf buf;
10    int msqid;
11    key_t key;
12    if ((key=ftok("msgq.dat",'b'))== -1) {           // 送信側と同じキーを作る
13        perror("ftok");
14        exit(1);
15    }
16    if ((msqid=msgget(key,0644))== -1) {             // ipcUnixMessageReader が作った
17        perror("msgget");                            // メッセージキューを取得
18        exit(1);
19    }
20    printf("ready to receive messages.\n");
21    for(;;) {
22        if (msgrcv(msqid,&buf,MAXMSG,0,0)== -1) { // 先頭のメッセージを読み出す
23            perror("msgrcv");                      // メッセージキューが削除され
24            exit(1);                                // エラーが発生したら終了
25        }
26        printf("%ld:%s",buf.mtype,buf.mtext);      // 受信したメッセージを表示
27    }
28    exit(0);
29 }
```

図 6.12 UNIX のメッセージ通信プログラム例（メッセージ受信側）

[Terminal No.1] \$./ipcUnixMessageWriter Enter lines of text, ^D to quit: abcdefg 1234567 ^D \$	[Terminal No.2] \$./ipcUnixMessageReader ready to receive messages. 1:abcdefg 1:1234567 msgrcv: Identifier removed \$
--	--

図 6.13 UNIX のメッセージ通信プログラム実行例

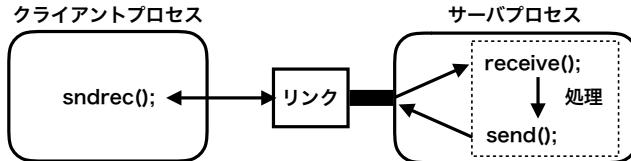


図 6.14 TacOS のメッセージ通信

6.3 TacOS のメッセージ通信機構

TacOS ではマイクロカーネルがメッセージ通信機構を提供し、ユーザ・プロセスとサーバプロセス、サーバプロセスとサーバプロセスの通信にメッセージを用いる。TacOS のメッセージ通信は、クライアントプロセスがサーバプロセスの機能を利用する、クライアント・サーバの通信に特化したものである。図 6.14 に TacOS のメッセージ通信の様子を示す。メッセージ通信の手順は次のようになる。

1. サーバプロセスがリンクを所有し他プロセスからの通信を待ち受ける。
2. クライアントプロセスは `sndrec()` 関数を用いてリンクに処理内容をメッセージとして送信する。
3. サーバプロセスは `receive()` 関数を用いてメッセージを受信する。
4. サーバプロセスはメッセージの内容に合った処理を行う。
5. サーバプロセスは処理結果を `send()` 関数を用いて返信する。
6. `sndrec()` 関数が完了し、クライアントプロセスは処理結果を受取る。

TacOS のメッセージ通信機構は、間接指定方式、固定長、ランデブー方式と言える。クライアントプロセスとサーバプロセスが並列に処理することができないが、サービスモジュールをサーバプロセスにすることでき、オペレーティングシステムを構築するためのプログラミングを容易にしている。

6.3.1 リンク構造体

図 6.15 に TacOS のリンク構造体の宣言を示す^{*5}。Link 構造体はランデブー用のリンクを定義している。`server` はリンクを所有するサーバプロセスの PCB、`client` はリンクを使用中のクライアントプロセスの PCB である。`s1`, `s2`, `s3` は相互排除と同期のために使用されるセマフォである。TacOS のリンクはセマフォを基盤にしている。`op`, `prm1`, `prm2`, `prm3` が固定長のメッセージ本体になる。

6.3.2 リンクの作成

図 6.16 に、TacOS のマイクロカーネル内にある、リンク作成ルーチンを示す^{*6}。`newLink()` 関数はサーバプロセスがリンクを作り所有するために呼出す。TacOS のサーバプロセスはカーネルモードで実行され、カーネル内ルーチンを呼出すことができる。複数のプロセスが `newLink()` 関数を呼出す可能性があるので、6 行から 15 行の範囲は割込み禁止による相互排除を行っている。

空きリンクは 7 行で管理している。リンクの廃棄手段は準備されていないので、空きリンクの管理は単純である。11 行でリンクを所有するサーバプロセスを記録する。12, 13, 14 行で、三つのセマフォをリンクに割当てている。16 行では作成したリンクの番号を返している。

^{*5} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/process.hmm> の一部である。

^{*6} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm> の一部である。

```

1 #define LINK_MAX 5          // リンクは最大 5 個
2
3 struct Link {             // リンクを表す構造体
4     PCB server;           // リンクを所持するサーバ
5     PCB client;           // リンクを使用中のクライアント
6     int s1;                // サーバがメッセージ受信待ちに使用するセマフォ
7     int s2;                // クライアント同士が相互排除に使用するセマフォ
8     int s3;                // クライアントがメッセージ返信待ちに使用するセマフォ
9     int op;                // メッセージの種類
10    int prm1;              // メッセージのパラメータ 1
11    int prm2;              // メッセージのパラメータ 2
12    int prm3;              // メッセージのパラメータ 3
13 };

```

図 6.15 TacOS のリンク構造体

```

1 Link[] linkTbl = array(LINK_MAX);           // リンクの一覧表
2 int linkID = -1;                           // リンクの通し番号
3
4 // リンクを生成する(サーバが実行する)
5 public int newLink() {
6     int r = setPri(DI|KERN);                 // 割り込み禁止、カーネル
7     linkID = linkID + 1;                     // 通し番号を進める
8     if (linkID >= LINK_MAX)                 // リンクが多すぎる
9         panic("newLink");
10    Link l = linkTbl[linkID];                // 新しく割り当てるリンク
11    l.server = curProc;                     // リンクの所有者を記録
12    l.s1 = newSem(0);                       // server が受信待ちに使用
13    l.s2 = newSem(1);                       // client が相互排他に使用
14    l.s3 = newSem(0);                       // client が返信待ちに使用
15    setPri(r);                            // 割り込み復元
16    return linkID;                         // 割当てたリンクの番号
17 }

```

図 6.16 TacOS のリンク作成ルーチン

6.3.3 サーバ用のメッセージ通信ルーチン

図 6.17 に、TacOS のマイクロカーネル内にある、サーバプロセス用のメッセージ通信プログラムを示す^{*7}。2 行の `receive()` 関数はメッセージの受信に使用する。引数 `num` は `newLink()` が返したリンク番号である。4 行でリンクの所有者を調べている。所有者が自身ではないならオペレーティングシステムのバグなので `panic()` 関数を用いてシステムを停止する。5 行で初期値 0 のセマフォ (`s1`) に P

^{*7} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm> の一部である。

```

1 // サーバ用の待ち受けルーチン
2 public Link receive(int num) {
3     Link l = linkTbl[num];
4     if (l.server != curProc) panic("receive");           // 登録されたサーバではない
5     semP(l.s1);                                         // サーバをロック
6     return l;
7 }
8
9 // サーバ用の送信ルーチン
10 public void send(int num, int res) {
11     Link l = linkTbl[num];
12     if (l.server != curProc) panic("send");           // 登録されたサーバではない
13     l.op = res;                                       // 処理結果を書込む
14     semV(l.s3);                                     // クライアントを起こす
15 }
```

図 6.17 TacOS のメッセージ通信ルーチン（サーバ用）

操作を行い、クライアントがリンクにデータを書き込むのを待つ。6行でデータが書き込まれたリンクを返す。

10行の `send()` 関数は、クライアントプロセスにメッセージを返信するために使用する。引数 `num` はリンク番号、`res` は返信するデータである。サーバが行った処理の結果を16ビット（2バイト）で表現する。12行では `receive()` 関数と同様にリンクの所有者を調べている。13行で処理結果をリンクに書き込み、14行でクライアントが待ち合せているセマフォ (`s3`) に V 操作を行う。これでクライアントが処理結果を受取り処理を再開する。

6.3.4 サーバプロセスの例

図 6.18 にサーバプロセスの例として、プロセスマネージャのメインルーチンを示す^{*8}。プロセスマネージャは `exec` システムコール等を処理するサーバプロセスである。3行でリンクを作成し `pmLink` グローバル変数^{*9}に記録する。5行でクライアントプロセスからのメッセージを待ち受ける。メッセージを受信したら 6 行に進み、リンクに書き込まれていた内容とクライアントプロセスの PCB を引数に、プロセスマネージャのシステムコール処理ルーチンを実行する。処理結果は 7 行の `send()` 関数を用いてクライアントプロセスに返信する。

6.3.5 クライアント用のメッセージ通信ルーチン

図 6.19 に TacOS のクライアントプロセス用のメッセージ通信プログラム (`sendrec()`) を示す^{*10}。
`sendrec()` 関数はサーバプロセスのリンクにメッセージを書き込み、サーバプロセスに処理を依頼する。サーバプロセスが処理を完了したら、`sendrec()` 関数は処理結果を返り値として終了する。

4行では初期値 1 のセマフォ (`s2`) を用いてリンクをロックし、他のクライアントプロセスとの相互

^{*8} <https://github.com/tctsigemura/TacOS/blob/master/os/pm/pm.cmm> の一部である。

^{*9} <https://github.com/tctsigemura/TacOS/blob/master/os/pm/pm.hmm> で宣言されている。

^{*10} <https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm> の一部である。

```

1 // プロセスマネージャサーバのメインルーチン
2 public void pmMain() {
3     pmLink = newLink();                                // リンクを生成する
4     while (true) {                                     // システムコールを待つ
5         Link l = receive(pmLink);                     // システムコールを受信
6         int r=pmSysCall(l.op,l.prm1,l.prm2,l.prm3,l.client);// システムコール実行
7         send(pmLink, r);                             // 結果を返す
8     }
9 }
```

図 6.18 TacOS のメッセージ通信使用例（サーバ側）

```

1 // クライアント用メッセージ送受信ルーチン
2 public int sndrec(int num, int op, int prm1, int prm2, int prm3) {
3     Link l = linkTbl[num];                           // 他のクライアントと相互
4     semP(l.s2);                                    // 排除しリンクを確保
5     l.client = curProc;                            // リンク使用中プロセス記録
6     l.op = op;                                     // メッセージを書込む
7     l.prm1 = prm1;
8     l.prm2 = prm2;
9     l.prm3 = prm3;
10    int r = setPri(DI|KERN);                      // 割り込み禁止、カーネル
11    iSemV(l.s1);                                  // サーバを起こす
12    semP(l.s3);                                    // 返信があるまでブロック
13    setPri(r);                                    // 割り込み復元
14    int res = l.op;                               // 返信を取り出す
15    semV(l.s2);                                   // リンクを解放
16    return res;
17 }
```

図 6.19 TacOS のメッセージ通信ルーチン（クライアント用）

排除を行っている。6 行から 9 行でリンクにメッセージを書き込む。iSemV() 関数を使用するために、10 行から 13 行まで割込み禁止による相互排除を行っている。11 行でメッセージを書き込んだことをサーバに知らせ、12 行で初期値 0 のセマフォ (s3) に P 操作を行いサーバが処理を終了するのを待つ。サーバの処理が終了したら 14 行に進みサーバがリンクに書き込んだ処理結果を取り出す。15 行でリンクのロックを解除し 16 行で処理結果を持って関数を終了する。

6.3.6 クライアントプロセスの例

図 6.20 にクライアントプロセスの例として、プロセスマネージャ（サーバプロセス）に exec システムコールの処理を依頼するプログラムを示す^{*11}。TacOS の exec システムコールは、新しいプロセスを

^{*11} <https://github.com/tctsigemura/TacOS/blob/master/os/pm/pm.cmm> の一部である。

```

1 public int exec(char[] path, char[][] argv) {
2     int r=sndrec(pmLink,EXEC,_AtoI(path),_AtoI(argv),0);
3     return r;                                     // 新しい子の PID を返す
4 }
```

図 6.20 TacOS のメッセージ通信使用例（クライアント側）

作ってプログラムを実行させる。引数はプログラムファイルのパス名 (`path`) と、新しいプログラムの `main()` 関数に渡すコマンド行引数 (`argv`) である。

1行はクライアントプロセスの `exec` システコールの入口になる。カーネルモードで動作する他のサーバプロセスは `exec()` 関数を直に呼出す。ユーザモードで動作するユーザプロセスは SVC 機械語命令で割込みを発生し、SVC 割込みハンドラから `exec()` 関数を呼出す。割込みハンドラは現在のプロセスのコンテキストで実行されるので、`exec()` 関数はカーネルモードに切り換わった状態のユーザプロセスによって実行されることになる。

2行でプロセスマネージャ（サーバプロセス）とランデブーを行う。`pmLink` は図 6.18 でプロセスマネージャが生成したリンクである。`EXEC` がシステムコールの種類を表している。システムコールの二つの引数は `_AtoI()` 関数を用いて `int` 型に変換して渡している。処理結果は子プロセスのプロセス番号 (PID) である。3行で PID を呼び出し側に返す。

6.4 まとめ

この章ではプロセス間通信 (IPC) について学んだ。IPC には共有メモリとメッセージ通信の二種類があった。UNIX の共有メモリとメッセージ通信についてプログラム例を示した。TacOS のメッセージ通信について、それを実現するカーネル内プログラムと利用例を示した。

練習問題

6.1 プロセス間の共有メモリとスレッド間の共有変数の違いは何か？

6.2 UNIX の共有メモリ使用例（図 6.3, 図 6.4）を実際に実行し動作確認しなさい。なお、ソースプログラムは以下から入手可能である。

<https://github.com/tctsigemura/OSTextBook/blob/master/Lst/>

`ipcUnixShearedMemoryServer.c` (サーバ側)

<https://github.com/tctsigemura/OSTextBook/blob/master/Lst/>

`ipcUnixShearedMemoryClient.c` (クライアント側)

6.3 動作確認したプログラムでは、サーバプログラムは共有メモリが変更されたことを確認しないで、一定の時間間隔で共有メモリの内容を表示している。

(a) どのような不都合が予想されるか？

(b) クライアントとサーバで同期をする方法はあるか？

6.4 メッセージ通信でバッファを大きくすることのメリットは何か？

6.5 UNIX のメッセージ通信プログラム例（図 6.10, 図 6.11, 図 6.12）を実際に実行し動作確認し

なさい。なお、ソースプログラムは以下から入手可能である。

<https://github.com/tctsigemura/OSTextBook/blob/master/Lst/ ipcUnixMessage.h>
(ヘッダファイル)

<https://github.com/tctsigemura/OSTextBook/blob/master/Lst/ ipcUnixMessageWriter.c> (送信側)

<https://github.com/tctsigemura/OSTextBook/blob/master/Lst/ ipcUnixMessageReader.c> (受信側)

6.6 UNIX のメッセージ通信プログラム例は生産者と消費者の問題の解になっている。複数生産者と複数消費者の問題の解にもなっているか？

6.7 UNIX のメッセージ通信プログラム例が複数生産者と複数消費者の問題の解にもなっているか、動作確認する手順を説明しなさい。

6.8 TacOS のメッセージ通信機構について正しいか正しくないか答えなさい。

- (a) メッセージの形式に柔軟性がある。
- (b) リンクに三つのセマフォが含まれる。
- (c) TacOS のメッセージ通信機構は相互排除と同期にセマフォだけを用いている。
- (d) 複数生産者と複数消費者の問題の解に使用できる。

第7章

モニタ

複数のプロセス（スレッド）で資源を共有する際に、プロセスの同期や相互排除にセマフォを用いることを既に学んだ。しかし、セマフォは基本的な機能を提供するだけで使い方はプログラマ任せなので、間違った使用がされる可能性が高い。その結果、タイミングに依存した発見の難しいバグを持ったプログラムが作成される。そこで、プログラミング言語と一体になり^{*1}、プログラマに同期機構を強制的に利用させる仕組みが考案された。そのような仕組みの一つとしてモニタ（Monitor）を紹介する。

7.1 概要

モニタはリソース管理用の機能と制約を持った抽象データ型 [40] である。C++ や Java などを学んだことのある人なら、「抽象データ型はクラスのこと」と言えば分かりやすいと思われる。

モニタは抽象データ型の一種であるが、プロセス（スレッド）間の同期を行う機構が組込まれ、その制約の下で使用するものである。モニタの特長を以下に箇条書きにする。

- プログラムが定義できる型である。（抽象データ型で一般的）
- データと操作をまとめて定義する。（抽象データ型で一般的）
- 同期のための機能が組込まれている。（モニタ独特）

なお、Java のクラスは同期のための機構も持っており、一定のルールに従って使用すればモニタに近い使用もできる。モニタをサポートするプログラミング言語は Concurrent Pascal が有名である。

7.2 構成要素

モニタは次の要素を持つ抽象データ型である。図 7.1 にモニタの模式図^{*2}を示す。

- 資源（データ、変数）
- 手続き（操作、メソッド）
- ガード
- 条件変数

^{*1} 本章の話題はオペレーティングシステムではなくプログラミング言語である。

^{*2} 図 7.1 では、初期化プログラムが省略してある。

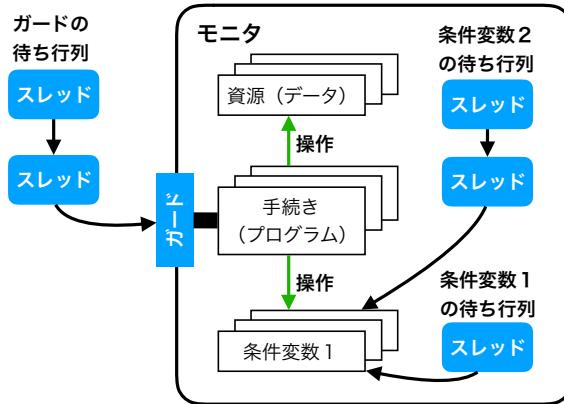


図 7.1 モニタの模式図

7.2.1 資源（データ、変数）

複数のスレッドによって共有される変数のことである。モニタの内部に必要に応じて名前付きで宣言する。モニタの外から直接アクセスすることはできない。

7.2.2 手続き（操作、メソッド）

外部から呼び出されるプログラムである。モニタの内部に必要に応じて名前付きで定義する。モニタの外部から資源にアクセスできるインターフェースは手続きだけである。手続きの実行はガードの働きにより排他的に行われ、同時に実行される手続きは必ず一つ以内である。

7.2.3 ガード

モニタに一つのガードが存在し、手続きを排他的に実行するために用いられる。手続きを実行するときは自動的にガードにロックがかけられる。複数のスレッドが同時にモニタに入ることはできない。

7.2.4 条件変数

モニタの内部に必要に応じて名前付きで宣言する。条件変数には wait と signal の二つの操作ができる。wait 操作を行ったスレッドはガードを外して条件変数の待ち行列に入る。signal 操作は条件変数の待ち行列から一つのスレッドを選んで実行可能にする。実行可能になったスレッドはただちに実行を再開する。待ち行列にスレッドが複数ある時、どのスレッドが実行可能になるか明確な決まりはない。

7.2.5 初期化プログラム

モニタのインスタンスを作成する時に、初期化のために実行されるプログラムである。

7.3 相互排除問題の解

前出の架空の銀行口座管理プログラムの例をモニタに置換える。残高がスレッド間で共有される変数である。共有変数をスレッド間で安全に共有させるためにモニタを用いる。

7.3.1 共有変数の記述

図 7.2 に Java 風の仮想言語による銀行口座の記述を示す。本物の口座は他にも情報を持っているだろうが、ここでは口座は残高だけ持つことにする。

2 行 この仮想言語では、Java の class 定義に似た monitor 定義ができるものとす

```

1 // 銀行口座の残高を管理するモニタ
2 monitor MonAccount {
3     // 資源
4     int money;                                // スレッド間の共有変数(残高)
5     // 初期化プログラム
6     MonAccount(int m) {
7         money = m;                            // 口座の残高を初期化する
8     }
9     // 手続き
10    public void receive(int r) {              // 入金手続き
11        money = money + r;
12    }
13    public void pay(int p) {                  // 引落し手続き
14        money = money - p;
15    }
16}

```

図 7.2 モニタによる相互排除の実現（仮想言語版）

3~4 行 資源の例である。この例では残高を表すスレッド間の共有変数（`money`）が資源である。

5~8 行 初期化プログラムの例である。モニタのインスタンス生成時に残高を引数で初期化する。

9~15 行 手続きの例である。手続きは共有資源を書き換えるのでクリティカルセクションであるが、自動的にガードをロックし排他的に実行されるので明示的な相互排除を行う必要はない。

7.3.2 共有変数の利用

図 7.3 に、図 7.2 で定義した `MonAccount` モニタを利用した相互排除問題の解を示す。

2 行 図 7.2 に示した `MonAccount` モニタ型のインスタンスを生成する。

3~8 行 入金管理スレッドが実行するメソッドである。入金額を受信し口座に入金する。

9~13 行 引落し管理スレッドが実行するメソッドである。支払い金額を受信し口座から引落す。

14~17 行 プログラムは `main()` から実行を開始する。`main()` では「入金管理スレッド」と「引落し管理スレッド」を起動する。これらのスレッドがそれぞれ、`receiveThread()` メソッドと `payThread()` メソッドを実行するものとする。

7.4 生産者と消費者問題の解

この問題で資源はデータを保管する FIFO 構造のバッファである。このバッファを以下ではキューと呼ぶことにする。キューとキューを操作する手続きは、全てモニタの中にまとめられる。キューを使用するユーザプログラムには、排他や同期に関わる難しいプログラムが含まれない。資源の操作に関する難しいプログラムが一箇所にまとめられることも、モニタを使用するメリットである。

以下では生産者と消費者問題の解を示すために、まず、データのバッファになるキューをモニタ用

