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

目次

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. ユーザインタフェース（GUI，CLI）
4. ユーティリティソフトウエア（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，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．

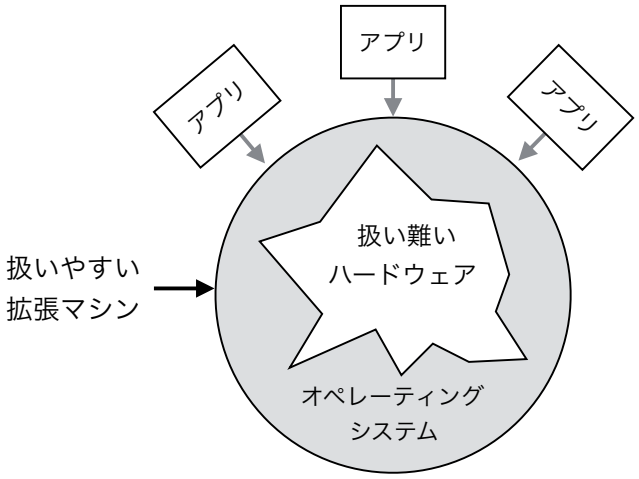


図 1.1　抽象化

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フェース（使いやすい**抽象化**されたインタフェース）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

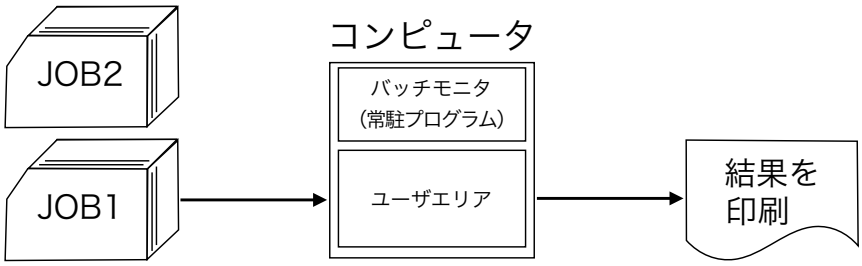


図 1.3　バッチ処理

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，トランジスタの時代）

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．そこで人手を介する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．

図 1.4　紙カード

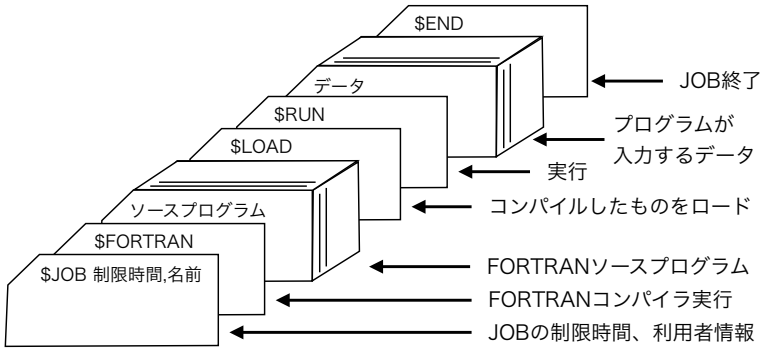


図 1.5　ジョブの構成

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．

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する恐

ウィキメディア / 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

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）して入出力を行う必要がある．

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**

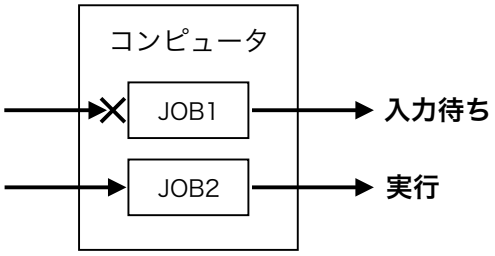


図 1.7　マルチプログラミングシステム

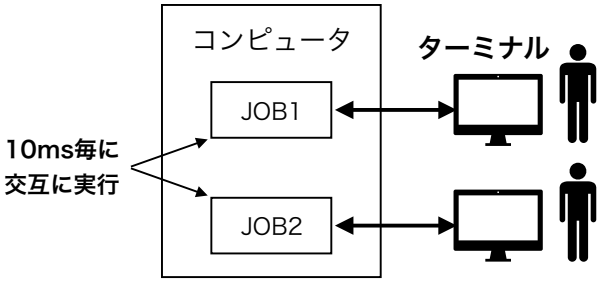


図 1.8　タイムシェアリングシステム

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，その中で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

写真：http://commons.wikimedia.org/wiki/File:Televideo925Terminal.jpg （パブリックドメイン）

図 1.9　ターミナル

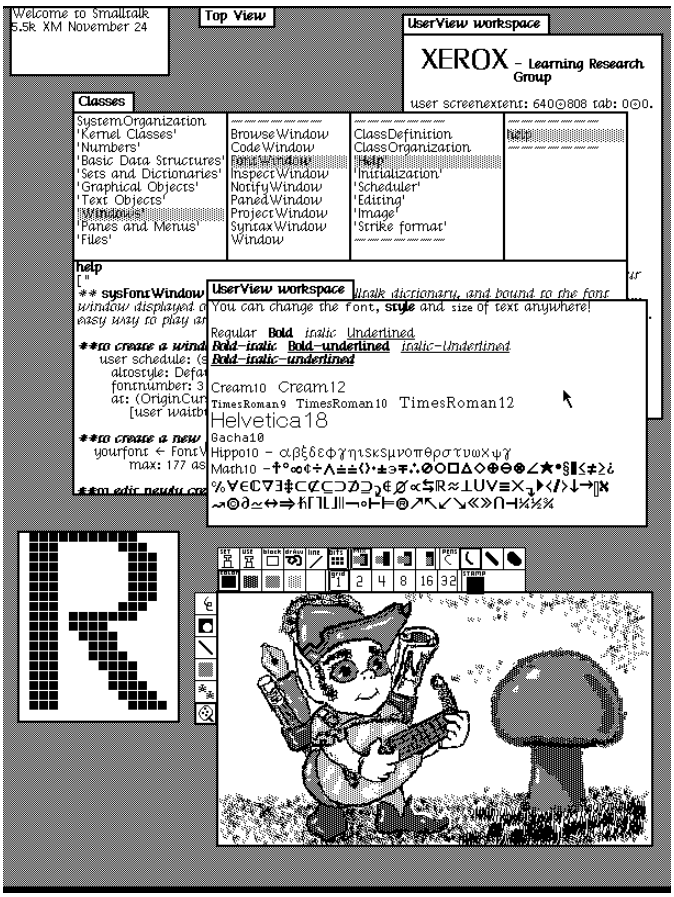
るプロジェクトである．強力な一台のコンピュータで都市一つ分のコンピュータサービスを提供する構想だった．完成までに長い期間を要し（その間にベルと GE が脱落し），商業的には失敗であったがその後のオペレーティングシステムに影響を与える多くのアイデアが出てきた．

- 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 Digtal Reserch 社が CP/M（Control Program for Microcomputer）と呼ばれる 8bit



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

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．

ウィキメディア / Bundesarchiv, B 145 Bild-F077948-0006 / Engelbert Reineke / CC-BY-SA 3.0 de