```

1 class MonAccoutMain {
2     static MonAccount account = new MonAccount(0); // 残高 0 円の口座を作る
3     static void receiveThread() { // 入金管理スレッド
4         for ( ; ; ) { // 以下を繰り返す
5             int receipt = receiveMoney(); // ネットワークから入金を受信
6             account.receive(receipt); // 口座に入金する
7         }
8     }
9     static void payThread() { // 引落し管理スレッド
10        for ( ; ; ) { // 以下を繰り返す
11            int payment = payMoney(); // ネットワークから支払いを受信
12            account.pay(payment); // 口座から引落す
13        }
14    public static void main(String[] args) {
15        入金管理スレッドを起動;
16        引落し管理スレッドを起動;
17    }
18 }
```

図 7.3 モニタによる相互排除の利用（仮想言語版）

いて記述する。次に、キューを使用する生産者スレッドと消費者スレッド作る。

7.4.1 キューの記述

図 7.4 に Java 風の仮想言語によるキューの記述例を示す。このプログラムは次のようになっている。

1行 この仮想言語では、Java の `class` 定義に似た `monitor` 定義ができるものとする。

2~5行 資源の例である。キューとして使用するリングバッファのデータ構造を宣言している。このモニタの目的は、資源であるキューを管理することである。手続きを介すこと無しに資源にアクセスすることは禁止なので、モニタの外部からこれらのデータにアクセスできない。`N` がバッファの大きさ、`buf` がバッファ本体、`first` がバッファ中の次のデータ読みだし位置、`last` がバッファ中の次のデータ書き込み位置、`cnt` がバッファ中のデータ件数を表す。

6~8行 条件変数の例である。この仮想言語では、`Condition` 型の変数として条件変数を宣言する。`empty` は、キューが空の時にデータを取り出そうとしたスレッドが、キューにデータが書き込まれるまで待つために使用する条件変数である。`full` は、キューが満杯の時にデータを書き込もうとしたスレッドが、キューに空きができるまで待つために使用する条件変数である。

9~14行 初期化プログラムの例である。モニタのインスタンスを作る際に実行されるものとする。引数はバッファの大きさである。

15~30行 手続きの例である。手続きはモニタの外部から呼び出し可能なプログラムである。`append()` はキューにデータを追加する。`remove()` はキューからデータを取り出す。これらのプログラムが実行される時はガードによる排他制御がされる。次にバッファが満杯で待ちが発生する例を考えてみる。

```

1 monitor BoundedBuffer {
2     // 資源(リングバッファ)
3     int N;
4     int[] buf;
5     int first, last, cnt;
6     // 条件変数
7     Condition empty;
8     Condition full;
9     // 初期化
10    BoundedBuffer(int n) {
11        N = n;
12        buf = new int[N];
13        first = last = cnt = 0;
14    }
15    // 手続き
16    public void append(int x) {    // (1)
17        if (cnt==N) full.wait();    // (1)
18        buf[last] = x;            // (3)
19        last = (last + 1) % N;    // (3)
20        cnt++;                  // (3)
21        empty.signal();          // (3)
22    }
23    public int remove() {         // (2)
24        if (cnt==0) empty.wait(); // (2)
25        int x = buf[first];    // (2)
26        first = (first + 1) % N; // (2)
27        cnt--;                  // (2)
28        full.signal();          // (2)
29        return x;                // (4)
30    }
31 }
```

図 7.4 モニタによるキューの実現（仮想言語版）

データをキューに追加するために `append()` を呼出したスレッドは、コメントに (1) と記された行を実行し、キューが満杯の時 17 行の `full.wait()` で待ち状態になる。待ち状態になる時はガードを外すので、他のスレッドがモニタに入ることができる。

他のスレッドがデータをキューから取り出すために `remove()` を呼び出すと、(2) の行が実行され 28 行の `full.signal()` まで進む。`full.signal()` が実行されると待ち状態のスレッドが一つ起床し、(3) の行がただちに実行される。(3) の実行が終了した後に (4) の行が実行される。(2) から (4) の実行の間、ガードは外さないので他のスレッドがモニタに入ることはない。

```

1 class BoundedBufferMain {
2     static BoundedBuffer queue = new BoundedBuffer(10); // 大きさ 10 のキュー
3     static void producer() { // 生産者スレッドが実行する
4         while(true) {
5             int x = データを作る();
6             queue.append(x); // キューにデータを追加する
7         }
8     }
9     static void consumer() { // 消費者スレッドが実行する
10        while(true) {
11            int x = queue.remove(); // キューからデータを取り出す
12            データを使用する(x);
13        }
14    }
15    public static void main(String[] args) { // main から実行を開始する
16        生産者スレッドを起動;
17        消費者スレッドを起動;
18    }
19 }
```

図 7.5 モニタによる生産者と消費者問題の解（仮想言語版）

7.4.2 生産者と消費者スレッドの記述

図 7.5 に Java 風の仮想言語による生産者と消費者問題の解を示す。このプログラムは図 7.4 で定義したキューを使用する。

2 行 図 7.4 に示した BoundedBuffer モニタ型のインスタンスである。

3~8 行 生産者スレッドが実行するメソッドである。無限にデータを生産し queue に追加し続ける。

9~14 行 消費者スレッドが実行するメソッドである。queue からデータを取り出し処理することを無限に繰り返す。

15~18 行 プログラムは main() から実行を開始する。main() では「生産者スレッド」と「消費者スレッド」を起動する。これらのスレッドがそれぞれ、producer() メソッドと consumer() メソッドを実行する。

7.5 セマフォによるモニタの実装

モニタの仕組みをより正確に理解するために、セマフォによるモニタの実装方法を考えてみる。図 7.4 の仮想言語で記述されたモニタを Java クラスに書換えたものを図 7.6 と図 7.7 に示す。モニタをサポートする言語のコンパイラは、自動的にこのような変換を行っている。

7.5.1 モニタ機能の Java による実装

図 7.6 に SemBoundedBuffer クラスがモニタと同等な動作をするために必要な機能の実装を示す。

```

1 import java.util.concurrent.Semaphore;           // セマフォ型を利用可能にする
2 public class SemBoundedBuffer {
3     private Semaphore guard = new Semaphore(1); // ガード用のセマフォ
4     private Semaphore next = new Semaphore(0); // signal 時にブロックするためのセマフォ
5     private int nextCont = 0;                  // signal 時にブロックしたスレッド数
6     private class Condition {                // 条件変数型を内部クラスとして定義する
7         Semaphore sem = new Semaphore(0);    // 条件変数で待つためのセマフォ sem
8         int count = 0;                     // 条件変数を待つスレッドの数
9         void await() {                   // 条件変数を待つメソッド
10            count++;
11            if (nextCont>0) {           // この条件変数を待つスレッドの数
12                next.release();       // 起床後に await() した場合なら
13            } else {                 // 起こすスレッドがないなら
14                guard.release();     // ガードを外してからブロック
15            }
16            sem.acquireUninterruptibly(); // 条件変数のセマフォで待つ
17            count--;
18        }
19        void signal() {             // 条件変数で待つスレッドを起こす
20            if (count>0) {           // 待っているスレッドがあれば
21                nextCont++;          // signal 途中のスレッド数
22                sem.release();       // 待ちスレッドを起こす
23                next.acquireUninterruptibly(); // 起きたスレッドを先に実行する
24                nextCont--;          // signal 完了
25            }
26        }
27    }
28    private void exitProc() {           // 手続きの出口処理
29        if (nextCont>0) {             // signal された後なら
30            next.release();          // signal したスレッドを起こす
31        } else {                   // そうでなければ
32            guard.release();        // ガードを外す
33        }
34    }
}

```

図 7.6 モニタと同等なキューをセマフォで実現（Java 版、前半）

- 1 行 `java.util.concurrent` パッケージの `Semaphore` クラスを使用する。 `Semaphore` クラスはカウンティングセマフォ型である。
- 2 行 セマフォを使用したキュークラスを `SemBoundedBuffer` と名付ける。
- 3 行 セマフォ (`guard`) はモニタのガードの役割りを持っている。スレッドは、モニタ (`SemBoundedBuffer` クラス) 内の手続き (メソッド) を実行する前に、`guard` をロックする。
- 4~5 行 モニタの条件変数に `signal()` 操作を行った時、条件変数で待っていたスレッドがあればた

```

35 // 資源(リングバッファ)
36 private int N;
37 private int[] buf;
38 private int first, last, cnt;
39 // 条件変数
40 private Condition empty = new Condition();
41 private Condition full = new Condition();
42 // 初期化
43 public SemBoundedBuffer(int n) {
44     N = n;
45     buf = new int[N];
46     first = last = cnt = 0;
47 }
48 // 手続き
49 public void append(int x) {           // (1)
50     guard.acquireUninterruptibly();    // (1) ガードを取得
51     if (cnt==N) full.await();          // (1)
52     buf[last] = x;                  // (3)
53     last = (last + 1) % N;            // (3)
54     cnt++;                          // (3)
55     empty.signal();                 // (3)
56     exitProc();                     // (3) 手続きの出口処理
57 }
58 public int remove() {                // (2)
59     guard.acquireUninterruptibly();    // (2) ガードを取得
60     if (cnt==0) empty.await();         // (2)
61     int x = buf[first];              // (2)
62     first = (first + 1) % N;          // (2)
63     cnt--;                           // (2)
64     full.signal();                  // (2)
65     exitProc();                     // (4) 手手続きの出口処理
66     return x;                        // (4)
67 }
68 }
```

図 7.7 モニタと同等なキューをセマフォで実現（Java 版、後半）

だちに実行しなければならない。待っていたスレッドを先に実行させるために、`signal()` 操作を行ったスレッドを待ちにするセマフォ (`next`) と `next` を待っているスレッドの数を記憶する変数 (`nextCont`) を準備する。

6 行 条件変型型 (`Condition`) を内部クラスとして定義する。

7~8 行 条件変数に `wait` 操作を行った時にスレッドを待ち状態にするためのセマフォ (`sem`) と待っているスレッドの数をカウントする変数 (`count`) である。

9行 `await()` は条件変数の `wait` 操作を行うメソッドである。Java の `Object` クラスに別の `wait()` メソッドが定義されているので、名前を `await` にした。

11~15行 `release()` はセマフォに V 操作を行う。`nextCont` は `signal()` 中で待っているスレッドの数である。待っているスレッドがある場合は起こす。そうでなければモニタのガードを外す。

16行 `acquireUninterruptibly()` はセマフォに P 操作を行う。`sem` は初期値 0 のセマフォなので、`await()` を呼出したスレッドがここでブロックする。

19行 条件変数に `signal` 操作を行うメソッドである。

20~25行 条件変数を待っているスレッドの数 (`count`) を調べ、1 以上なら 22 行で起床させる。自身は 23 行でブロックし、起床したスレッドが `exitProc()` を実行しモニタを出るのを待つ。

28行 モニタ手続きの最後の行で呼び出すメソッドである。`signal()` 中の 23 行でブロックしているスレッドがあれば起床させる。なければモニタのガードを外す。

7.5.2 モニタ機能の使用

図 7.7 に `SemBoundedBuffer` クラスの後半を示す。ここでは、`SemBoundedBuffer` クラスの前半で定義したモニタ機能を使用している。

35~38行 資源（リングバッファ）を表現するための変数を宣言する。モニタ（クラス）の外部から資源を隠蔽するために `private` 修飾子を付けて宣言する。

39~41行 条件変数は前半で定義した `Condition` クラスのインスタンス変数である。

42~47行 初期化はクラスのコンストラクタとして実装する。

48~67行 手続きは仮想言語で定義したものに、50行と59行の「ガード取得」、56行と65行の「手続きの出口処理」が追加されている。

7.6 Java のモニタ風機構による生産者と消費者問題の解

Java 言語はモニタに似た同期機構をサポートしている。図 7.8 に Java による生産者と消費者問題の解を示す。Java には条件変数に相当するものが無い。

1行 Java のモニタ風機構を利用したキュークラスを `MonBoundedBuffer` と名付ける。

2~5行 資源（リングバッファ）を表現するための変数を宣言する。クラスの外部から資源を隠蔽するために `private` 修飾子を付けて宣言する。

6~11行 初期化をコンストラクタとして実装する。

13行 クラスの内部だけで呼び出す `private` なメソッドである。`await()` メソッドは条件変数の `wait()` に似た働きをする。

14行 `await()` メソッドは `Object` クラスの `wait()` メソッドを呼び出す。Java オブジェクトは暗黙の条件変数が一つあるような構造になっている。`wait()` メソッドは暗黙の条件変数の待ち行列^{*3} にスレッドを入れる。`wait()` メソッドは割込みなどでも終了するので try-catch 構文で使用する。

16, 23行 外部から呼び出すことができるメソッドは `synchronized` 修飾子を付けて定義する。

^{*3} Java では待機セットと呼ぶ。

```

1 public class MonBoundedBuffer {
2     // 資源(リングバッファ)
3     private int N;
4     private int[] buf;
5     private int first, last, cnt;
6     // 初期化
7     public MonBoundedBuffer(int n) {
8         N = n;
9         buf = new int[N];
10        first = last = cnt = 0;
11    }
12    // 手続き
13    private void await() {
14        try{wait();}catch(InterruptedException e){}
15    }
16    public synchronized void append(int x) {    // (1)
17        while (cnt==N) await();                // (1)
18        buf[last] = x;                      // (3)
19        last = (last + 1) % N;              // (3)
20        cnt++;                            // (3)
21        if (cnt==1) notify();              // (3)
22    }
23    public synchronized int remove() {        // (2)
24        while (cnt==0) await();            // (2)
25        int x = buf[first];             // (2)
26        first = (first + 1) % N;        // (2)
27        cnt--;                           // (2)
28        if (cnt==N-1) notify();          // (2)
29        return x;                      // (2)
30    }
31 }

```

図 7.8 Java のモニタ風機構による生産者と消費者問題の解

`synchronized` メソッドはモニタの手続きと同様に、オブジェクトのガードをロックした^{*4}スレッドだけが実行できる。オブジェクトのガードをロックできない場合はガードの待ち行列に入る。

17行 バッファが満杯の場合に `await()` を用いて暗黙の条件変数で待ち状態になる。`await()` は割込みなど別の理由でも終了するので、バッファに空きができるまで繰り返し `await()` を呼び出す。

21行 `notify()` は暗黙の条件変数の `signal()` に相当する。暗黙の条件変数は一つしかないのに、スレッドが `remove()` で待ちになっている可能性ある（直前までバッファが空だった）場合だけ `notify()` するようにしている。無条件に `notify()` を実行するようにすると、`append()` で複数

^{*4} Java ではモニタを所有すると言う。

- のスレッドが待ちになっている場合に、それらを起床させてしまう。
- 24 行 バッファが空の場合に `await()` を用いて暗黙の条件変数で待ち状態になる。
- 28 行 暗黙の条件変数は一つしかないので、スレッドが `append()` で待ちになっている可能性ある（直前までバッファが満杯だった）場合だけ `notify()` するようにしている。
- 29 行 取り出したデータを呼出し側に返す。16 行から 28 行の右端に書いてあるコメントは図 7.4 と同様に、生産者スレッドが 17 行でブロックした後、消費者スレッドが 28 行で生産者スレッドを起床させるときの実行順である。これまでの例と比較して 29 行が異なっている。Java の `notify()` はモニタの `signal()` と異なり、`synchronized` メソッドの最後まで実行した後、スレッドの切換えが起こるからである。

7.7 まとめ

モニタについて学んだ。モニタはスレッド間の同期と相互排除に使用できる「高級言語に組込まれた仕組み」である。モニタ内に資源と、資源を操作する手続きを記述する。資源の相互排除と同期に関する難しい処理がプログラムのあちこちに分散しない。また、モニタ内の手続き（プログラム）の実行は自動的に相互排除されるので、クリティカルセクションを明示する必要もない。

モニタで記述した「生産者と消費者問題」の解を、セマフォを用いて実装し直す例を示した。この例をよく観察するとモニタの動作が細部まで理解できる。

Java 言語はモニタに似た機能をサポートする言語であるが、資源が外部からアクセスできないよう `private` 修飾が必要なこと、条件変数がないこと、`wait()` が `signal()` 以外でも終了すること、`signal()` を実行した後手続きの最後まで実行されること等が異なる。

練習問題

- 7.1 抽象データ型の定義を調べなさい。
- 7.2 図 7.4 のプログラムにおいて、`cnt` なしにキュー（リングバッファ）を記述できるか？
- 7.3 図 7.4 のプログラムにおいて、キューが空のとき一つのスレッドが `remove()` を実行した。その後、別のスレッドが `append()` を実行した。この時の `append()`, `remove()` 内が実行される順を答えなさい。
- 7.4 図 7.4 のプログラムは、複数生産者と複数消費者問題の解に使用できるか？
- 7.5 Java 風仮想言語のモニタを用いてリーダ・ライタ問題の解を示しなさい。
- 7.6 Java 風仮想言語のモニタを用いてセマフォを記述しなさい。
- 7.7 `semBoundedBuffer` (図 7.6, 図 7.7) を実際に実行しなさい。メインルーチンを含むソースプログラムは以下から入手できる。
<https://github.com/tctsigemura/OSTextBook/blob/master/SourceCode/SemBoundedBuffer/>
- 7.8 `signal()` は手続きの最後でしか使用できることにすると、`semBoundedBuffer` (図 7.6, 図 7.7) はどのように簡略化できるか。
- 7.9 `MonBoundedBuffer` (図 7.8) を実際に実行しなさい。メインルーチンを含むソースプログラム

は以下から入手できる。

[https://github.com/tctsigemura/OSTextBook/blob/master/SampleCode/
MonBoundedBuffer/](https://github.com/tctsigemura/OSTextBook/blob/master/SampleCode/MonBoundedBuffer/)

- 7.10 図 7.4 のプログラムと、図 7.8 でコメントに示すように実行順序が異なる。Java のモニタ風機構と従来のモニタのどのような違いによるものか？
- 7.11 その他に従来のモニタと Java のモニタ風機構の違いは何があるか？
- 7.12 モニタの signal と、セマフォの V 操作の違いは何があるか？

第 IV 部

主記憶管理

第 8 章

主記憶（メモリ）

コンピュータシステムにおいて、主記憶（メモリ）^{*1}は CPU と同様に重要な装置である。CPU を仮想化し複数のプロセスを同時に実行可能にするには、主記憶も管理し複数のプロセスに適切に主記憶が割り振られ、かつ、プロセス同士が干渉しないように分離する必要がある。この章では主記憶と主記憶管理の基本的なアイデアについて学ぶ。

8.1 ハードウェア構成

主記憶は CPU がプログラムを実行する際に、プログラムの機械語やデータ、スタック領域等を置くメモリのことである。TeC の主記憶は 256 バイトの RAM 領域であったし、4 年生の実験で使用した H8/3664 では 32KiB の ROM と 2KiB の RAM であった。現代の PC なら 4GiB から 16GiB 程度の大きさを持つ「メモリ」のことである。

本書で前提とするコンピュータのハードウェア構成は図 2.1 に示した。この章では CPU とメモリに着目するので、図を単純化し図 8.1(a) のようなモデルを用いる。この図は CPU がアドレスを指定してメモリのデータを読み書きすることを表している。

プログラム実行時に CPU は以下のようにメモリをアクセスする。

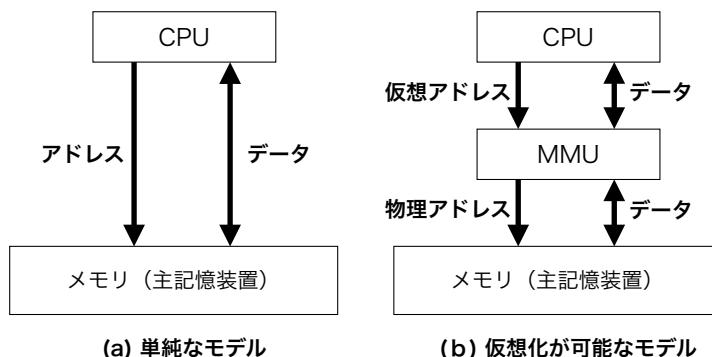


図 8.1 CPU とメモリの関係を表す単純なモデル

^{*1} 本章で「主記憶」と「メモリ」は同じ意味で用いられる。

1. 命令フェッチ (fetch)
PC の値をアドレスとして出力し主記憶からデータ（命令コード）を読む.
2. 命令デコード (decode)
フェッチした命令の種類を調べる.
3. 命令実行 (execution)
命令を実行する際に必要に応じてデータのアドレスを出力し主記憶のデータを読み書きする.

図 8.1(a) のモデルは、 TeC や H8/3664 のようなマイクロコンピュータの様子を表すためには十分である。しかし、この単純なモデルでは現代の本格的なオペレーティングシステムを作動させるには次の点で不十分である。

1. メモリ保護機構がない。
ユーザプロセスが OS のカーネルや、他のプロセスを破壊することを防ぐことができない.
2. メモリの再配置機構がない。
同時に複数のプロセスが主記憶にロードされる環境では、プロセスの起動と終了が繰り返されるうちに使用できない小さなメモリの断片（フラグメント）ができる。フラグメントを解消するために、実行中プロセスをメモリ内で移動する機能が必要である.
3. 仮想記憶機構が実現できない。
メモリより大きなプログラムを実行するために、仮想記憶機構を導入したいができない.

そこで、図 8.1(b) のモデルを用いる。CPU とメモリの間に **MMU (Memory Management Unit : メモリ管理装置)** を追加する。MMU は CPU が出力した仮想アドレスを OS が指示したルールに則り物理アドレスに変換してメモリに送るハードウェアである。OS の主記憶管理プログラムが MMU を制御することによって、使いやすく安全な仮想の主記憶をプロセスに提供する。

8.2 メモリ保護機構

CPU を仮想化したことによって、複数のユーザプロセスをメモリに同時にロードし並列実行することが可能になった。これにより、CPU の使用効率が良くなるだけでなく、コンピュータの使い勝手が非常に良くなった。しかし、ユーザプログラムのバグや悪意によって、OS カーネルや他のユーザプログラムが破壊される可能性がでてきた。OS カーネルはその性質上全てのメモリ領域にアクセスする必要がある。一方でユーザプロセスは自身に割当てられたメモリ以外にアクセスできない仕組みが必要である。

8.2.1 上限・下限レジスタ

プロセスがアクセスしても良いメモリのアドレスの範囲をレジスタに設定し、メモリアクセスする度に CPU が出力するアドレスとレジスタの値を比較する。図 8.2(a) はプロセス 2 が実行中の上限・下限レジスタの状態を表している。図 8.2(b) はアドレスを比較するハードウェアの構成を示している。

1. OS カーネルはプロセスの実行を開始する前に、プロセスの上限ドレスと下限ドレスを上限・下限

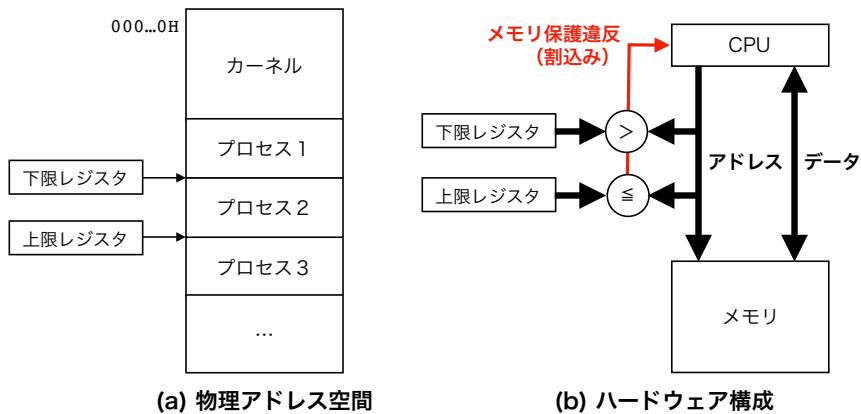


図 8.2 上限・下限レジスタの仕組み

レジスタに設定する。上限・下限レジスタを操作できるのはカーネルモード^{*2}で実行されるカーネルだけである。ユーザプロセスが自身のアクセスできる領域を変更することはできない。

2. カーネルはプロセスの実行を開始させる。
3. プロセスはユーザモードで実行される。ユーザモードで実行中はハードウェアがCPUの出力するアドレスを上限・下限レジスタと比較する。
4. 上限・下限アドレスの範囲外へのアクセスの場合、ハードウェアがメモリアクセスを阻止しCPUに割込みをかける。
5. 割込みが発生するとユーザプロセスの実行が打ち切られ、制御がカーネルに移る。

8.2.2 ロック／キー機構

主記憶をページに分割しページ毎にアクセス許可情報を持たせる。64KiBのメモリを256ページに分割した例を図8.3(a)に示す。16bitのアドレスはページ番号を表す上位8bitと、ページ内オフセットを表す下位8bitに分割される。

図8.3(b)に示すようにCPUは、アドレス、アクセスキー、R/W/XをMMUに出力する。アクセスキーはプロセス毎に決まる数字^{*3}、R/W/Xはメモリアクセスの種類を表す次のどれかである。R(Read)は読み込みを、W(Write)書き込みを、X(exeCute)は命令のフェッチを意味する。

MMUは許可情報表を内蔵している。MMUはCPUが出力したアドレスからページ番号を求め表を引く。表のプロテクションキーがアクセスキーと一致していない場合、または、CPUのR/W/Xが表のアクセスモードに含まれていない場合はメモリ保護違反の割込みを発生する。MMUを操作できるのはCPUの実行モードがカーネルモードの時だけ、MMUがメモリ保護違反の割込みを発生するのはユーザモード時だけである。

特別なプロテクションキー(例えば0)のページは全てのプロセスがアクセス可能とすれば、プロセス間の共有メモリを実現できる。

^{*2} 実行モードは1.2.2で紹介したので忘れた人は再確認すること。

^{*3} プロセス番号でも良い。

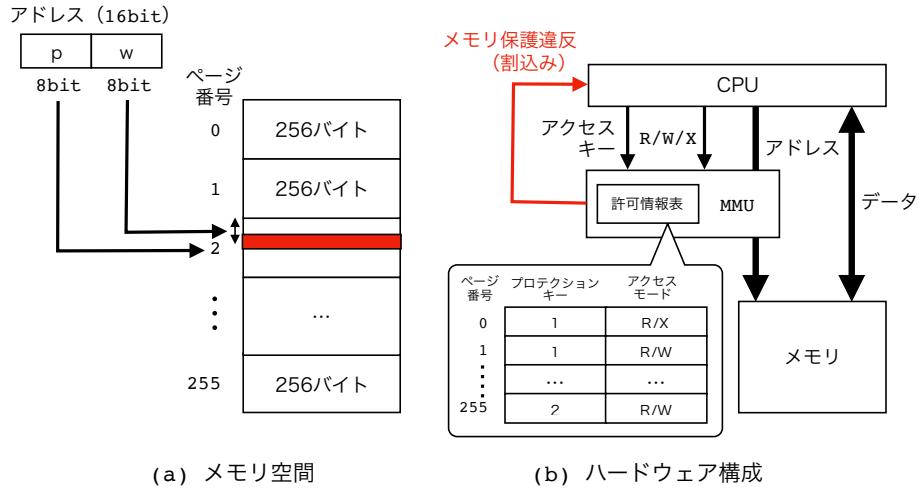


図 8.3 ロック／キー機構の仕組み



図 8.4 プログラムの動的再配置

8.3 プログラムの再配置

コンパイルされたプログラムはメモリにロードされる時にアドレスが確定する。ファイルに格納された実行可能形式プログラムは、ロード時にアドレスを変更できる必要がある。

また、実行途中のプログラムのアドレスを変更することがある。図 8.4 のようにメモリが多くの領域に分断され、領域の間に小さなメモリの断片（メモリフラグメント）が沢山できた場合は、プログラムの詰め合わせ（メモリコンパクション）を行う。実行中のプログラムを移動することを動的再配置と呼ぶ。

8.3.1 再配置可能オブジェクトファイル

プログラミング言語で記述されたプログラムは、コンパイルされ実行可能な機械語ファイルに変換される。しかしコンパイル時には、プログラムがメモリの何番地にロードされるか分からない。そこで、実行可能形式の機械語プログラムはジャンプ先アドレスや、データアドレスの確定をロード時に行うこ

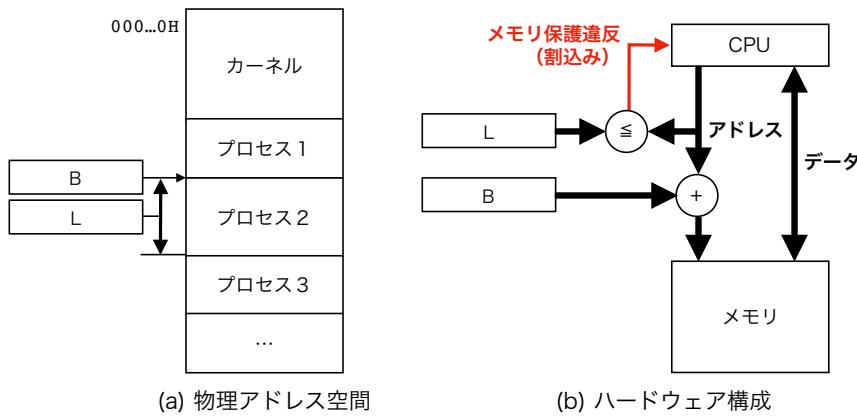


図 8.5 リロケーションレジスタ

とができなければならない。

ロードアドレスが確定しおらず、アドレスを変更可能な機械語プログラムは再配置可能オブジェクト (relocatable object) と呼ばれる。再配置可能オブジェクトファイルは、コンパイル済みの機械語プログラムの他に、ファイル中のどの部分がアドレスであるかを記録した再配置表も含む。プログラムを主記憶にロードする際に再配置表を参照し、プログラムやデータ中の全てのアドレス情報にロードアドレスを足し込む必要がある。例えばプログラムを 1234H 番地にロードすると、JMP 0100H の機械語は JMP 1334H に書換える必要がある。

8.3.2 リロケーションレジスタ

動的再配置を行うためには、実行中のプログラムがどこにアドレスを記憶しているか全て管理する必要がある。しかし、CPU レジスタやスタックに書き込まれたアドレス、リスト構造に含まれるポインタ等、すべてのアドレスデータを追跡することは困難である。

動的再配置を可能にするための一つのアイデアは、リロケーションレジスタと呼ばれる特別なハードウェアを用いることである。図 8.5(a) に示すように^{*4} リロケーションレジスタは、プロセスのロードアドレス (B:Base) と大きさ (L:Limit) を記録するレジスタである。ディスパッチャがプロセスを実行する時に値を設定する。

図 8.5(b) に示すように CPU が出力したアドレスはプロセスの大きさ (L) と比較される。アドレスが L 以上の場合はプロセス領域外のアドレスになるのでメモリ保護違反の割込みを発生する。CPU のアドレスにプロセスのロードアドレス (B) を足した値がメモリアドレスになる。

動的再配置を行うにはプロセスが Running 以外の状態の時に、主記憶上でプロセスのメモリ領域を新しい領域にコピーする。次回プロセスが実行される時、ディスパッチャは新しい領域のアドレスを B にロードする。ユーザプログラムは再配置されたことを知る必要はない。しかし、プロセスの領域の移動は大量のメモリコピーを伴うので、オーバーヘッドが大きい処理である。

^{*4} 図はプロセス 2 を実行するための設定を表している

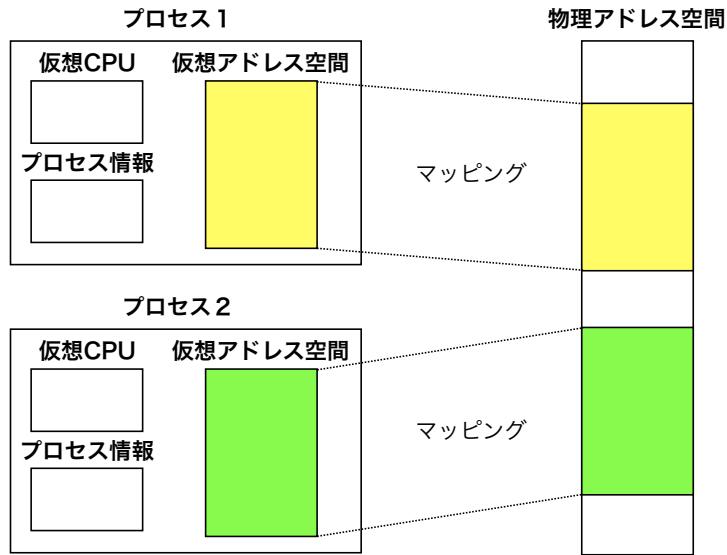


図 8.6 仮想アドレス空間から物理アドレス空間へのマッピング

8.4 アドレス空間の仮想化

図 2.6 で示したように、プロセスは各自が専用の仮想アドレス空間（仮想メモリ空間）を持つ。仮想アドレス空間は仮想アドレスで番地付けされている。それに対しハードウェアとしてのメモリはシステム全体で一つしかない。ハードウェアメモリは物理アドレスで番地付けされており、物理アドレス空間を形成する。図 8.6 にプロセスの仮想アドレス空間が、物理アドレス空間にマッピングされる様子を示す。マッピングは、MMU よる仮想アドレスから物理アドレスへの変換によってなされる。

8.4.1 単一仮想記憶

多重仮想記憶に移行する中間的な形式である。プロセスの仮想アドレスと物理アドレスと同じ方式である。メリットが少ないので通常は次に紹介する多重仮想記憶を用いる。

8.4.2 多重仮想記憶

アドレス空間が仮想化されることにより、全てのプロセスが 0 番地から始まるアドレス空間を持つことが可能になる。プロセス毎に完全に独立したアドレス空間を持つ方式を多重仮想記憶と呼ぶ。実行可能形式のプログラムは、いつも 0 番地にロードされ実行される。再配置可能なオブジェクトでなくとも良い。

8.4.3 仮想アドレス空間の配置

仮想アドレス空間にプログラムや変数を配置する方法はオペレーティングシステムの種類により一定ではない。図 8.7 に UNIX の例を示す。

- 初期化済みのグローバル変数^{*5}は、初期化データ領域（data セグメント）に配置される。
- 初期化されないグローバル変数^{*6}は、非初期化データ領域（bss セグメント）に配置される。

^{*5} 正確には初期化済みの静的な変数。関数内で `static` 修飾した変数も含まれる。

^{*6} 正確には初期化されない静的な変数。関数内で `static` 修飾した変数も含まれる。

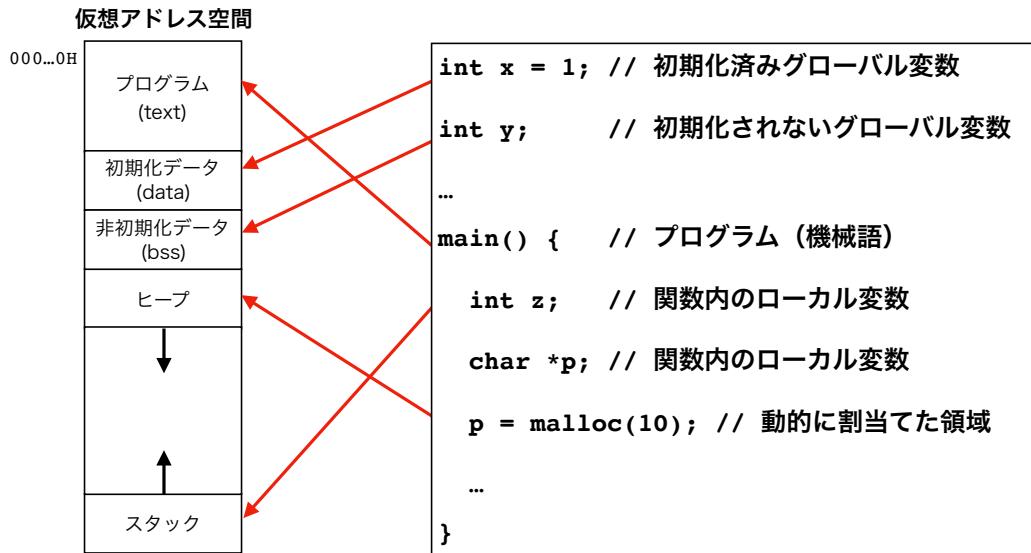


図 8.7 仮想アドレス空間の配置例

- `main()` 関数は機械語に変換され、0番地から始まるプログラム領域 (`text` セグメント) に配置される。
- 関数のローカル変数^{*7}は、関数の実行開始時にスタックまたはCPUレジスタに割り付けられ、関数を終了する時に破棄される。同じスタックを関数呼び出しのためにCALL 機械語命令も使用する。スタックは、必要に応じて仮想アドレス空間を0番地側に伸びていく。
- `malloc` 関数等を用いて動的に領域を割り当てるときヒープが使用される。ヒープは必要に応じて仮想アドレス空間を0番地とは逆の方向に伸びていく。

^{*7} 正確には自動変数。関数内で `static` 修飾した変数は含まれない。

リスト 8.1 C 言語プログラムを TaC の機械語に変換した例