図 1.11　IBM PC

ウィキメディア / w:User:Grm wnr / File:Macintosh 128k transparency.png /GFDL

図 1.12　初代 Macintosh

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

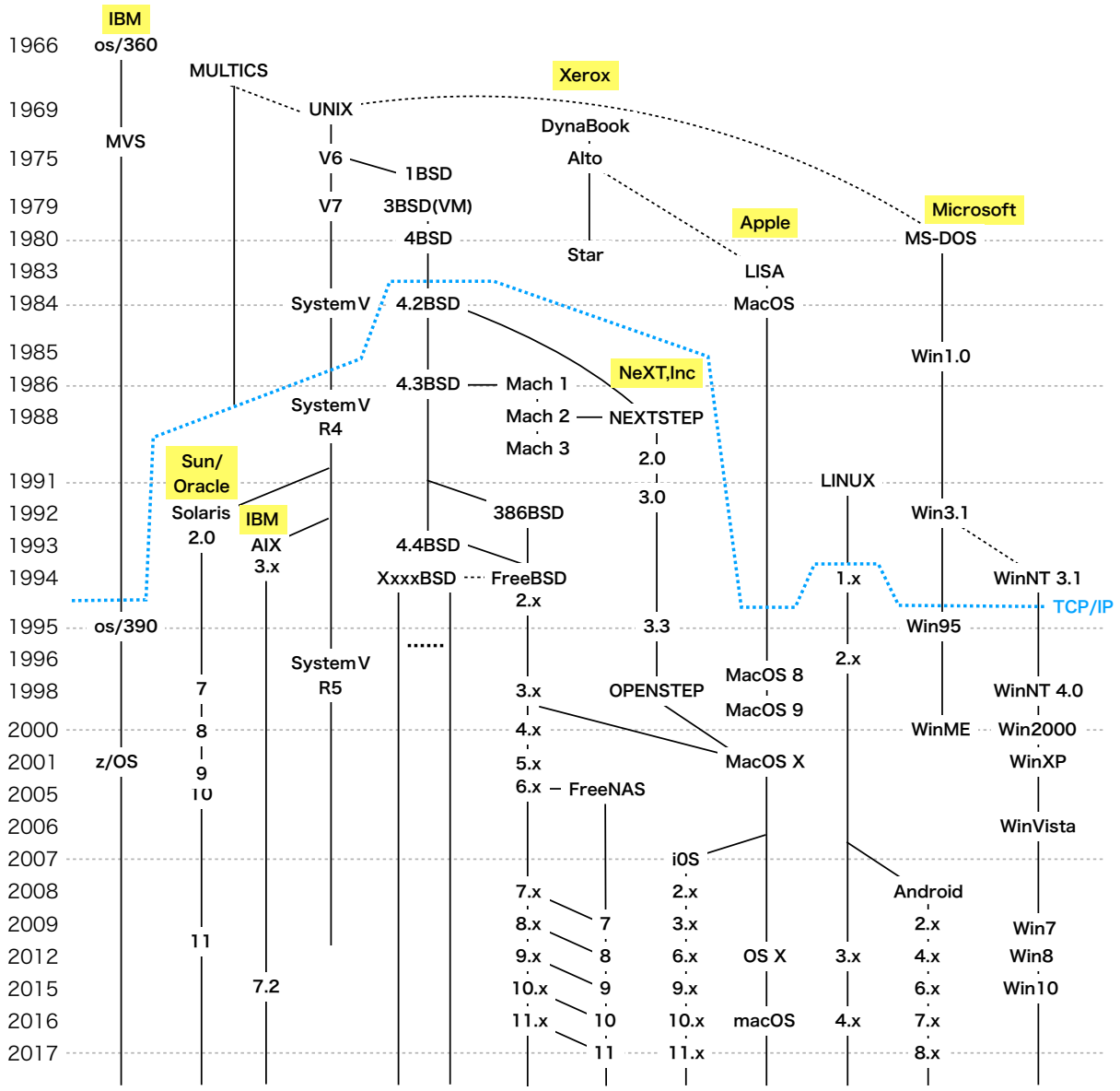
システムに生まれ変わった．

- 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



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

[24]．これ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．

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無い．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．