```
_x    DW      1      // int x = 1;
_y    WS      1      // int y;
_main PUSH    FP      // void main() {
    LD     FP,SP
    PUSH   G3      // int z;
    PUSH   G4      // char *p;
    LD     G0,#10  //
    PUSH   G0      //
    CALL   _malloc // p = malloc(10);
    ADD    SP,#2   //
    LD     G4,G0   //
    POP    G4
    POP    G3
    POP    FP
    RET        // }
```

第9章

メモリ割付け方式

プロセスが実行を開始する前に、物理メモリの一部をプロセスに割付けプログラムをロードする。物理メモリを複数のプロセスで分割し利用するために幾つかの方式が考案されてきた。ここでは、固定分割方式と可変分割方式について解説する。

9.1 固定区画方式

予めメモリを大小数種類の区画に分割しておく。プロセスのサイズにより適切な区画を選択し利用する。その様子を図 9.1 に示す。

図 9.1 の例では利用可能なメモリは予め五つの区画に分割されている。プロセス 1 から 4 を実行したい時、ロード可能な区画を選択しロードする。プロセス 4 はロード可能な区画が無いので実行できない。

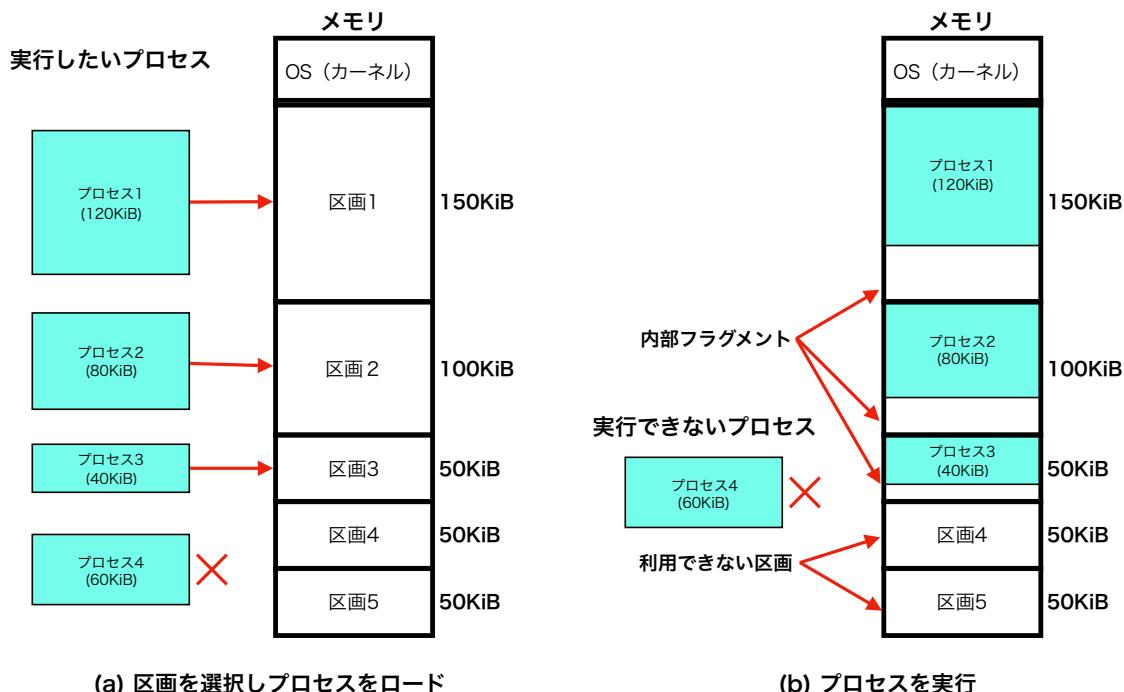


図 9.1 固定区画方式

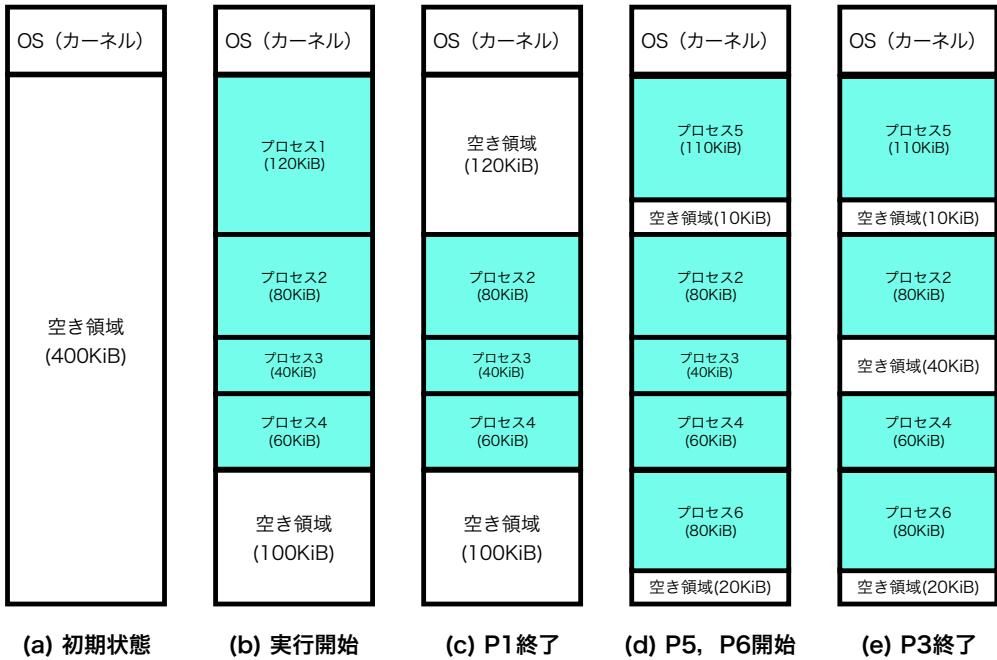


図 9.2 可変区画方式

区画の大きさとプロセスの大きさは一致するとは限らない。一致しない場合は区画 1 から 3 ように、内部に使用されない領域（内部フラグメント）が生じる。区画 4 と区画 5 を合わせるとプロセス 4 をロード可能であるが、固定区画方式では区画 4 と区画 5 は利用できない。仕組みが簡単だがメモリの利用率が低い。特徴を以下にまとめると。

1. 空き領域の管理が容易である。
2. 領域内部に無駄な領域（内部フラグメント）が生じる。
3. 小さな領域が複数空いていても大きなプロセスは実行できない。
4. 実行可能なプロセスのサイズに強い制約がある。
(図の例では、151KiB のプロセスは実行できない。)
5. 同時に実行できるプロセスの数に制約がある。
(図の例では、同時に五つ以上のプロセスは実行できない。)

9.2 可変区画方式

メモリの空き領域から、プロセスのサイズに合わせたメモリ区画を割付ける方式である。図 9.2 に模式図を示す。

(a) 初期状態

メモリはカーネルと、大きな単一の空き領域に分割される。

(b) 実行開始

図 9.1 と同じ四つのプロセスが実行をロードし実行を開始した。図 9.1 の例では実行できなかった

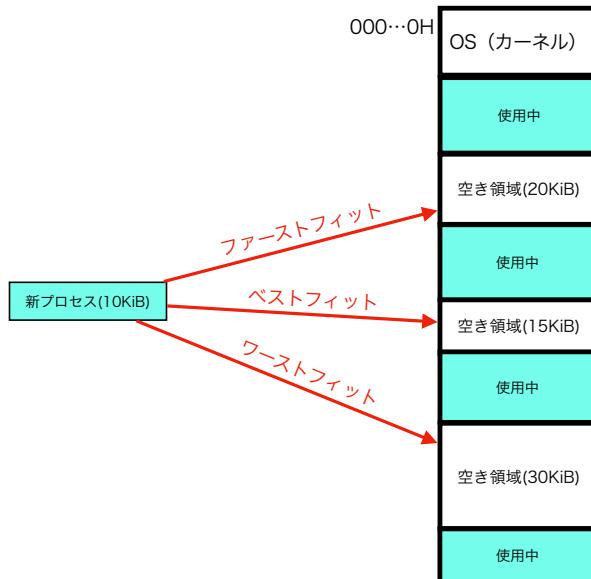


図 9.3 空き領域の選択方式

「プロセス 4」も実行できる。(メモリの利用効率は良い。)

(c) プロセス 1 (P1) 終了

終了したプロセスが利用していた領域は再利用可能な空き領域になる。

(d) プロセス 5 (P5), プロセス 6 (P6) 実行開始

120KiB の空き領域は、「110KiB の領域」と「10KiB の空き領域」に分割する。100KiB の空き領域は、「80KiB の領域」と「20KiB の空き領域」に分割する。プロセス 5 とプロセス 6 を新しい領域にロードし実行する。

(e) プロセス 3 (P3) 終了

プロセス 3 が利用していた領域は再利用可能な 40KiB 空き領域になる。メモリ全体では、10KiB, 40KiB, 20KiB の空き領域ができている。

以上のように可変区画方式では、プロセスの開始と終了が繰り返されるに従い小さな空き領域ができる。このような区画の外にできる小さなメモリ領域を外部フラグメントと呼ぶ。

9.3 可変区画方式の空き領域選択方式

以下の三つの方式が知られている。図 9.3 に三つの方式で選択される空き領域の例を示す。

- ファーストフィット (first-fit) 方式

アドレス順に空き領域を探索し、最初に見つかった十分な大きさの領域を選択する。

- ベストフィット (best-fit) 方式

プロセスを格納可能な領域の中で最小のものを選択する。

- ワーストフィット (worstfit) 方式

最も大きな領域を選択する。

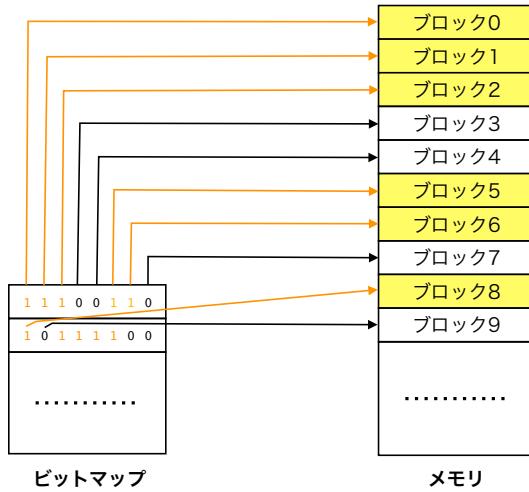


図 9.4 ビットマップ方式

シミュレーションの結果メモリ利用率の点で、ワーストフィット方式は最も性能が劣るが、ファーストフィットとベストフィットの性能は互角だと言われている。しかし実行時間の点で、ファーストフィットがベストフィットより優れていると言われている。

9.4 空き領域の管理方式

プロセスによって使用中のメモリ区画はプロセスの PCB 等に記録しあけば見失う心配はない。しかし、どのプロセスにも属さない空き領域はメモリ管理側で記録しておく必要がある。

- ビットマップ (bitmap) 方式

図 9.4 のようにメモリを一定の大きさのブロックに分割し、1 ブロックをビットマップの 1 ビットに対応させる。ビットが 0 ならブロックが空き状態、ビットが 1 なら使用中の意味になる。

ビットマップはメモリ上に記録する。ビットマップの大きさは次のように計算できる。仮に 8GiB のメモリを 4KiB のブロックに分割して管理すると仮定すると、ブロックの総数は $8GiB \div 4KiB = (8 \times 2^{30}) \div (4 \times 2^{10}) = 2 \times 2^{20}$ 個となる。ビットマップの大きさはブロック数と同じ 2×2^{20} ビットになる。これをバイト単位に換算すると、 $(2 \times 2^{20}) \div 8 = 2^{18} = 256KiB$ となる。

ビットマップに使用するメモリは無視できるほど小さいものではない。ビットマップをより小さくするにはブロックサイズを大きくすれば良い。しかし、ブロックサイズを大きくすると内部フラグメントが大きくなる。

- リスト (linked-list) 方式

空き領域をリストにして管理する方式である。使用中の領域が解放されると空き領域リストに追加される。解放される領域が、別の空き領域に隣接している場合は一つの空き領域になる。その様子を図 9.5 に示す。

リスト方式で用いるデータ構造の例を図 9.6 に示す。新しい空き領域と前後の空き領域をマージする処理が簡単に行えるように、空き領域はアドレス順にソートしてリストに挿入される。

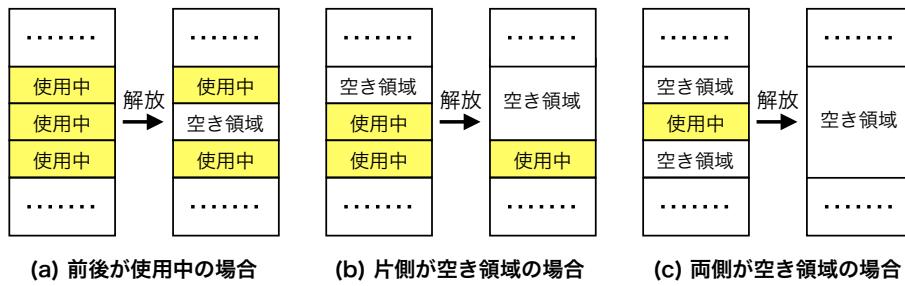


図 9.5 領域開放時に空き領域を連結する様子

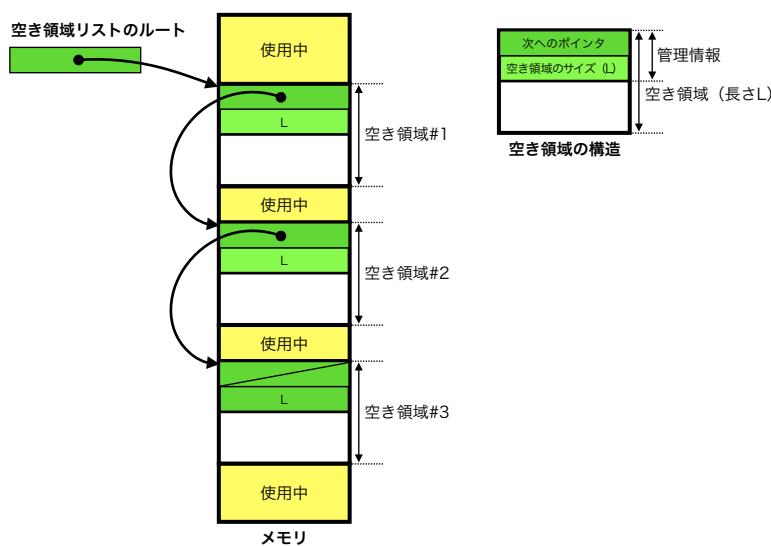


図 9.6 空き領域リスト

アドレス順に領域がソートしてあると、ファーストフィット方式で領域を探索するためにも適している。ベストフィット方式の場合は領域サイズ順にソートしてあると良いが、前述の空き領域のマージ処理には適さない。

練習問題

可変区画方式で管理される 100KiB の空き領域がある時、次の順序で領域の割付け解放を行った。ファーストフィット方式を用いた場合とベストフィット方式を用いた場合について、実行後のメモリマップを図示しなさい。

1. 30KiB の領域を割付け
2. 40KiB の領域を割付け
3. 20KiB の領域を割付け
4. 先程割付けた 40KiB の領域を解放
5. 10KiB の領域を割付け

第 10 章

メモリ割付けプログラムの例

TacOS のメモリ管理プログラムを例に、ファーストフィット方式を用いる可変区画方式のメモリ管理プログラムを紹介する。以下で紹介するプログラムは C-- 言語で記述してある。また、プログラムのオリジナル版は、<https://github.com/tctsigemura/TacOS/blob/master/os/mm/mm.cmm> から入手可能である。このプログラムは、OS 内部でプロセスの生成時に、プロセスの領域を割り付けるために使用されるものである^{*1}。

10.1 データ構造の初期化

メモリ管理用構造体 (MemBlk) の宣言と、初期化プログラムをリスト 10.1 に示す。変数 memPool と番兵 (MemBlk 構造体) は、OS カーネルがメモリにロードされた時点で、OS カーネルのデータ領域に初期化された状態で置かれる。_end() 関数は OS カーネルが使用している領域の最後のアドレス（空き領域の先頭のアドレス）を知るために用いる特殊な関数である。mmInit() 関数は最初に一度だけ実行されデータ構造の初期化を行う。mmInit() 関数は、まず、空き領域の先頭部分を MemBlk 構造体と見做し番兵の next がこの構造体を指すようにする（15 行）。次に空き領域サイズを計算し、この構造体の size に代入する（16 行）。最後にこの構造体がリストの最後になるように next に null を代入する（17 行）。

以上の初期化処理が完了した時点のデータ構造を図 10.1 に示す。このとき、memPool 変数を起点に番兵付きで長さが 1 の空き領域リストが完成している。

OS カーネルはメモリの 0000 番地から配置されている。この領域に OS カーネルのプログラムとデータがロードされる。図では分かりにくいが、memPool 変数と番兵はこの領域に配置される。最初は、OS カーネルの直後からメモリ最後の使用不可領域の直前までが一つの空き領域になっている。空き領域の先頭に MemBlk 構造体を置いたと見做し、番兵がそこを指すように初期化する。空き領域の先頭の MemBlk 構造体が空き領域サイズを記憶し、E000H 番地まで空き領域が続いていることを表現している。

^{*1} 本当は、プロセス生成以外でも使用されている。

リスト 10.1 データ構造と初期化

```

1 #define MBSIZE sizeof(MemBlk) // MemBlk のバイト数
2 #define MAGIC (memPool) // 番兵のアドレスを使用する
3
4 // 空き領域はリストにして管理される
5 struct MemBlk { // 空き領域管理用の構造体
6     MemBlk next; // 次の空き領域アドレス
7     int size; // 空き領域サイズ
8 };
9
10 // メモリ管理の初期化
11 MemBlk memPool = {null, 0}; // 空き領域リストの番兵
12 public int _end(); // カーネルの BBS 領域の最後
13
14 void mmInit() { // プログラム起動前の初期化
15     memPool.next = _ItToA(addrOf(_end)); // 空き領域
16     memPool.size = 0xe000 - addrOf(_end); // 空きメモリサイズ
17     memPool.next.next = null;
18 }

```

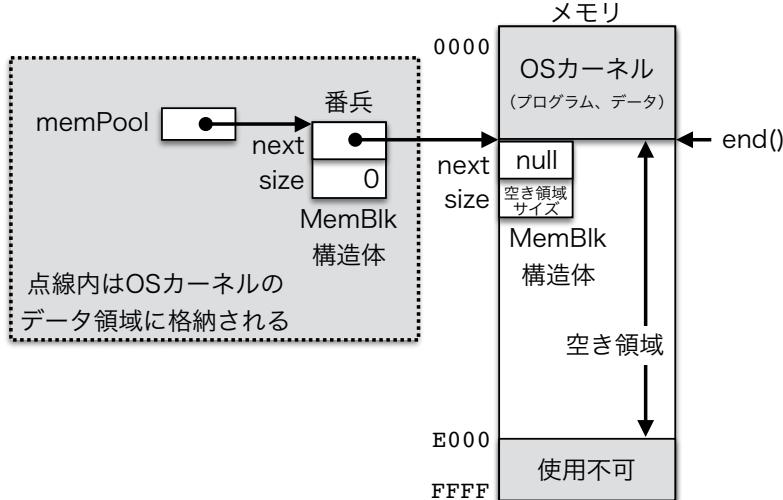


図 10.1 初期化直後のデータ構造

10.2 メモリの割り付け

メモリ領域の割り付けは `mmAlloc()` 関数が行う。`mmAlloc()` 関数は引数に与えられたバイト数の領域を割り付け、領域の先頭アドレスを返す。リスト 10.2 の手順で 1KiB の領域を三つ割り付けた状態を図 10.2 に示す。`mmAlloc()` は、空き領域の前の方に要求された大きさの領域を割り付ける。リスト 10.2 では `mmAlloc()` が 3 回実行され、メモリの先頭に 1KiB の使用中領域を三つ割り付けている。そ

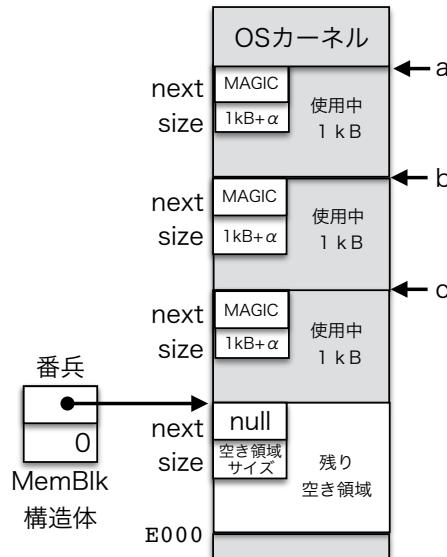


図 10.2 三つの領域を割り付けた状態

の結果、空き領域が小さくなっている。

リスト 10.2 データ構造と初期化

```

1 a = mmAlloc( 1024 );      // 1KiB の領域を割り付ける
2 b = mmAlloc( 1024 );      // 1KiB の領域を割り付ける
3 c = mmAlloc( 1024 );      // 1KiB の領域を割り付ける

```

リスト 10.3 に `mmAlloc()` 関数の本体を示す。`mmAlloc()` 関数は領域の要求サイズ（2 行目の `size`）を引数に呼び出され、割り付けた領域のアドレスを整数（`int` 型）で返す。実際に割り付ける領域は、要求されたサイズに `MemBlk` 構造体のサイズを加えた後に偶数に切上げた大きさである（3 行目）。TaC では 16bit データ（2 バイトデータ）をメモリに格納する時、連続した $2i$ 番地と $2i + 1$ 番地（ i は適当な整数）を使う決まりになっているので、領域は常に偶数番地から始まるようにする必要がある^{*2}。

`mmAlloc()` 関数は、空き領域リストを探索し割り付けるサイズ以上の領域を見つける（7 行～）。サイン比較に使用される `_uComp()` 関数は符号なし整数用の大小比較関数である。領域を探す間はポインタ `m` が目的の領域を、ポインタ `p` が一つ前の領域を指している。リストの最後に達した場合は、適切な領域が見つかることになる。`mmAlloc()` 関数は 0 を返して終了する。

適切な領域（`m`）が見つかったら、それを使用中領域と空き領域に分割するべきか判断する（13 行）。ちょうどピッタリか少しだけ大きい領域なら分割しない。分割しない場合は領域をリストから外す（16 行）。

分割する場合は領域の前半（`m`）を使用中、残りを空き領域とする。空き領域のアドレスはアドレス用の足算関数 `_addrAdd()` で `n` に求める（18 行）。領域 `m` をリストから外し代わりに領域 `n` をリストに挿入する（19 行～21 行）。領域 `n` の大きさを設定する（22 行）。

^{*2} 割り当てた領域に 2 バイトデータの配列として使用される場合を想像して欲しい。

最後に、領域 n が正当に割当てられたことを表すマジックナンバー (MAGIC) を next に書込む (24 行). mmAlloc() 関数が返すアドレスは MemBlk 構造体直後である (25 行). MAGIC は mmFree() 関数が領域を解放する時に、正当に割当てられた領域かどうかチェックするために使用される.

リスト 10.3 メモリ割り付けプログラム

```

1 // メモリを割り付ける
2 int mmAlloc(int siz) {                                // メモリ割り当て
3     int s = (siz + MBSIZE + 1) & ~1;                  // 制御データ分大きい偶数に
4     MemBlk p = memPool;                            // 直前の領域
5     MemBlk m = p.next;                           // 対象となる領域
6
7     while (_uCmp(m.size,s)<0) {                   // 領域が小さい間
8         p = m;                                     // リストを手繰る
9         m = m.next;
10        if (m==null) return 0;                      // メモリが不足する場合は
11        // エラーを表す null ポインタ
12    }
13
14    if (_uCmp(m.size ,s+MBSIZE+2)<=0) {           // 分割する価値がない領域サイズ
15        if (memPool.next==m && m.next==null)       // リストの長さがゼロにならない
16            return 0;                               // ようにする
17        p.next = m.next;                          // リストから外す
18    } else {                                    // 領域を分割する価値がある
19        MemBlk n = _addrAdd(m, s);                // 残り領域
20        n.next = m.next;
21        n.size = m.size - s;
22        p.next = n;
23        m.size = s;
24    }
25    m.next = MAGIC;                            // マジックナンバー格納
26    return _AtoI(_addrAdd(m, MBSIZE));          // 管理領域を除いて返す
}

```

10.3 メモリの解放

`mmFree(b);` を実行し領域 b を開放した状態を図 10.3 に示す. 続けて `mmFree(c);` 実行し領域 c も開放した状態を図 10.4 に示す. 図 10.3 では、領域 b が開放され空き領域が二つになり、空き領域リストの長さが 2 になっている. 図 10.4 では、領域 c が開放され空き領域を一つに合体することができたので、空き領域リストの長さが 1 になっている.

メモリの解放を行う `mmFree()` 関数の本体をリスト 10.4 に示す. `mmFree()` 関数は解放する領域 mem を引数に実行される (2 行). 領域の MemBlk 構造体に MAGIC が格納されていない場合は `mmAlloc()` 関数で割り付けられた正当な領域では無いのでエラーを表示してシステムを停止する (8 行)^{*3}.

^{*3} このプログラムは OS 内部で動くものである. このような事象が発生するのは OS のバグが原因と考えられるのでシステ

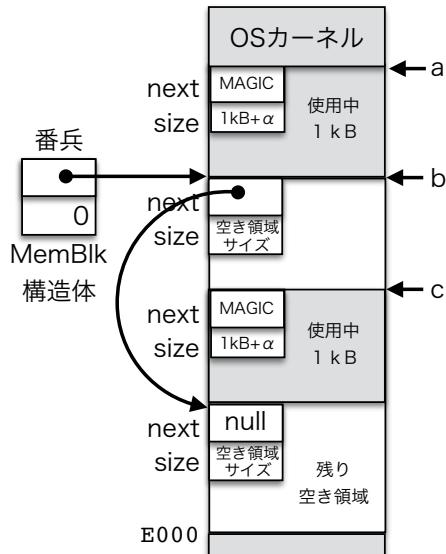


図 10.3 領域 b を開放した状態

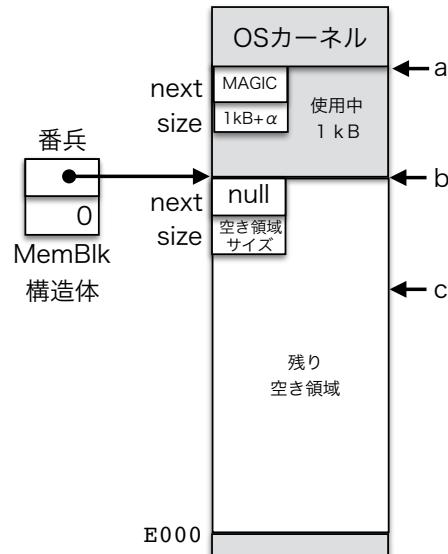


図 10.4 領域 c も開放した状態

解放された領域は新しい空き領域になる。空き領域リストがアドレス順になるように、新しい空き領域を挿入すべき位置を探す処理を行う（10 行～）。

解放する領域が直前の空き領域に重なっていたり、直後の空き領域に重なっていたりしていないかチェックしている（19 行）*4。

解放する領域が直前の空き領域に隣接している場合は、直前の空き領域サイズを大きくすることで一つの空き領域にしてしまう（23 行）。直後の空き領域とも隣接している場合は、直前の空き領域サイズを更に大きくし直後の空き領域も一つの領域にする（25 行）。この時は、空き領域が一つにまとめられたので、空き領域リストから直後の空き領域を削除する（26 行）。

更に、直後の空き領域だけと隣接している場合（28 行）、どの空き領域とも隣接していない場合（32 行）の処理が続いている。

*4 ムを停止する。

*4 これも OS のバグ以外では発生しない。OS 自身がバグを含んでいないかチェックする機会として利用している。

リスト 10.4 メモリ解放プログラム