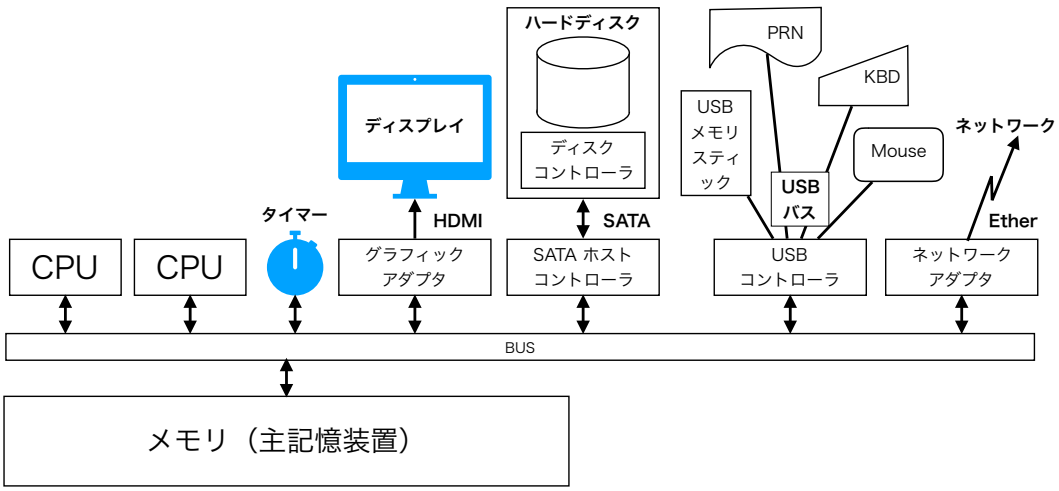


図 2.1　ハードウエア構成

2. メモリ（主記憶装置）
   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．
   (a) 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．
   (b) 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．
   (c) 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．
   (d) 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．

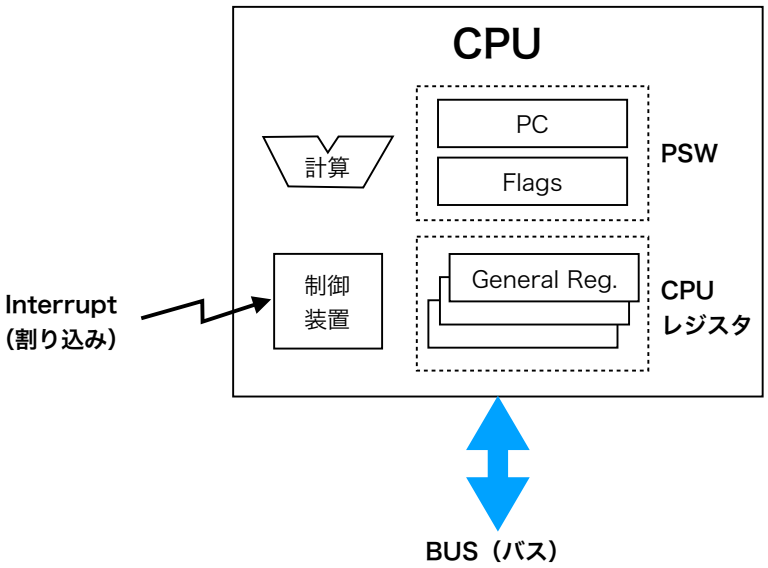


図 2.2　CPU の構成

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．

[*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] 教科書によっては，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．