```
1 // メモリを解放する
2 int mmFree(void[] mem) {                                // 領域解放
3     MemBlk q = _addrAdd(mem, -MBSIZE);                  // 解放する領域
4     MemBlk p = memPool;                                  // 直前の空き領域
5     MemBlk m = p.next;                                 // 直後の空き領域
6
7     if (q.next!=MAGIC)                                // 領域マジックナンバー確認
8         badaddr();
9
10    while (_aCmp(m, q)<0) {                           // 解放する領域の位置を探る
11        p = m;
12        m = m.next;
13        if (m==null) break;
14    }
15
16    void[] ql = _addrAdd(q, q.size);                   // 解放する領域の最後
17    void[] pl = _addrAdd(p, p.size);                   // 直前の領域の最後
18
19    if (_aCmp(q,pl)<0 || m!=null&&_aCmp(m,ql)<0) // 未割り当て領域では？
20        badaddr();
21
22    if (pl==q) {                                       // 直前の領域に隣接している
23        p.size = p.size + q.size;
24        if (ql==m) {                                   // 直後の領域とも隣接している
25            p.size = p.size + m.size;
26            p.next = m.next;
27        }
28    } else if (ql==m) {                               // 直後の領域に隣接している
29        q.size = q.size + m.size;
30        q.next = m.next;
31        p.next = q;
32    } else {
33        p.next = q;
34        q.next = m;
35    }
36    return 0;
37 }
```

第 11 章

セグメンテーション

プロセスが使用するメモリ領域のサイズを動的に変化させたり、領域ごとに異なる性質を持たせたりすることが可能な、より高度なメモリ管理手法であるセグメンテーションを紹介する。

11.1 リロケーションレジスタ方式の問題点

ユーザは図 8.7 のように仮想アドレス空間にプログラムやデータを配置する。既に学んだリロケーションレジスタを用いた方法では、仮想アドレス空間は図 8.6 のようにメモリの連続領域にマッピングされる。この方式は以下の問題点を持っている。

- 必要なメモリの見積もりが難しい。
十分な大きさの仮想アドレス空間を準備しないと、実行時にヒープ領域やスタック領域が不足する可能性がある。しかし無闇に大きくするとヒープ領域とスタック領域の間が広くなりすぎメモリが無駄になる。実行前に必要なメモリの大きさを見積もる必要があり使い勝手が悪い。
- 領域の性質応じたメモリ保護ができない。

リロケーションレジスタを用いて他プロセスやカーネルメモリは保護可能である。しかし、プロセスが自身の領域を適切に使用することを強制できない。例えば、次のようなメモリ保護が望まれる。

- プログラム領域は機械語プログラムと定数データだけを格納しているので、読み出しと実行だけ許可する。
- データ、ヒープ、スタック領域に機械語プログラムは置かないので、データの読み出しと書き込みだけ許可する。

11.2 セグメント

プロセスに複数のアドレス空間を持たせることで前記の問題を解決する。複数持つことができるアドレス空間のことをセグメントと呼ぶ。図 11.1 に複数のセグメントが存在する仮想アドレス空間の例を示す。プロセスの仮想アドレス空間は、セグメント番号とセグメント内アドレスの二つでアドレス付けされる。プロセスの仮想アドレス空間が二次元になった。

図 11.1 は以下のことを表している。プログラム、データ、ヒープ、スタックの領域を番号付けされたセグメントにした。セグメントに付記した `rwx` はセグメントの保護モードを表している。セグメン

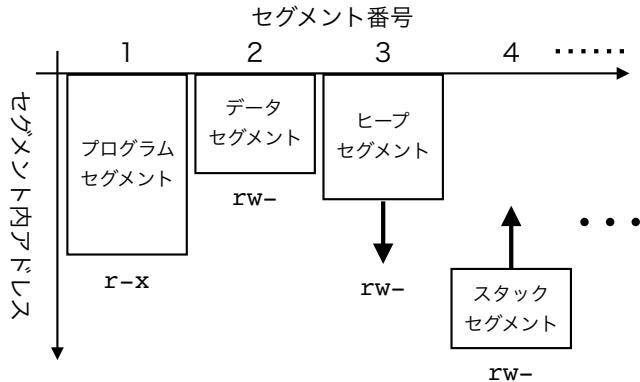


図 11.1 セグメントからなる仮想アドレス空間

トの大きさは内容の大きさとぴったり同じサイズにできる。ヒープ領域は実行時に必要に応じて長くすることができる。スタックが前に向かって伸びる場合は、前向きに伸びるセグメントが使いやすい。IA-32^{*1}等では前に向かって伸びるセグメントもサポートされている [41]。

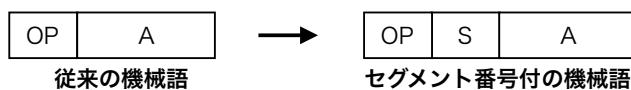
複数のセグメントを用いることで、仮想アドレス空間内の領域配置の問題から解放された。

11.3 セグメント番号

仮想アドレスにセグメント番号が新たに必要になった。セグメント番号を供給するために CPU に変更が必要になる。

命令コード

機械語命令コードを変更し、セグメント番号を含める方法が考えられる。各命令がセグメント番号のために大きくなるので、プログラムサイズが大きくなる。



カレントセグメントレジスタ

CPU 内部に現在のセグメント番号を格納するレジスタを置く方式である。別のセグメントへプログラムをジャンプさせるセグメント間ジャンプ命令 (JMPS と仮に命名する)、セグメント間コール命令 (CALLS と仮に命名する)、セグメント間リターン命令 (RETS と仮に命名する) がカレントセグメントレジスタに新しいセグメント番号をロードする。

図 11.2 に模式図を示す。まず、セグメント 1 のプログラムが実行される。この時点ではカレントセグメントはセグメント 1 なので、LD G0,A はセグメント 1 内のデータ A を参照する。

CALLS 3:0 はセグメント 3 の 0 番地に配置されたサブルーチンを呼び出す。その際、カレントセグメントレジスタの値とプログラムカウンタの値がスタックに保存される。その後、カレントセグメントはセグメント 3 に、プログラムカウンタは 0 に変更される。サブルーチン実行中のカレントセグメントはセグメント 3 なので、サブルーチン中の ADD G0,A はセグメント 3 のデータ A を参照する。

^{*1} 32bit パーソナルコンピュータで広く使われてきたインテル社 CPU のアーキテクチャのことである。

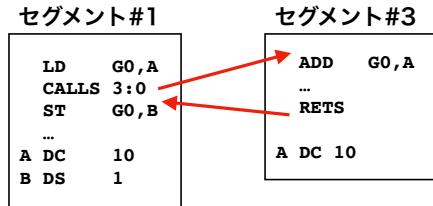


図 11.2 セグメント間のサブルーチンコール

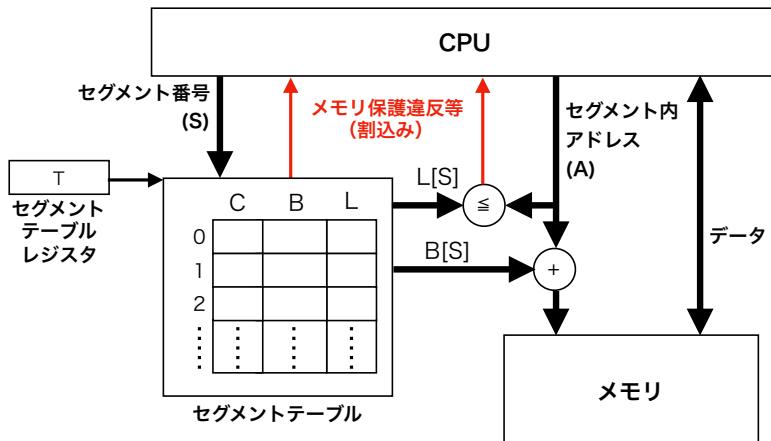


図 11.3 セグメンテーション機構

RETS が実行されるとカレントセグメントレジスタとプログラムカウンタの値がスタックから復元され、セグメント 1 のプログラムの実行が再開される。

IA-32 では、プログラム用 (CS), データ用 (DS), スタック用 (SS) 等、数個のカレントセグメントレジスタを持つ。機械語命令のフェッチには CS、データのアクセスには DS、スタックの操作には SS が暗黙の内に使用される [42]。

11.4 セグメンテーション機構

セグメンテーション機構の模式図を図 11.3 に示す。CPU が output したセグメント番号 (S) とセグメント内アドレス (A) の組を、セグメントテーブルを使用して物理アドレスに変換する。

11.4.1 セグメントテーブル

現在のプロセスが使用できるセグメントの一覧表である。表の一行 (エントリー) が一つのセグメントを表現する。B (Base) フィールドはセグメントの物理アドレス、L (Limit) フィールドはセグメントサイズであり、セグメントテーブルのエントリはリロケーションレジスタと同様な内容を含んでいる。

エントリの C (制御) フィールドは、表 11.1 のビットを含んでいる。V (Valid) ビットが 0 の場合、そのセグメントはメモリ上に存在しない。存在しないセグメントをアクセスしようとするとセグメント不在割込みが発生する。

D (Dirty) ビットはセグメントの内容がメモリにロードされた後に変更されたことを記録する。メモリが不足してセグメントをスワップアウトする際、D ビットが 0 ならセグメントを二次記憶装置へ書

表 11.1 セグメントテーブル C フィールドの例

名称	ビット数	意味
V (Valid)	1	メモリにロードされている。
D (Dirty)	1	ロード後に変更された。
RWX (Read/Write/eXecute)	3	許されるアクセス方法。

き戻す必要がない。

RWX (Read/Write/eXecute) の三ビットはセグメントに対して行って良い操作を表す。CPU が許可されていない操作を行うとメモリ保護違反の割込みが発生する。

図 11.3 では、セグメントテーブルが専用のハードウェアとして描かれているが、セグメントテーブルはメモリ上に置かれる。セグメントテーブルのアドレスはセグメントテーブルレジスタが記憶している。プロセスを切り換える際は、そのプロセスのセグメントテーブルを指すようにセグメントテーブルレジスタを書き換える。

11.4.2 物理アドレスへの変換

CPU が output したセグメント番号 (S) とセグメント内アドレス (A) の組は、以下の手順で物理アドレスに変換される。

1. セグメントテーブルのエントリ読み出し

セグメントテーブルは、セグメントテーブルレジスタによって示されるメモリ上のアドレスに配置されている。セグメント番号をセグメントテーブルのインデクスとして使用し、セグメントテーブルの一つのエントリをメモリから読み出す。

2. C フィールドのチェック

読み出したエントリの C フィールドを調べ、セグメントがメモリにロードされていない場合や、許可されていない種類 (RWX) のアクセスを CPU が行おうとしている場合は、割込みを発生する。

3. セグメント内アドレスのチェック

読み出したエントリの L フィールド ($L[S]$) と CPU が output したセグメント内アドレス (A) を比較する。 $L[S]$ はセグメントのサイズを表すので、A が $L[S]$ 以上の場合にはセグメント内アドレスがセグメントの後端を越えている。メモリアクセスを阻止した上で割込みを発生する。

4. 物理アドレスの計算

読み出したエントリの B フィールド ($L[S]$) と CPU が output したセグメント内アドレス (A) の和を求める。和が物理アドレスである。

11.4.3 セグメントテーブルエントリのキャッシング

前記の物理アドレスへの変換手順では、メモリアクセスの度にセグメントテーブルを参照していた。セグメントテーブルの参照はメモリアクセスなので、メモリアクセス回数が二倍になる。他の CPU や I/O 装置もメモリを使用するので、メモリへのアクセスは混み合っている。メモリアクセス回数は少なくてすべきである。

一方で、同時に使用されるセグメントの数は多くないので、必要なテーブルエントリを CPU や

セグメントレジスタ		裏レジスタ		
CS	セレクタ	ベース	リミット	属性
DS	セレクタ	ベース	リミット	属性
SS	セレクタ	ベース	リミット	属性
ES	セレクタ	ベース	リミット	属性
FS	セレクタ	ベース	リミット	属性
GS	セレクタ	ベース	リミット	属性

図 11.4 IA-32 のセグメントレジスタと裏レジスタ

MMU にキャッシングすることは容易である。例えば IA-32 では、カレントセグメントレジスタ (CS, DS, SS 等) 毎に、セグメントテーブルエントリのコピーを裏レジスタ [43] に持つ。カレントセグメントレジスタの値が変更された時、自動的に裏レジスタにエントリがコピーされる。図 11.4 にセグメントレジスタと裏レジスタの関係を示す。裏レジスタはハードウェアが自動的に使用し、プログラムからは見えない。

11.5 セグメンテーション機構による仮想記憶

プログラム実行中に必要なセグメントだけをメモリにロードするようにする。これにより、全体がメモリに収まらないような大きなプログラムも実行できる。メモリより大きなプログラムが実行できる点で、メモリが仮想化されたと言うことができる。仮想化されたメモリのことを仮想記憶と呼ぶ。

11.5.1 スワップイン

セグメントテーブルの V ビットが 0 のセグメントを参照すると、セグメント不在割込みが発生する。プロセスがセグメント不在割込みを発生するとオペレーティングシステムに制御が移る。

オペレーティングシステムは、まず、割込みの原因になったセグメントを二次記憶装置から読み込む(スワップインする)。次に、セグメントテーブルを書き換える。最後に割込みを発生した命令の再実行から再開するようにプロセスをディスパッチする。

11.5.2 スワップアウト

オペレーティングシステムがセグメントをスワップインする際にメモリが不足するかも知れない。その場合、オペレーティングシステムは適切なセグメントを選択し二次記憶に追い出し(スワップアウトし)、メモリに空きを作らなければならない。今後、使用されそうに無いセグメントを選択すると良いが、どのセグメントが該当するか判断することは難しい問題である。

図 11.5 にセグメントがスワップアウト／スワップインされる様子を示す。図は新しくセグメント 3 が必要になりスワップインする様子を表している。セグメント 3 をロードするためにはメモリが不足するので、まず、使用頻度が低いセグメント 1 をスワップアウトし、次に、セグメント 3 をスワップインする。

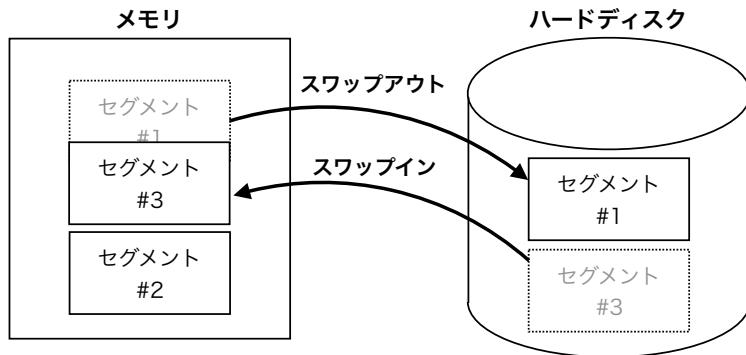


図 11.5 セグメントのスワッピング

11.6 セグメントの共用

プロセス間でセグメントを共用することでメモリの節約ができる。プログラムや定数等を格納し、書き込みが禁止のセグメントは複数のプロセスで共用できる。逆に、書き込みが許可されているデータ、ヒープ、スタックセグメント等は共用できない。また、セグメントの共用を積極的に利用し、プロセス間の共有メモリも実現できる。

図 11.6 に三つのプロセスがセグメントを共用している様子を示す。

- C 言語ライブラリ (`printf()` 等を含む) は、C 言語で使用する関数を提供する。ライブラリはプログラムだけを格納し変更されないとすると、全てのプロセスで共用することができる^{*2}。
- プロセス 1 とプロセス 2 はどちらも emacs を実行している。プログラムは変更されないので、プロセス 1 とプロセス 2 で「emacs プログラムセグメント」を共用できる。
- プロセス 3 は `a.out` を実行している。プログラムは変更されないが、同じプログラムを実行中のプロセスが存在しないので「`a.out` プログラムセグメント」は共用はできない。
- プロセスが書き換えるデータセグメントやスタックセグメントは、プロセス毎に内容が異なるので共用できない。別々のデータセグメントやスタックセグメントが必要になる。プロセス 1 とプロセス 2 はどちらも emacs を実行しているが、編集している文書が異なるのでデータセグメントの内容も異なるハズである。

11.7 セグメンテーションの利点・欠点

セグメンテーションの利点と欠点を以下にまとめる。

利点

- セグメントには、例えば「C 言語ライブラリセグメント」のような、論理的な意味を持たせることができる。
- セグメントの論理的な意味を反映したメモリ保護が可能である。

^{*2} 本当は、ライブラリが使用するグローバル変数（例えば `errno`）をどうするか問題である。

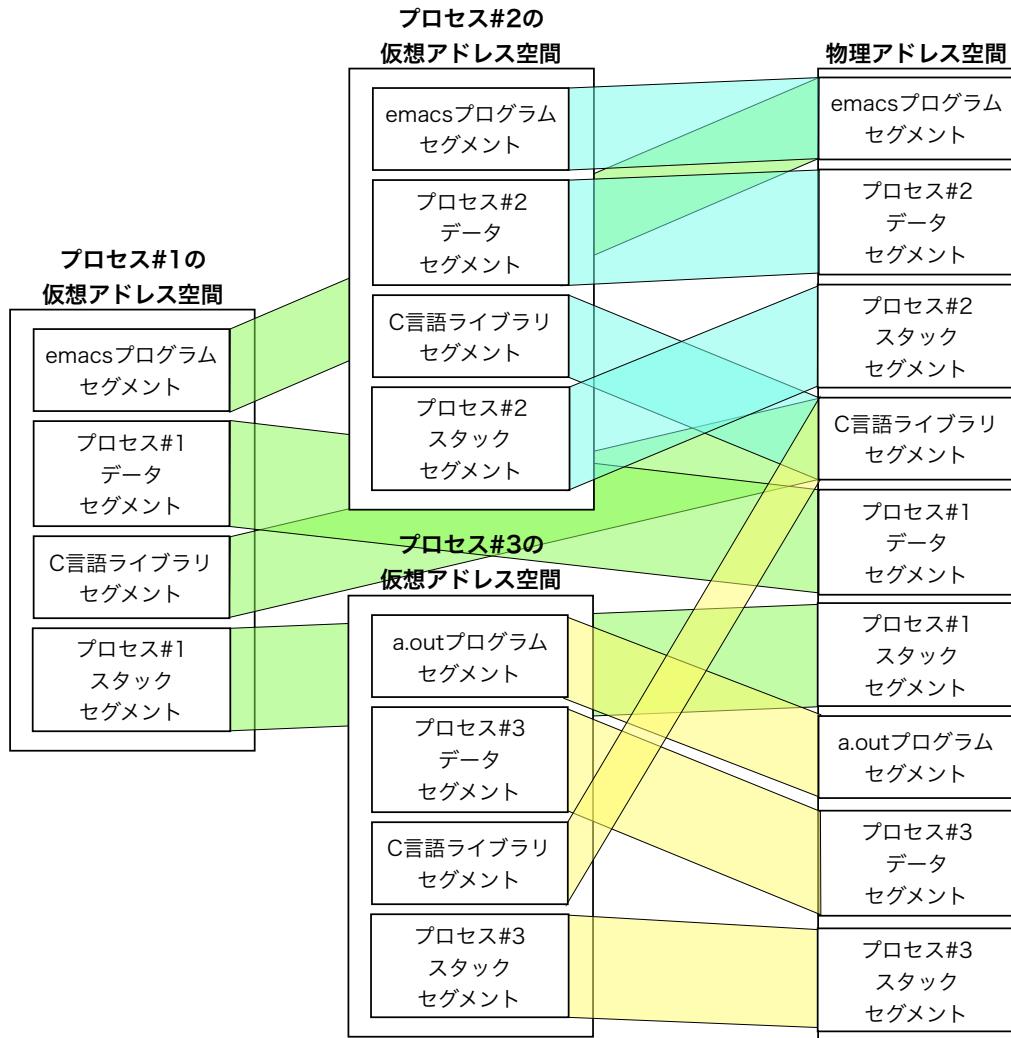


図 11.6 セグメントの共用

- プログラムやデータの共用が容易である。
- セグメントの長さは自由に決められるので内部フラグメントが発生しない。
- セグメントの長さは動的に変化させることも可能である。
- セグメント単位のスワッピングを用いて仮想記憶を実現できる。

欠点

- 物理アドレス空間に外部フラグメントが生じる。
- 外部フラグメントの解消にはメモリコンパクションが必要である。
- 物理メモリ上に連続した領域が必要である。
- 物理メモリより大きいセグメントを作ることができない。

外部フラグメント問題と、物理メモリサイズによるセグメントサイズの制約を解決するために、次の章で紹介するページングと組合せてセグメンテーションを利用するシステムもある。例えば、IA-32 は

そうである。IA-32 ではセグメンテーション機構が出力する一次元のアドレスをページング機構が物理アドレスに変換する。

練習問題

セグメントテーブルが次のような状態の時、以下の間に答えなさい。なお、仮想アドレスは「セグメント番号：物理アドレス」と表記するものとします。また、物理アドレスは8ビットとします。

	C	B	L
0	V=1	0x30	0x20
1	V=1	0x80	0x30
2	V=1	0x00	0x20
3	V=0	0x50	0x20
...

1. 次の仮想アドレスに対応する物理アドレスを答えなさい。但し、物理アドレスに変換できない場合はエラーと答えなさい。
 - (a) 0x0:0x10
 - (b) 0x1:0x10
 - (c) 0x1:0x40
 - (d) 0x2:0x10
 - (e) 0x2:0x20
 - (f) 0x3:0x10
2. セグメントの配置を記入した物理アドレス空間のメモリマップを作成しなさい。

問題

スタックセグメントを意識した前向きに伸びるセグメントも利用可能なセグメンテーション機構を設計しなさい。

1. セグメントテーブルに必要な変更は？
2. 図 11.3 に必要な変更は？
3. 他に必要な変更は？

第 12 章

ページング

メモリを一様なページに分割し、ページ単位で管理することで使いやすい仮想メモリを提供する。メモリより大きな仮想アドレス空間を使用でき、メモリコンパクションが不要なメモリ管理方式である。Windows, macOS, Linux 等の多くのオペレーティングシステムがページングを採用している。

12.1 基本概念

図 12.1 に示すように、プロセスの仮想アドレス空間は固定サイズのページに分割される。物理アドレス空間もページと同じサイズのフレーム^{*1}に分割される。ページはフレームにマッピングされる。

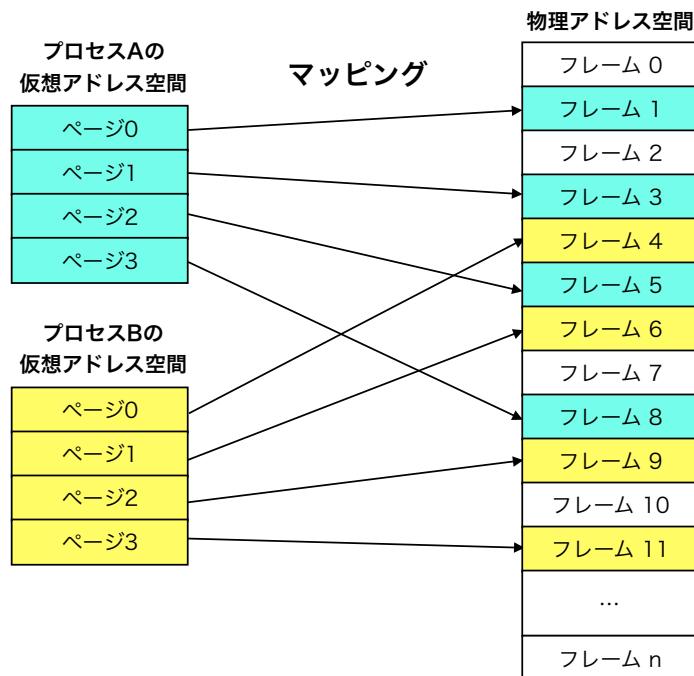


図 12.1 ページからフレームへのマッピング

^{*1} 「フレーム」は、「物理ページ」、「ページフレーム」と呼ばれることがある。

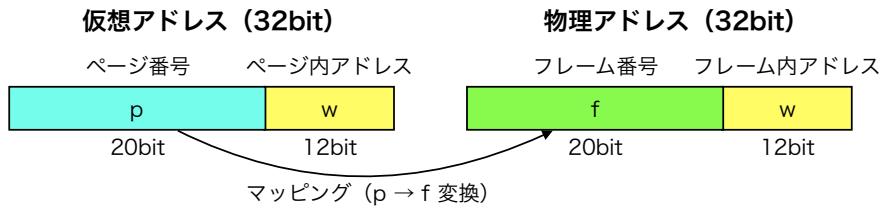


図 12.2 ページング使用時のアドレス例

12.1.1 ページとフレーム

通常、ページのサイズは2の累乗にする。これにより、仮想アドレスの上位ビットをページ番号、下位ビットをページ内アドレスに分割して扱うことができる。物理アドレスでは上位ビットをフレーム番号、下位ビットをフレーム内アドレスに分割して扱う。図12.2に、32ビットの仮想アドレスを4KiB^{*2}のページに分割した例と、32ビットの物理アドレス空間を4KiBのフレームに分割した例を示す。4KiBのページ（フレーム）をバイト毎にアドレス付けするためには、次の計算から分かるように12ビットのページ内（フレーム内）アドレスが必要である。

$$4KiB = 4 \times 1KiB = 2^2 \times 2^{10}B = 2^{12}B$$

上位20ビットがページ（フレーム）番号を下位12ビットがページ内（フレーム内）アドレスを表現する。

仮想アドレスのページ番号（p）を物理アドレスのフレーム番号（f）に変換することで、ページがフレームにマッピングされる。図12.2の例では仮想アドレスも物理アドレスも同じ32ビットであるが、異なるサイズでも構わない^{*3}。図12.1は物理アドレス空間の方が広い例になっていた。

12.1.2 マッピング関数

仮想アドレス由来のページ番号（p）を、物理アドレスの一部であるフレーム番号（f）にマッピングする。マッピング関数はページテーブルと呼ばれる表として実装する。メモリ管理ハードウェア（MMU: Memory Management Unit）が実行時にページテーブルを参照し動的にマッピングを行う。

プロセス毎に異なる仮想アドレス空間をマッピングするので、プロセス毎に異なるマッピング関数（ページテーブル）が必要である。ディスパッチャはプロセスの実行を開始する前にMMUを操作し、新しいプロセスのマッピング関数を有効にする必要がある。図12.1の例では、プロセスA実行時にはページ0がフレーム1へマッピングされる関数を使用するが、プロセスB実行時にはページ0がフレーム4へマッピングされる関数に切り換える必要がある。

12.1.3 外部フラグメンテーション

全てのフレームは、任意のプロセスの任意のページにマッピング可能である。連続したフレームが存在しないと使用できない等の制約は無いのでフレームが無駄になることはない。ページングを用いることで割り付け単位（フレーム）の外にフラグメントが発生しなくなる。外部フラグメンテーション問題は解決しメモリコンパクションも不要になった。

^{*2} IA-32のページサイズは基本的に4KiBである。x86-64でも基本は4KiBである。

^{*3} 同じアーキテクチャですら、時期によって関係が変化することがある。IA-32の物理アドレスは、当初は32ビットであったが途中から36ビットに変更された。この間、論理アドレスは32ビットのまま変更されていない。

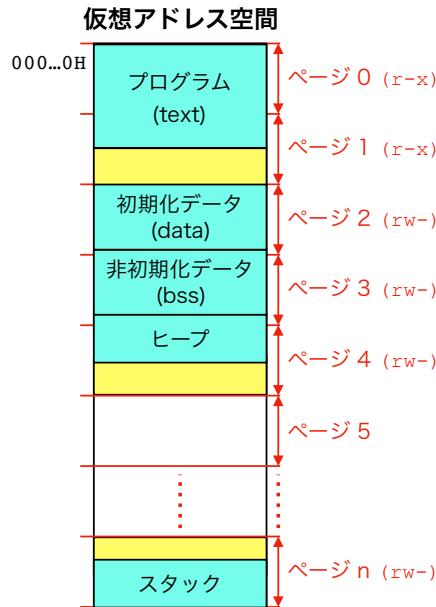


図 12.3 ページング使用時の仮想アドレス空間の例

12.1.4 内部フラグメンテーション

例えば UNIX プロセスの仮想アドレス空間は、図 12.3 のように配置される。プログラム領域はページ 0 からページ 1 の途中までを使用する。これらのページは読み出しと実行だけ (r-x) ができるようメモリ保護を行う。データとヒープは読み出しと書き込みだけ (rw-) ができるようにするので、プログラムとは異なるページに配置する必要がある。そこで、ページ 1 の後半は使用しないで、ページ 2 からページ 4 にデータを配置する。ページ 5 からページ n-1 までは使用しないのでフレームを割り付けていない。仮想アドレス空間に穴が空いた状態にする^{*4}。スタックはページ n に割り付ける。

以上のように配置すると、ページ 1 の後半、ページ 4 の後半、ページ n の前半に、フレームが割り付けられているにも係わらず使用されない領域ができる。このようにページ内部（フレーム内部）に無駄な領域が発生することを内部フラグメンテーションと呼ぶ。フラグメント領域は使用されないはずだが、ユーザプログラムが誤ってアクセスするかもしれない。ページングではメモリ保護をページ単位で行うので、このような不正なアクセスを検知できない問題がある。

12.2 ページング機構

以上で説明したページングを実現するためのハードウェア機構について考える。

12.2.1 ページング機構の概要

図 12.4 にページング機構の模式図を示す。CPU が送出した仮想アドレスは、ページ番号 (p) とページ内アドレス (w) に分けられる。ページ番号は、ページテーブルから一つのエントリを選択するためのインデックスとして使用される。選択されたエントリのフレーム番号 (f) フィールドとページ内アドレス (w) を結合して物理アドレスを得る。

^{*4} sparse address spaces と呼ばれる。

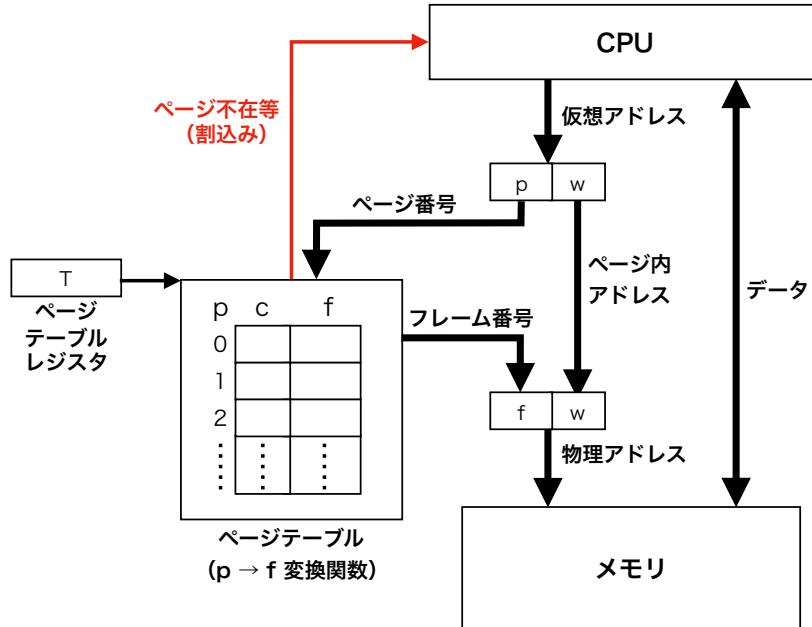


図 12.4 ページング機構の概要

表 12.1 ページテーブルの C フィールドの例

名称	ビット数	意味
V (Valid)	1	フレームが割り付けられている。
R (Reference)	1	ページの内容が参照された。
D (Dirty)	1	ページの内容が変更された。
RWX (Read/Write/eXecute)	3	許されるアクセス方法。

12.2.2 ページテーブルエントリ

ページテーブルのエントリはページ番号で選択する。エントリの内容は c (制御) と f (フレーム番号) フィールドである。f フィールドの内容がページテーブルの出力になる。

c フィールドの内容は、例えば表 12.1 のようなものである。ページにフレームが割り付けられていない場合は V ビットが 0 になっている。V ビットが 0 のページにアクセスした場合は CPU にページ不在割込みを発生する。R ビットはページが参照された時に 1 に変化する。R ビットはページの使用頻度を調べるために使用される^{*5}。D ビットはページが書き換えられた時に 1 に変化する。D ビットはページをスワップアウトする際に使用される。

12.2.3 ページテーブル

ページテーブルはかなり大きな表である。例えば図 12.2 のようにページ番号が 20 ビットで表現されるなら、ページテーブルは $2^{20} = 2Mi$ エントリの大きさを持つことになる。また、メモリアクセスの度に参照されるので、通常のメモリと比較して桁違いに高速でなければならない。

図 12.4 ではページテーブルが専用のハードウェアのように描かれているが、このような大きくて高速

^{*5} 詳しくは仮想記憶の章で説明する。

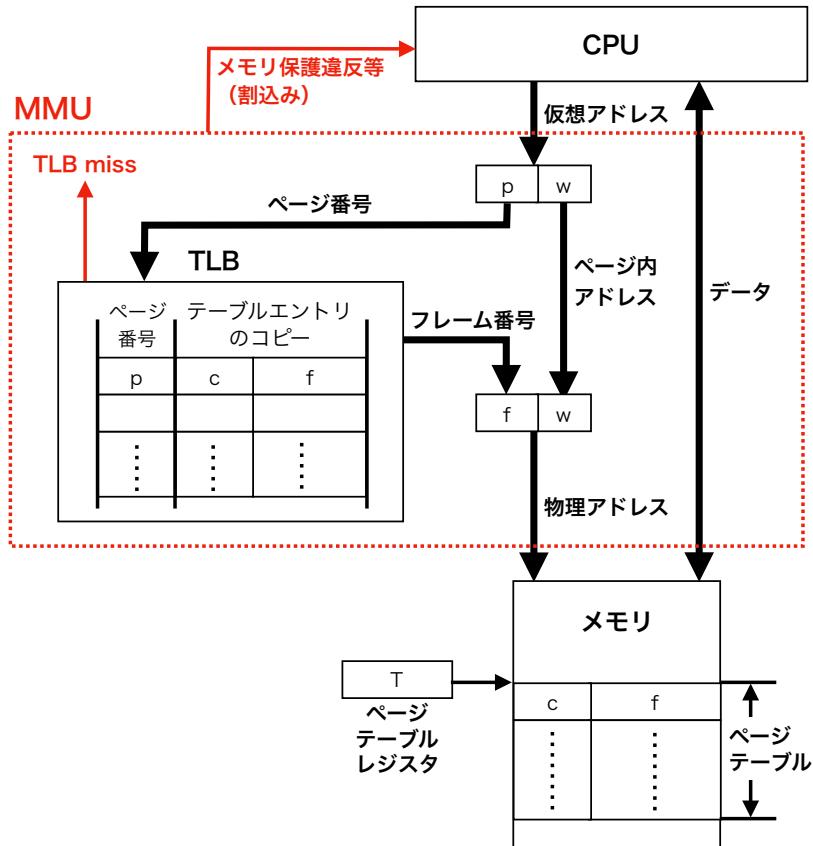


図 12.5 TLB を使用するページング機構

な表を MMU の内部に持つことは困難である。また、プロセス毎にページテーブルが必要なので、プロセススイッチの度にページテーブル全体を MMU にロードし直すのも効率が悪い。そこで、ページテーブルはメモリ上に置くことになる。ページテーブルのメモリ上のアドレスはページテーブルレジスタが記憶している。

12.2.4 TLB (Translation Look-aside Buffer)

CPU がメモリをアクセスする度にメモリ上のページテーブルをアクセスすると、メモリのアクセス回数が二倍になる^{*6}。そこで、変換結果を MMU 内の TLB と呼ばれる高速なメモリにキャッシュするようとする。TLB はページ番号とフレーム番号の対応表を記憶し、ページ番号をキーにして非常に高速に検索できる特殊なメモリである。このような記憶したキーで高速に検索できるメモリを連想メモリと呼ぶ。TLB のサイズは機種により異なり、数十エントリから数千エントリ程度である。

図 12.5 に TLB を使用したページング機構の模式図を示す。CPU が output したページ番号を用いて TLB を検索し、見つかれば TLB からフレーム番号が output される。その際、TLB 上にコピーされた D (Dirty) ビットが操作されたり、RWX フィールドがチェックされたりする。チェックの結果、違反が見つかれば CPU に割込みを発生する。

^{*6} セグメントテーブルの章で説明したことと同じである。

12.2.5 Page Table Walk

ページ番号が TLB に見つからない場合は、*TLB miss* になりページテーブル上を検索する必要がある。ページテーブル上の検索のことを *page table walk* と呼ぶ。page table walk を行い TLB を更新する作業を MMU のハードウェアが自動的に行う機種と、CPU に割込みを発生しソフトウェアで行う機種がある。前者はハードウェアで行うことで高速化を狙う。後者は MMU を単純にしたことで余ったチップ面積を TLB のエントリ数を増やすために使用し TLB miss の頻度を低くすることを狙う。

TLB に空きエントリが無い場合、新しいページテーブルエントリをロードする前に、どれかのエントリを TLB から捨てる必要がある。TLB 上のページテーブルエントリのコピーは、ロードされた後に D (Dirty) ビットが変更されている可能性がある。TLB のエントリを捨てる前にメモリ上のページテーブルに書き戻すことがある。

12.2.6 TLB エントリのクリア

プロセスは専用の仮想アドレス空間を持つので、プロセス毎に専用のページテーブルを持つことになる。プロセススイッチ時に、ディスパッチャが新しいプロセスのページテーブルのアドレスをページテーブルレジスタにロードする。

TLB は古いページテーブルの内容を反映しているので、ページテーブルを交換する際にクリアする。TLB の全てのエントリがクリアされると、直後に同じプロセスに戻ってきた場合や、カーネル領域などがプロセス間で共有される場合に効率が悪いので様々な工夫^{*7}が凝らされている場合もあるが、基本的にはプロセススイッチを行う際は TLB をクリアする。また、ページテーブルが変更された場合は、プロセススイッチが発生しなくとも TLB をクリアする必要がある。

12.3 ページの共用

セグメンテーションではプロセスがセグメントを共用することができた。ページングではプロセス間でページを共用することができる。図 12.6 は、プロセス A とプロセス B が同じプログラム X を、プロセス C がプログラム C 実行している例である。各プロセスのページテーブルを適切に設定することで、図のようなマッピングをすることができる。

プログラム本体やライブラリの機械語は読み出しと実行専用 (R-X) になっており、プロセスが変更することは無いので共用することができる。プログラム X の機械語は第 1 フレームに格納されプロセス A とプロセス B で共用する。プログラム C の機械語は第 4 フレームと第 7 フレームを合わせた領域に格納される。プログラム C を実行するプロセスは他に無いので共用する必要がない。

ライブラリの機械語は第 3 フレームに格納されプロセス A, プロセス B, プロセス C が共用して利用する。ライブラリはプロセス A・B とプロセス C で異なる仮想アドレスにマッピングされている。ライブラリ内の機械語プログラムは何番地にロードされても実行できる位置独立コード^{*8}でなければならぬ。セグメンテーションには、このような制約は無かった。

プロセス毎に内容が異なるデータやスタッカクは共用できない。プロセス毎に専用の領域を割当てる。

^{*7} TLB のエントリにプロセス番号も記録しクリア不要にする方式や、仮想アドレスを指定して特定のエントリだけクリアする方式が知られている。

^{*8} JMP や CALL 命令のアドレッシングは全てプログラムカウンタ相対で行う。データのアドレッシングは CPU レジスターに格納したアドレスを基準にした相対で行う。

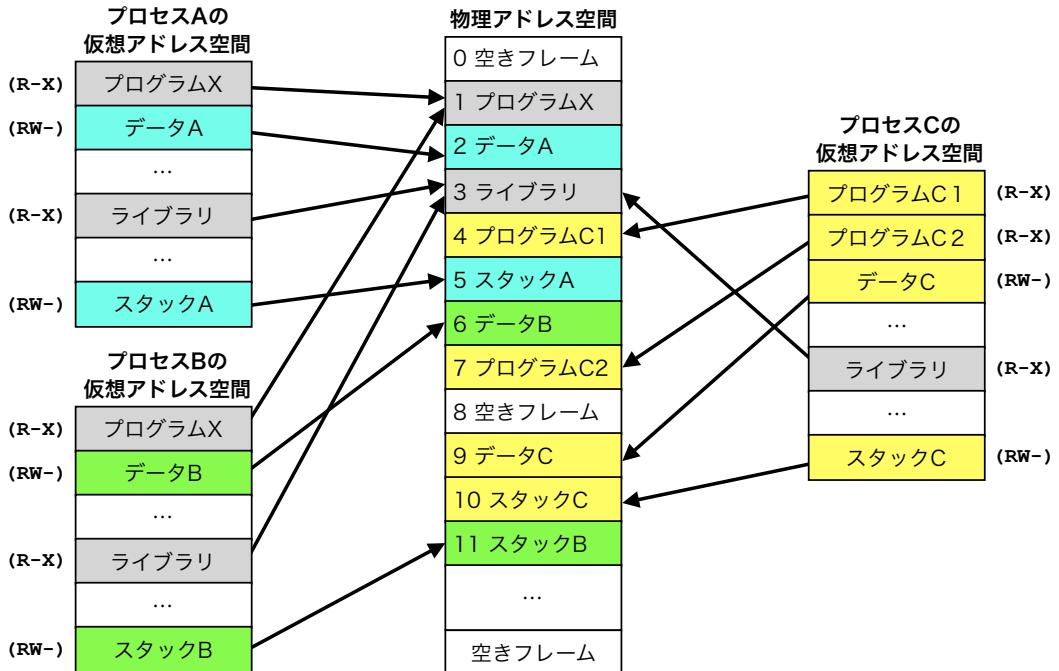


図 12.6 プロセス間でのページ共用

12.4 ページテーブルの編成方法

図 12.5 に示したようにページテーブルはメモリに置かれる。しかし、ページテーブルのサイズは無視できるほど小さなものではない。例えば、32 ビットマイクロプロセッサが PC に普及してきた 1990 年代の前半には、PC が備えるメモリは 4MiB から 16MiB 程度であった。IA-32 を用いる場合、ページテーブルの一つのエントリは 4 バイトなので、32 ビットの仮想アドレス空間を 4KiB のページで分割すると、次の計算のようにページテーブル全体では 4MiB になる。更に、ページテーブルはプロセス毎に必要である。メモリのほとんどをページテーブルに使用しても足らない。このままではページングは実用にならない。

$$2^{32}B \div 2^{12}B = 2^{20} = 1Mi\text{エントリ}$$

$$1Mi\text{エントリ} \times 4B = 4MiB$$

近年の 64 ビットマイクロプロセッサの場合も同様である。x86-64 では 48 ビットの仮想アドレス空間を 4KiB のページで分割する。ページテーブルの一つのエントリが 8 バイトなので、下の計算のようにページテーブルのサイズが 512GiB になる。これでは現代の PC にもページテーブルが大きすぎる。ページテーブルを小さくする必要がある。

$$2^{48}B \div 2^{12}B = 2^{36} = 64Gi\text{エントリ}$$

$$64Gi\text{エントリ} \times 8B = 512GiB$$

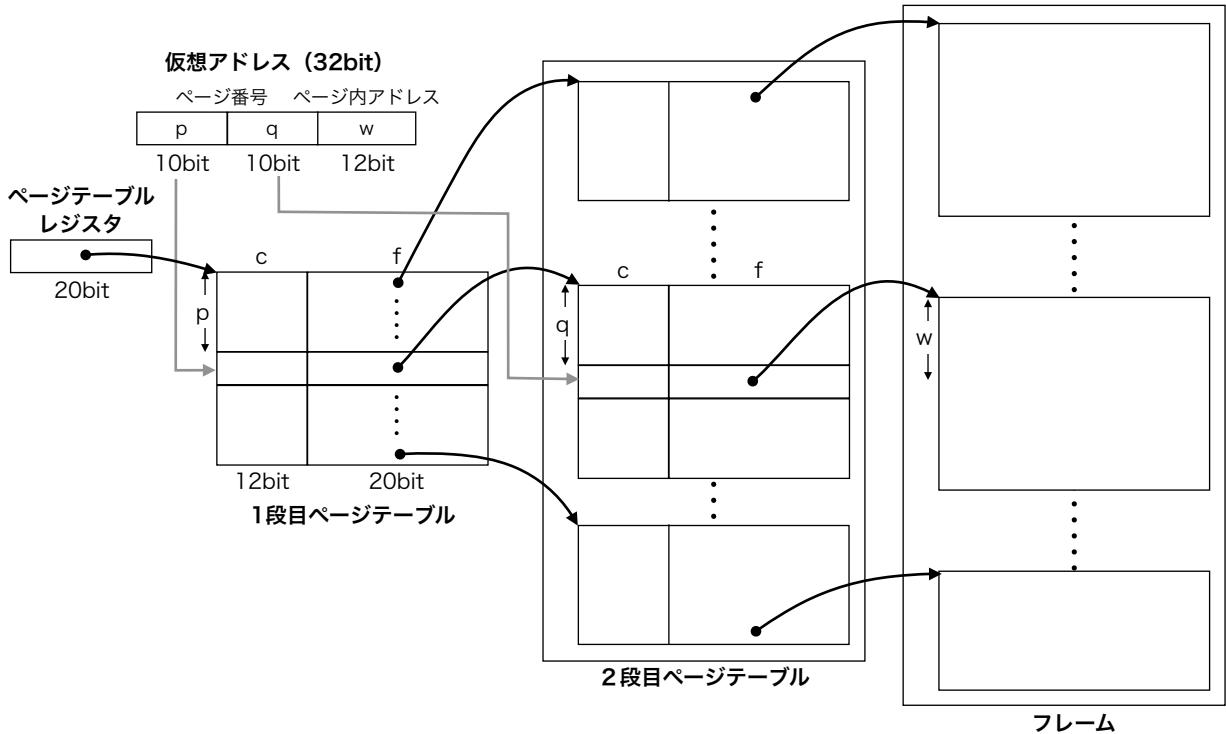


図 12.7 二段ページテーブルの構造

12.4.1 二段のページテーブル

ページテーブルを二段にすることで、二段目に使用するメモリを節約することができる。図 12.7 に IA-32 で使用される二段のページテーブルの例を示す^{*9}。図の左上に示すように、32 ビットの仮想アドレスは 10 ビットのページ番号フィールド二つ (p, q) と、12 ビットのページ内アドレス (w) に分けられる。ページサイズは w が 12 ビットなので $2^{12} = 4KiB$ である。物理アドレスも 32 ビット^{*10}なので、フレーム番号は 20 ビットで表現する。

- *Page Table Walk*

まず、ページテーブルレジスタから一段目のページテーブルの位置を知る。次に、一段目のページテーブルを p をインデクスにして参照することで、二段目のページテーブルの位置を知ることができる。最後に、二段目のページテーブルを q をインデクスにして参照することで、フレームの位置を知る。フレームの w バイト目が目的のアドレスである。このように二つのページテーブルを用いた page table walk を行うことで目的の物理アドレスに辿り着く。

- 一段目のページテーブル

一段目のページテーブルの大きさは、p が 10 ビット、エントリサイズが 4 バイトより $2^{10} \times 4B = 4KiB$ となりフレームと同じである。そこで、どれか一つのフレームに一段目のページテーブルを格納することにする。ページテーブルレジスタはフレームの番号（10 ビット）を記録すれば良い。

^{*9} 図 12.7 では IA-32 の用語ではなく、より一般的な用語を使用している。

^{*10} 初期の IA-32 の場合である。途中から 36 ビットに拡張された。

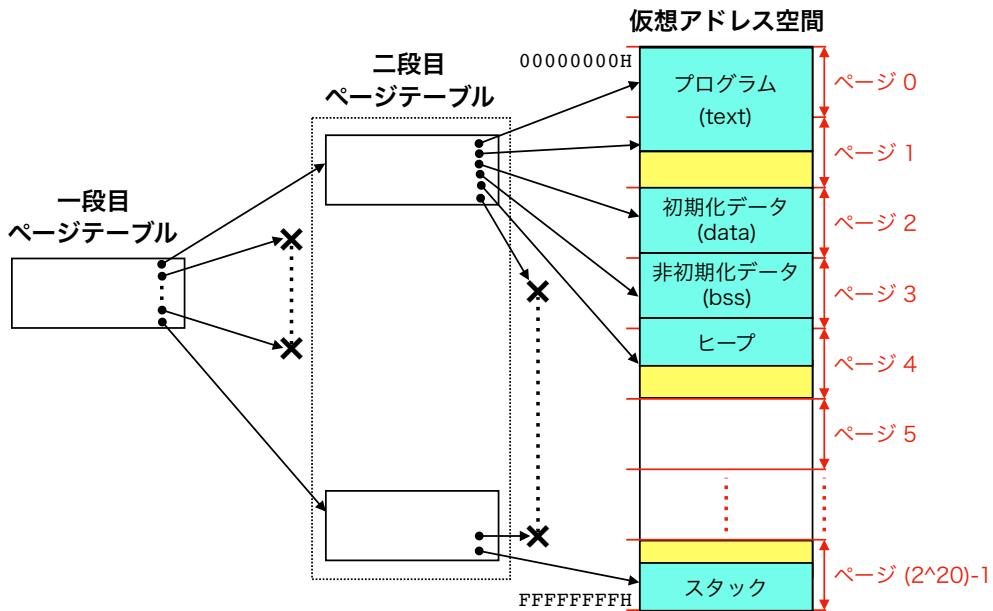


図 12.8 穴空き仮想アドレス空間のページテーブル

- 二段目のページテーブル

一段目のページテーブルの一つのエントリが、二段目のページテーブルの一つの区画を選択する。区画は 10 ビットの q を用いて参照されるので、大きさは一段目のページテーブルと同じ $4KiB$ になる。区画も一つのフレームに格納される。一段目のページテーブルの f フィールドには、二段目ページテーブルの一つの区画のフレーム番号（10 ビット）を格納する。

- メモリの節約

図 12.3 や図 12.6 で示したように、プロセスの仮想アドレス空間にはフレームが割り付けられていない大きな穴が空いている。図 12.8 のように、穴の部分に二段目のページテーブルを割当てないことでメモリを節約できる。図の例では一段目と二段目合わせて 3 フレームしかページテーブルのために使用していない。もしも、二段目のページを全て割り付けたなら $2^{10} + 1 = 1,025$ フレームを使用するので効果は大きい。また、二段目のページテーブルとフレームで使用頻度の低いものを二次記憶装置にスワップアウトすることも可能である^{*11}。

12.4.2 多段ページテーブル

前の節では二段のページテーブルを紹介した。仮想アドレス空間が更に広い場合は、より段数の多いページテーブルが使用されることがある。例えば、x86-64 では 64 ビット（実質的には 48 ビット）^{*12} の広い仮想アドレス空間が使用できる。IA-32 では仮想アドレス空間が 32 ビットだったので 4GiB より大きなプロセス（セグメント）を作ることはできなかった。x86-64 ではより大きなプロセスを作ることができるので、4GiB の上限を気にしないでプログラミングできる。

*11 詳しくは仮想記憶の章で述べる。

*12 図 12.9 に示すように 64 ビットの上位 16 ビットは未使用なので、実質的な仮想アドレスは 48 ビットになる。 $2^{48}B = 2^8 \times 2^{40}B = 256TiB$ なので、48 ビットでも十分に広いアドレス空間である。

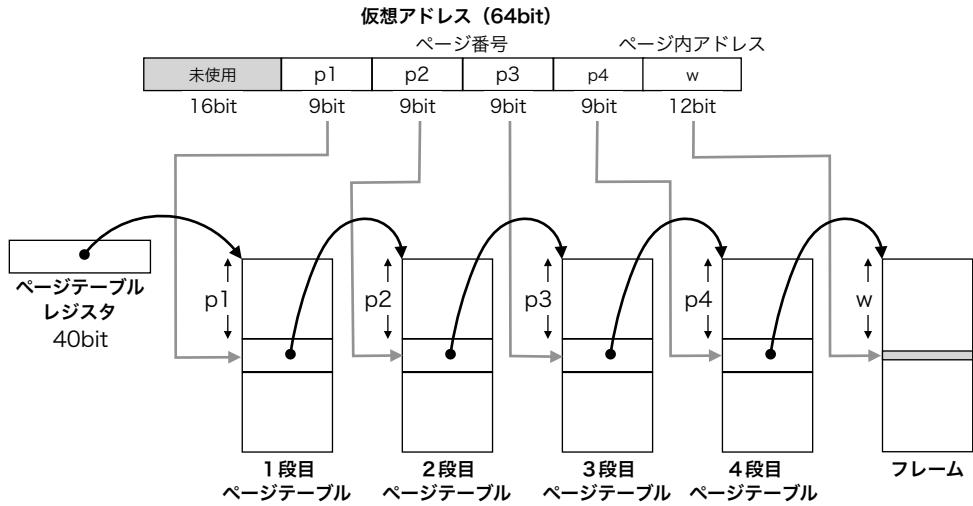


図 12.9 四段のページテーブルの例

しかし、二段のページテーブルを使用し続けると、ページテーブルの区画が大きくなりメモリの無駄が多くなる。例えば、仮想アドレスが 48 ビット、二段のページテーブルを用い、エントリのサイズが 8 バイトと仮定する。48 ビットの仮想アドレスを、18 ビット (p), 18 ビット (q), 12 ビット (w) に分割して扱う場合、ページテーブル一区画のサイズは $2^{18} \times 8B = 2MiB$ となる。図 12.8 と同様な考え方で、最低限である 3 区画がページテーブルに割当てられたとするとプロセス当たり 6MiB になる。システム内にプロセスが 400 個^{*13}あったとすると、最低でも 2.4GiB のメモリがページテーブルのために消費されることになる。ページテーブルに使用されるメモリが多すぎる^{*14}。

ページテーブルの区画を小さくするために仮想アドレスをより小さく区切る。例えば、x86-64 では図 12.9 のよう仮想アドレスのページ番号部分を四つに区切っている。ページ内アドレスが 12 ビットなので、ここでもページ（フレーム）サイズは 4KiB である。ページテーブルは 9 ビットの p1, p2, p3, p4 でインデクスされるので 512 エントリである。エントリサイズは x86-64 の場合 8 バイトなので、ページテーブルは 4KiB になりフレームサイズと同じになる。図 12.8 と同様な仮想アドレス空間をマッピングする場合、ページテーブルに使用するメモリは、1 段目に 1 フレーム、2 段目に 2 フレーム、3 段目に 2 フレーム、4 段目に 2 フレームの合計 7 フレーム (28KiB) で済む。プロセスが 400 個あったとしても、ページテーブルに使用するメモリの合計は約 11MiB にしかならない^{*15}。

12.4.3 逆引きページテーブル

従来のページテーブルは、仮想アドレス空間の大きさにより大きくなるし、仮想アドレス空間の数だけ必要である。これは、仮想アドレスから物理アドレスに変換するために、仮想アドレス（ページ番号）をインデクスとし物理アドレス（フレーム番号）を内容とする表を、仮想アドレス空間毎に用いる自然な発想から生まれた。逆引きページテーブルは、従来のページテーブルとは逆に、物理アドレス（フ

^{*13} この原稿を書いている MacBook では、現在 354 個のプロセスが走っている。

^{*14} この原稿を書いている MacBook のメモリは 8GiB である。仮定どおりならメモリの 30% が、たった一つのプロセスのページテーブルに消費されることになる。

^{*15} プロセスあたり、8GiB の 0.14% しか使用しない。

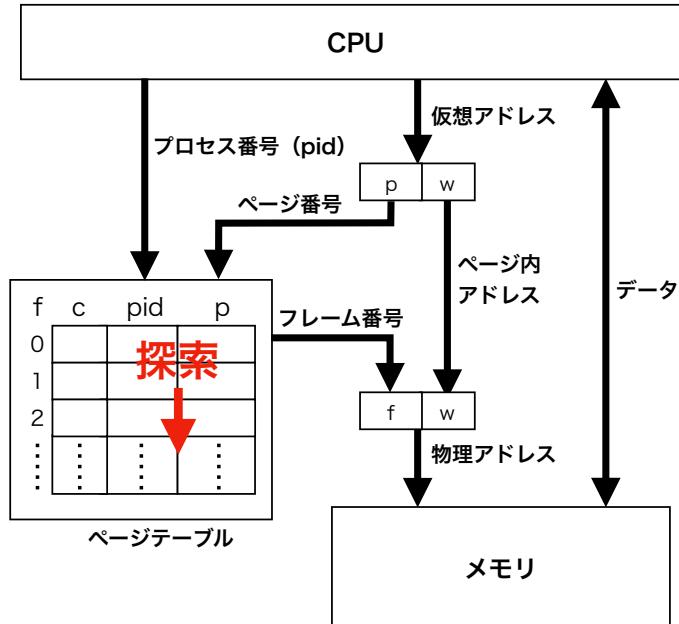


図 12.10 逆引きページテーブルの概要

フレーム番号) をインデクスとし、仮想アドレス (ページ番号) を内容とする表をシステム全体で一つだけ用いる方式である。

- ページテーブルのサイズ

ページテーブルのエントリ数はフレーム数と等しくなるので、ページテーブルが使用する領域の大きさは、メモリ全体と比較して小さく、かつ、一定である。例えば 8GiB のメモリを管理するために、ページサイズが 4KiB なら次の計算のように 2Mi エントリである。エントリサイズが 8 バイトと仮定するとページテーブル全体で 16MiB となる^{*16}。

$$8GiB \div 4KiB = 2^{33} \div 2^{12} = 2^{21} = 2Mi \text{ エントリ}$$

- *Page Table Walk*

図 12.10 に逆引きページテーブルの模式図を示す。システムに一つだけの表なので、どのプロセスの仮想アドレス空間にフレームが割り付けられているか識別するために、プロセス番号 (pid) がページテーブルに格納されている。ページテーブルをプロセス番号 (pid) とページ番号 (p) で検索し、見つかったエントリのインデクス (f) をフレーム番号として出力する。

- ハッシュ表を用いた *page table walk*

ページテーブルを先頭から順に探索していくは遅くて実用にならない。ハッシュ表を用いた検索を行う。図 12.11 に IBM 801 ミニコンピュータで用いられたメモリ管理方式 [44] を参考にした機構の模式図を示す。

チェインのためのフィールド (next) を設け、ページテーブルをチェインハッシュ表として扱う。

^{*16} システム全体で 8GiB の 0.2% しか使用しない。

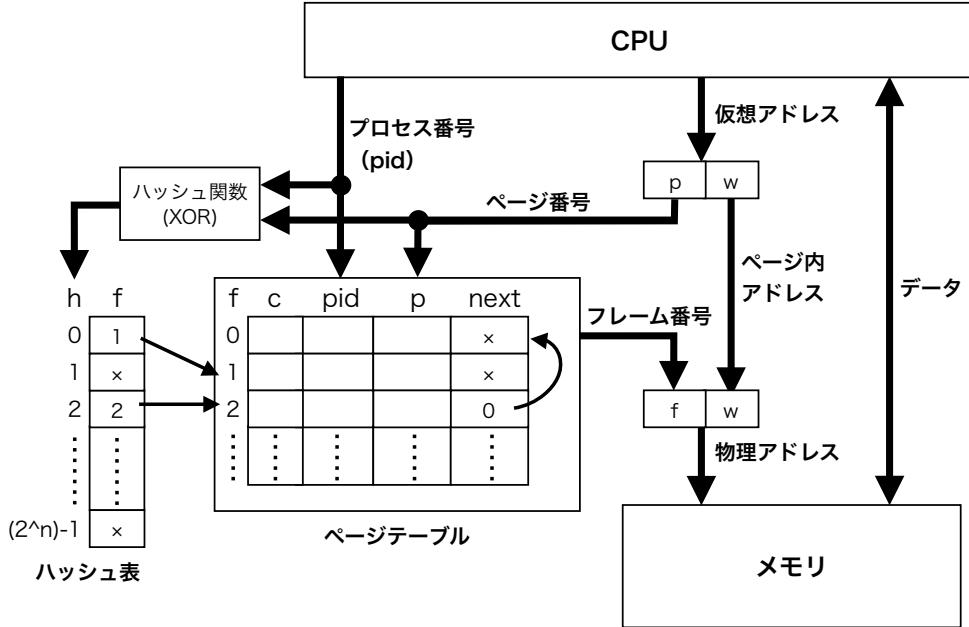


図 12.11 ハッシュを用いた逆引きページテーブルの構造

ハッシュ表の大きさはハッシュ関数の作りやすさから二の累乗とする^{*17}。ハッシュ値はプロセス番号^{*18}とページ番号の XOR で計算する。ハッシュ表はページテーブルの一つのエントリのインデクスを格納する。エントリのプロセス番号 (pid) とページ番号 (p) が目的のものであれば、エントリの番号 (f) がフレーム番号として使用される。目的のものでない場合はチェイン (next) を使用して次のエントリに進む。チェーンの最後まで調べて見つからなければページ不在である。

- **TLB**

実際はハッシュ表やページテーブルはメモリに配置する。メモリアクセスの度に page table walk を行っていては、メモリアクセスに時間がかかりすぎる。普段は TLB を用いる。

練習問題

1. 一回のメモリアクセス時間に 1ns, page table walk に 2ns かかるとする, TLB のヒット率が 90 パーセントの時の平均メモリアクセス時間を計算しなさい。
2. 図 12.7において, $p = 1$ の仮想アドレスの範囲を 8 桁の 16 進数で答えなさい。
3. 図 12.7において, $p = 1, q = 1$ の仮想アドレスの範囲を 8 桁の 16 進数で答えなさい。
4. 逆引きページテーブルを用いる場合, TLB に格納すべき最低限の情報の範囲を考察しなさい。
5. 図 12.11 に, $pid = 3, p = 2$ のページがフレーム 1 にマッピングされるようなページテーブルの状態を書き込みなさい。
6. 逆引きページテーブルを用いるシステムで, プロセス間でページの共有が可能か考察しなさい。

*17 ハッシュ表のサイズが二の累乗なら, ハッシュ値は計算結果の一部のビットを使用すれば良い。

*18 IBM 801 ではセグメント ID であった。

参考文献

- [1] ウキペディア, OS/360, <https://ja.wikipedia.org/wiki/OS/360> (2017.10.03 閲覧)
- [2] ウキペディア, MVS, https://ja.wikipedia.org/wiki/Multiple_Virtual_Storage (2017.10.03 閲覧)
- [3] ウキペディア, OS/390, <https://ja.wikipedia.org/wiki/OS/390> (2017.10.03 閲覧)
- [4] ウキペディア, z/OS, <https://ja.wikipedia.org/wiki/Z/OS> (2017.10.03 閲覧)
- [5] ウキペディア, UNIX (「UNIX および UNIX 系システムの系統図」を含む), <https://ja.wikipedia.org/wiki/UNIX> (2017.10.03 閲覧)
- [6] ウキペディア, Solaris, <https://ja.wikipedia.org/wiki/Solaris> (2017.10.03 閲覧)
- [7] ウキペディア, AIX, <https://ja.wikipedia.org/wiki/AIX> (2017.10.03 閲覧)
- [8] ウキペディア, Mach, <https://ja.wikipedia.org/wiki/Mach> (2017.10.03 閲覧)
- [9] ウキペディア, BSD の子孫, <https://ja.wikipedia.org/wiki/BSD%E3%81%AE%E5%AD%90%E5%AD%AB> (2017.10.03 閲覧)
- [10] ウキペディア, BSD, <https://ja.wikipedia.org/wiki/BSD> (2017.10.04 閲覧)
- [11] ウキペディア, 386BSD, <https://ja.wikipedia.org/wiki/386BSD> (2017.10.04 閲覧)
- [12] ウキペディア, FreeBSD, <https://ja.wikipedia.org/wiki/FreeBSD> (2017.10.03 閲覧)
- [13] ウキペディア, FreeNAS, <https://ja.wikipedia.org/wiki/FreeNAS> (2017.10.03 閲覧)
- [14] ウキペディア, NEXTSTEP, <https://ja.wikipedia.org/wiki/NEXTSTEP> (2017.10.03 閲覧)
- [15] ウキペディア, Classic Mac OS, https://ja.wikipedia.org/wiki/Classic_Mac_OS (2017.10.03 閲覧)
- [16] ウキペディア, ダイナブック, <https://ja.wikipedia.org/wiki/ダイナブック> (2017.10.03 閲覧)
- [17] ウキペディア, macOS, <https://ja.wikipedia.org/wiki/MacOS> (2017.10.03 閲覧)
- [18] ウキペディア, iOS (アップル), [https://ja.wikipedia.org/wiki/IOS_\(アップル\)](https://ja.wikipedia.org/wiki/IOS_(アップル)) (2017.10.03 閲覧)
- [19] ウキペディア, Linux, <https://ja.wikipedia.org/wiki/Linux> (2017.10.03 閲覧)
- [20] ウキペディア, Andriod, <https://ja.wikipedia.org/wiki/Android> (2017.10.03 閲覧)
- [21] ウキペディア, MS-DOS, <https://ja.wikipedia.org/wiki/MS-DOS> (2017.10.03 閲覧)
- [22] ウキペディア, Microsoft Windows (「Windows ファミリー系統図」含む), https://ja.wikipedia.org/wiki/Microsoft_Windows (2017.10.03 閲覧)

- [23] ウキペディア, IBM PC, https://ja.wikipedia.org/wiki/IBM_PC (2017.10.04 閲覧)
- [24] ウキペディア, UNIX System V, https://ja.wikipedia.org/wiki/UNIX_System_V (2017.10.04 閲覧)
- [25] 重村哲至, 情報電子工学科電算機室における PC-UNIX の歴史, <http://www2.tokuyama.ac.jp/giga/Sigemura/Public/IeNet/history.html> (2017.10.03 閲覧)
- [26] Linux kernel release 1.0, <https://www.kernel.org/pub/linux/kernel/v1.0/linux-1.0.tar.gz> (2017.10.04)
- [27] Andrew S. Tanenbaum, Herbert Bos : "The Third Generation(1965–1980):ICs and Multiprogramming", Modern Operating Systems (4th Edition), pp.9-14, Pearson Education, Inc (2014).
- [28] Andrew S. Tanenbaum, Herbert Bos : "The Fourth Generation(1980–Present):Personal Computers", Modern Operating Systems (4th Edition), pp.15–19, Pearson Education, Inc (2014).
- [29] Alan C. Kay : "A Personal Computer for Children of All Ages", Proceeding ACM '72 Proceedings of the ACM annual conference - Volume 1 Article No 1 (1972).
- [30] アラン・ケイ : すべての年齢の「子供たち」のためのパソコンコンピュータ, 阿部和広, 小学生からはじめるわくわくプログラミング, pp.130–141, 日経 BP 社 (2013).
- [31] アラン・ケイ : Dynabook とは何か? 「すべての年齢の「子供たち」のためのパソコンコンピュータ」の後日談, 阿部和広, 小学生からはじめるわくわくプログラミング, pp.142–149, 日経 BP 社 (2013).
- [32] 師尾 潤他: スーパーコンピュータ「京」のオペレーティングシステム, <http://img.jp.fujitsu.com/downloads/jp/jmag/vol63-3/paper07.pdf> (2017.10.03 閲覧), 富士通 (2012).
- [33] Marshall Kirk McKusick, George V. Neville-Neil, Robert N. M. Watson : The Zettabyte Filesystem, The Design and Implementation of the FreeBSD Operating System Second Edition, Pearson Education, Inc (2015).
- [34] Andrew S. Tanenbaum, Herbert Bos : "INTRODUCTION", Modern Operating Systems (4th Edition), pp.1-3, Pearson Education, Inc (2014).
- [35] Andrew S. Tanenbaum, Herbert Bos : "VIRTUALIZATION AND THE CLOUD", Modern Operating Systems (4th Edition), pp.471-516, Pearson Education, Inc (2014).
- [36] ヴイエムウェア株式会社 : "VMware 徹底入門 第3版", 廣済堂 (2012).
- [37] 仮想ハードディスクイメージのダウンロード, <https://www.ubuntulinux.jp/download/ja-remix-vhd> (2017.10.19 閲覧), Ubuntu Japanese Team (2012).
- [38] Andrew S. Tanenbaum, Herbert Bos : "Thread Usage", Modern Operating Systems (4th Edition), pp.97-102, Pearson Education, Inc (2014).
- [39] ウキペディア, ハイパースレッディング・テクノロジー, <https://ja.wikipedia.org/wiki/%E3%83%8F%E3%82%A4%E3%83%91%E3%83%BC%E3%82%B9%E3%83%AC%E3%83%83%E3%83%87%E3%82%A3%E3%83%B3%E3%82%B0%E3%83%BB%E3%83%86%E3%82%AF%E3%83%8E%E3%83%AD%E3%82%B8%E3%83%BC> (2017.11.02 閲覧)
- [40] B.H.Liskov, S.N.Zilles : "Programming with Abstract Data Type", SIGPLAN Notices, 9, 4, pp.50-59 (1974).

-
- [41] John H. Crawford, Patrick P. Gelsinger：“ベースとリミット”, 80386 プログラミング, 工学社, pp.413-414 (1988) .
 - [42] John H. Crawford, Patrick P. Gelsinger：“セグメント部：セグメント・レジスタ”, 80386 プログラミング, 工学社, pp.48-50 (1988) .
 - [43] John H. Crawford, Patrick P. Gelsinger：“デスクリプタ用の裏レジスタ”, 80386 プログラミング, 工学社, pp.420-421 (1988) .
 - [44] Albert Chang, Mark F. Mergen：“801 storage: architecture and programming”, ACM Transactions on Computer Systems, 6, 1, pp.28-50 (1988) .

付録 A

TaC に関する資料

A.1 CPU の概要

TaC で使用できるデータの形式、CPU 内部のレジスタ構成、機械語命令について説明する。

A.1.1 データ形式

図 A.1 に TaC が扱うことができるデータ形式を示す。16 ビットの整数データと、16 ビットのアドレスデータの他に、8 ビットのデータを扱うことができる。16 ビットのデータは CPU の内部でもメモリや I/O でも使用できる。メモリや I/O の 16 ビットデータにアクセスする場合は偶数番地を用いる。8 ビットデータはメモリと I/O の読み書きだけに使用できる。メモリや I/O の 8 ビットデータにアクセスする場合は、CPU レジスタの下位 8 ビットが使用される。

A.1.2 CPU レジスタと PSW

図 A.2 に CPU 内部のレジスタなどを示す。レジスタはどれも 16 ビット幅である。

CPU レジスタは、汎用の G0 から G11、フレームポインタとして使用する FP、カーネルモード用のスタックポインタ SSP、ユーザモード用のスタックポインタ USP からなる。これらは全て計算用にもアドレス用にも使用できる。FP、SSP、USP は、以下に説明する特別な意味も持っている。

FP はフレームポインタ相対アドレッシングモードで使用できる。このアドレッシングモードを用いると、スタックフレーム内のローカル変数や関数引数へ、1 ワードの機械語命令でアクセスできる。

SSP はカーネルモードで SP の位置にマップされスタックポインタとして使用される。USP はユーザモードで SP の位置にマップされスタックポインタとして使用される。USP は最後のレジスタとして常にマップされており、カーネルモードでも USP をアクセスすることができる。

PSW は PC と FLAG からなる。PC はプログラムカウンタのことである。FLAG には、計算結果で変化する V, C, S, Z と、割込み許可 E、カーネルモード P の各ビットがある。割込みが発生する

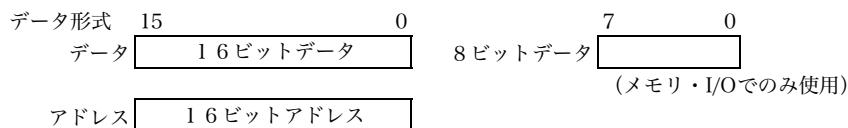


図 A.1 データ形式

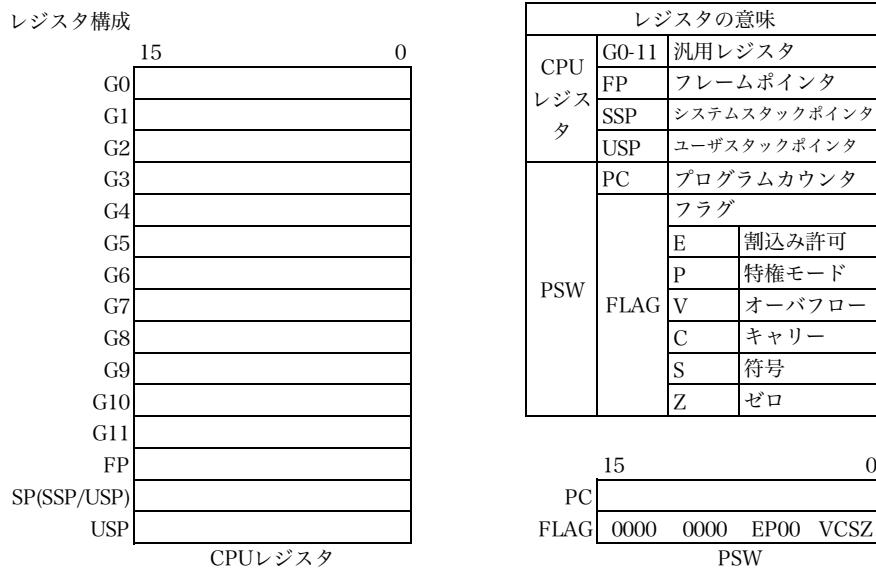


図 A.2 CPU 内部の記憶装置

と割込みが禁止された上でカーネルモードに切り換わり (E ビットが 0, P ビットが 1 になる), PC, FLAG の順にカーネルスタックに PUSH される.

A.1.3 機械語命令

図 A.3 に TaC の機械語命令の一覧表を示す. IN, OUT, RETI, EI, DI, HALT はカーネルモードでしか使用できない特権命令である. SVC 命令はシステムコールを発行するために SVC 割込みを発生する.

ほとんどの転送命令と計算命令で 8 種類のアドレッシング・モードが使用できる. Direct, Indexed, Immediate の三つのアドレッシング・モードを使用する場合は 2 ワードの機械語命令になる. 他のアドレッシング・モードの場合は全て 1 ワード命令である.

Byte Register Indirect アドレッシング・モードだけが, メモリの 8 ビットデータをアクセスする. ST 命令では CPU レジスタの下位 8 ビットがメモリに書き込まれる. その他の命令ではメモリから読み出した 8 ビットデータの上位に 00h を付加した 16 ビットデータに変換して使用する.

命令	ニーモニック	オペコード	アドレッシングモード（数値はステート数）									フラグ変化	説明
			Drc	Index	Imm	FP Rlt	Reg	Imm4	Indr	B Indr	Othr		
No Operation	NO	00h 0h 0h	--	--	--	--	--	--	--	--	3	x	何もしない
Load	LD Rd,EA	08h Rd EA	7	7	5	7	4	4	6	6	--	x	Rd ← [EA]
Load	LD Rd,FLAG	14h Rd 0h	--	--	--	--	--	--	--	--	4	x	Rd ← FLAG
Store	ST Rd,EA	10h Rd EA	6	6	--	6	--	--	5	5	--	x	[Dsp] ← EA
Add	ADD Rd,EA	18h Rd EA	7	7	5	7	5	4	6	6	--	○	Rd ← Rd + [EA]
Subtract	SUB Rd,EA	20h Rd EA	7	7	5	7	5	4	6	6	--	○	Rd ← Rd - [EA]
Compare	CMP Rd,EA	28h Rd EA	7	7	5	7	5	4	6	6	--	○	Rd ← [EA]
Logical And	AND Rd,EA	30h Rd EA	7	7	5	7	5	4	6	6	--	○	Rd ← Rd and [EA]
Logical Or	OR Rd,EA	38h Rd EA	7	7	5	7	5	4	6	6	--	○	Rd ← Rd or [EA]
Logical Xor	XOR Rd,EA	40h Rd EA	7	7	5	7	5	4	6	6	--	○	Rd ← Rd xor [EA]
Add with Scale	ADDS Rd,EA	48h Rd EA	8	8	6	8	6	5	7	7	--	○	Rd ← Rd + [EA]*2
Multiply	MUL Rd,EA	50h Rd EA	57	57	55	57	55	54	56	56	--	○	Rd ← Rd × [EA]
Divide	DIV Rd,EA	58h Rd EA	73	73	71	73	71	70	72	72	--	○	Rd ← Rd / [EA]
Modulo	MOD Rd,EA	60h Rd EA	73	73	71	73	71	70	72	72	--	○	Rd ← Rd % [EA]
Multiply Long	MULL Rd,EA	680h Rd EA	57	57	55	57	55	54	56	56	--	○	(Rd+1,Rd) ← Rd × [EA]
Divide Long	DIVL Rd,EA	70h Rd EA	73	73	71	73	71	70	72	72	--	○	Rd ← (Rd+1,Rd) / [EA], Rd+1 ← (Rd+1,Rd) % [EA]
Shift Left Arithmetic	SHLA Rd,EA	80h Rd EA	8+n	8+n	6+n	8+n	6+n	5+n	7+n	7+n	--	○	Rd ← Rd << [EA]
Shift Left Logical	SHLL Rd,EA	88h Rd EA	8+n	8+n	6+n	8+n	6+n	5+n	7+n	7+n	--	○	Rd ← Rd << [EA]
Shift Right Arithmetic	SHRA Rd,EA	90h Rd EA	8+n	8+n	6+n	8+n	6+n	5+n	7+n	7+n	--	○	Rd ← Rd >> [EA]
Shift Right Logical	SHRL Rd,EA	98h Rd EA	8+n	8+n	6+n	8+n	6+n	5+n	7+n	7+n	--	○	Rd ← Rd >> [EA]
Jump on Zero	JZ EA	A0h 0h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (Z) PC ← EA
Jump on Carry	JC EA	A0h 1h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (C) PC ← EA
Jump on Minus	JM EA	A0h 2h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (S) PC ← EA
Jump on Overflow	JO EA	A0h 3h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (V) PC ← EA
Jump on greater than	JGT EA	A0h 4h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (not (Z or (S xor V))) PC ← EA
Jump on greater or equal	JGE EA	A0h 5h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	if (not (S xor V)) PC ← EA
Jump on less or equal	JLE EA	A0h 6h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (Z or (S xor V)) PC ← EA
Jump on less than	JLT EA	A0h 7h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (S xor V) PC ← EA
Jump on Non Zero	JNZ EA	A0h 8h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (not Z) PC ← EA
Jump on Non Carry	JNC EA	A0h 9h EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (not C) PC ← EA
Jump on Non Minus	JNM EA	A0h Ah EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (not S) PC ← EA
Jump on Non Overflow	JNO EA	A0h Bh EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (not V) PC ← EA
Jump on higher	JHI EA	A0h Ch EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (not (Z or C)) PC ← EA
Jump on lower or same	JLS EA	A0h Eh EA	4/5	4/5	--	--	--	--	4/5	--	--	x	If (Z or C) PC ← EA
Jump	JMP EA	A0h Fh EA	5	5	--	--	--	--	5	--	--	x	PC ← EA
Call subroutine	CALL EA	A8h 0h EA	6	6	--	--	--	--	6	--	--	x	[-SP] ← PC, PC ← EA
Input	IN Rd,EA	B0h Rd EA	7	--	--	--	--	--	6	6	--	x	Rd ← IO[EA]
Output	OUT Rd,EA	B8h Rd EA	6	--	--	--	--	--	5	5	--	x	IO[EA] ← Rd
Push Register	PUSH Rd	C0h Rd 0h	--	--	--	--	--	--	--	--	5	x	[-SP] ← Rd
Pop Register	POP Rd	C4h Rd 0h	--	--	--	--	--	--	--	--	6	x	Rd ← [SP++]
Return from Subroutine	RET	D0h 0h 0h	--	--	--	--	--	--	--	--	6	x	PC ← [SP++]
Return from Interrupt	RETI	D4h 0h 0h	--	--	--	--	--	--	--	--	9	x	FLAG ← [SP++], PC ← [SP++]
Enable Interrupt	EI	E0h 0h 0h	--	--	--	--	--	--	--	--	5	x	割込み許可
Disable Interrupt	DI	E4h 0h 0h	--	--	--	--	--	--	--	--	5	x	割込み禁止
Supervisor Call	SVC	F0h 0h 0h	--	--	--	--	--	--	--	--	12	x	システムコール
Halt	HALT	FFh 0h 0h	--	--	--	--	--	--	--	--	5	x	CPU停止

アドレッシングモード（上の表中EAの詳細）について

アドレッシングモード	略記	ニーモニック (EA部分の標記方法)	命令フォーマット		EA(実効アドレス)の決め方						
			第1ワード	第2ワード	略記	解説					
Direct	Drc	OP Rd,Dsp	OP+0 Rd0h	Dsp	[Dsp]	Dsp番地					
Indexed	Index	OP Rd,Dsp,Rx	OP+1 RdRx	Dsp	[Dsp+Rx]	(Dsp+Rxレジスタの内容) 番地					
Immediate	Imm	OP Rd,#Imm	OP+2 Rd0h	Imm	Imm	Immそのもの					
FP Rerative	FP Rlt	OP Rd,Dsp4,FP	OP+3 RdD4	--	[Dsp4+FP]	(D4を符号拡張した値×2 + FPレジスタの内容) 番地(D4=Dsp4/2)					
Register	Reg	OP Rd,Rs	OP+4 RdRs	--	Rs	Rsレジスタの内容					
4bit Signed Immediate	Imm4	OP Rd,#Imm4	OP+5 RdI4	--	Imm4	14を符号拡張した値そのもの					
Register Indirect	Indr	OP Rd,0,Rx	OP+6 RdRx	--	[Rx]	Rxレジスタの内容番地					
Byte Register Indirect	B Indr	OP Rd,@Rx	OP+7 RdRx	--	[Rx]	Rxレジスタの内容番地 (但し番地の内容は8 bitデータ)					
Other	Othr	OP Rd	OP Rd0h	--	--	なし					
		OP	OP_0h0h	--	--	なし					

注 4

※アセンブリ言語でDspとDsp4、ImmとImm4の標記は同じ（値によりアセンブリが自動判定）。

※FP相对で、Dsp4は-16～+14の偶数

注 1 : MUL, DIV命令ではRdは偶数番号のレジスタ

注 2 : D4はDsp4(4bitディスプレースメント)の1/2の値

注 3 : I4はImm 4 (4 bit即値)のこと

注 4 : アドレッシングモードによりOPの値が変化する

図 A.3 命令表

A.2 メモリマップと I/O マップ

図 A.4 に TaC のメモリマップと I/O マップを示す。メモリや I/O は 8 ビット毎にアドレス付けされており、8 ビットデータ、16 ビットデータのどちらも読み書きできる。アドレッシング・モードによって、8 ビットデータと 16 ビットデータの区別をする。16 ビットデータは偶数アドレスを指定してアクセスしなければならない。

A.2.1 メモリ空間

TaC のメモリ空間は 0000h から FFFFh の 64KiB である。16 ビットデータは偶数アドレスからの 2 バイトに配置され、偶数アドレスを指定してアクセスする。16 ビットデータにアクセスするには、Byte Register Indirect モード以外のアドレッシング・モードを用いる。8 ビットデータにアクセスするには、Byte Register Indirect モードを用いる。

メモリ空間の最初から 56KiB は自由に使用できるメモリであり、ここに TacOS のカーネルやユーザプロセスがロードされる。E000h から EFFFh までは VRAM が配置されている。VRAM に ASCII コードを書き込むと対応する文字がディスプレイに表示される。VRAM のアドレスがディスプレイの表示位置に対応する。F000h から FFDFh は IPL (ROM) が配置される。IPL は μ SD カードから TacOS を読み出して起動する。FFE0h から FFFFh は割込みベクタ領域である。16 種類の割込みに対応するハンドラの入口番地を TacOS がセットする。

A.2.2 I/O 空間

TaC の I/O 空間は 00h から FFh の 256B である。I/O 空間のアドレス幅は 8 ビットだが、IN, OUT 命令では I/O アドレスが 16 ビットで表現される。そこで、I/O アドレスの上位 8 ビットは 00h になるようにする。上位 8 ビットが 00h 以外になった場合の動作は保証されない。

メモリ空間と同様に 8 ビットデータと 16 ビットデータの両方を読み書きできる。8 ビットデータと 16 ビットデータの区別は、IN, OUT 命令のアドレッシングモードにより行う。I/O の 8 ビットデータにアクセスするには、Byte Register Indirect モードを用いる。

メモリマップ			I/Oマップ			
+0番地	+1番地		+0番地	+1番地		
0000h	RAM(56kB)	RAM	00h	Timer0(In:現在値/Out:周期)		
0002h			02h	Timer0(In:フラグ/Out:コントロール)		
0004h			04h	Timer1(In:現在値/Out:周期)		
...			06h	Timer1(In:フラグ/Out:コントロール)		
DFFEh			08h	00H	SIO-Data	
E000h	予約 (アトリビュート)	VRAM(2kB)	0Ah	00H	SIO-Stat/Ctrl	
...			0Ch	00H	PS2-Data	
EFFEh			0Eh	00H	PS2-Stat/Ctrl	
F000h	IPL(4064B)	ROM	10h	00H	uSD-Stat/Ctrl	
...			12h	uSD-MemAddr		
FFDEh			14h	uSD-BlkAddrH		
FFFEh	Timer0		16h	uSD-BlkAddrL		
FFE0h	Timer1		18h	00H	拡張ポート(In/Out)	
FFE2h	INT2		1Ah	00H	ADC参照電圧(Out)	
FFE4h	INT3		1Ch	00H	I/Oポート(予約)	
FFE6h	SIO 受信		1Eh	00H	モード(In)	
FFE8h	SIO 送信		20h	00H	ADC(CH0)	
FFEAh	PS2 受信		22h	00H	ADC(CH1)	
FFECh	PS2 送信		24h	00H	ADC(CH2)	
FFEEh	PS2 送信		26h	00H	ADC(CH3)	
FFF0h	uSD		28h	空き	空き	
FFF2h	ADC			
FFF4h	不正(奇数)アドレス		F4h	下限アドレス		
FFF6h	上下限アドレス違反		F6h	上限アドレス		
FFF8h	ゼロ除算(※1)		F8h	データレジスタ(Out)/データSW(IN)		
FFF9h	特権違反(※1)		FAh	アドレスレジスタ(IN)		
FFFCCh	未定義命令(※1)		FCh	00H	ロータリーSW(IN)	
FFFFEh	SVC(※1)		FEh	00H	機能レジスタ(IN)	

※1：マイクロプログラムにより発生

IPLレーティングのエントリーポイント

番地	関数	意味
F000h	_ipl()	IPLに戻る

IOポートの詳細

番地	名称	ビット構成	説明
02h	Timer0 コントール	1000 ... 000S	I=Enable Interrupt, S=Start
04h	Timer1 コントール	1000 ... 000S	I=Enable Interrupt, S=Start
0Bh	SIO-Stat (in)	TR00 0000	T=Transmitter Ready, R=Receiver Ready
0Bh	SIO-Ctrl (out)	TR00 0000	T=Enable Transmitter Interrupt, R=Enable Receiver Interrupt
0Fh	PS2-Stat (in)	TR00 0000	T=Transmitter Ready, R=Receiver Ready
0Fh	PS2-Ctrl (out)	TR00 0000	T=Enable Transmitter Interrupt, R=Enable Receiver Interrupt
11h	uSD-Ctrl	0000 EIRW	E=INT_ENA, I=INIT, R=READ, W=WRITE
13h	uSD-Stat	0000 IE00	I=IDLE, E=ERROR
1Fh	モード	0000 00MM	MM : 00=TeC, 01=TaC, 10=DEMO1, 11=DEMO2
FDh	ロータリー-SW(IN)	000S SSSS	SSSS : 0=G0, 1=G1, … 11=G11, 12=FP, 13=SP, 14=PC, 15=FLAG, 16=MD, 17=MA
FFh	機能レジスタ(IN)	0000 FFFF	FFFF : 0=ReadReg, 1=WriteReg, 13=ReadMem, 14=WriteMem

図 A.4 メモリマップと I/O マップ

オペレーティングシステム Ver. 0.0.0

発行年月 2017年10月 Ver.0.0.0

発 行 独立行政法人国立高等専門学校機構

徳山工業高等専門学校

情報電子工学科 重村哲至

〒745-8585 山口県周南市学園台

sigemura@tokuyama.ac.jp