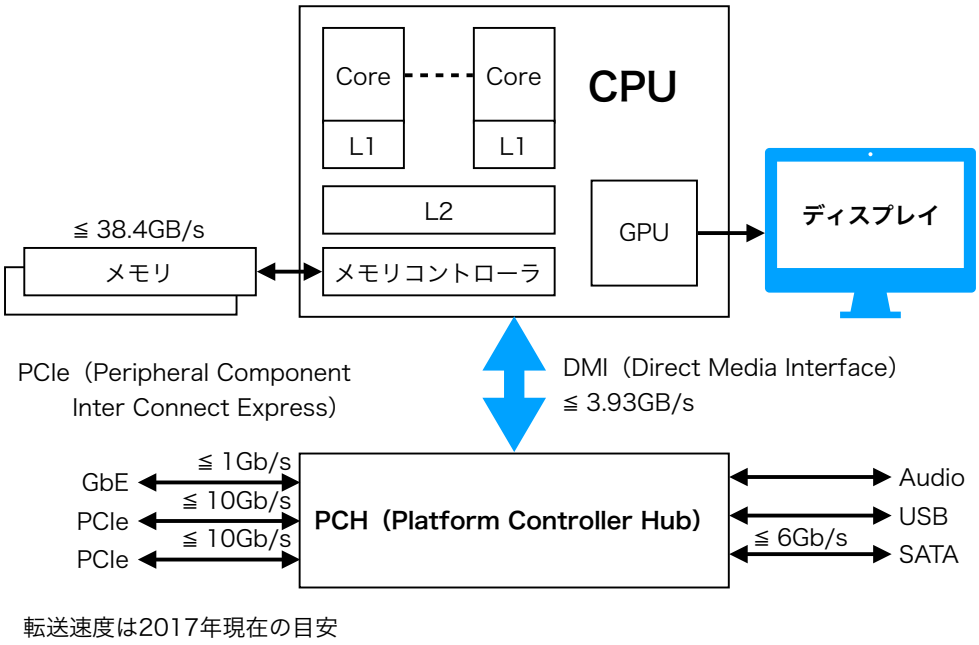


図 2.3　デスクトップ PC の構成

2.3　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．

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．

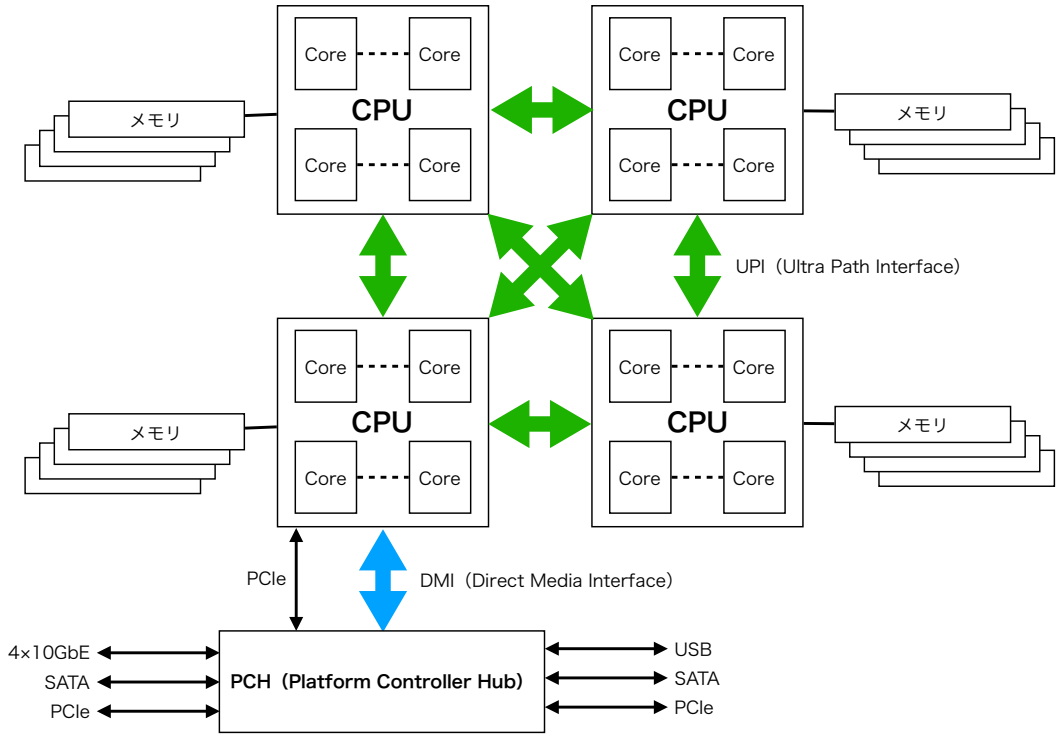


図 2.4　サーバ PC の構成

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．

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. **サービスモジュール**
   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

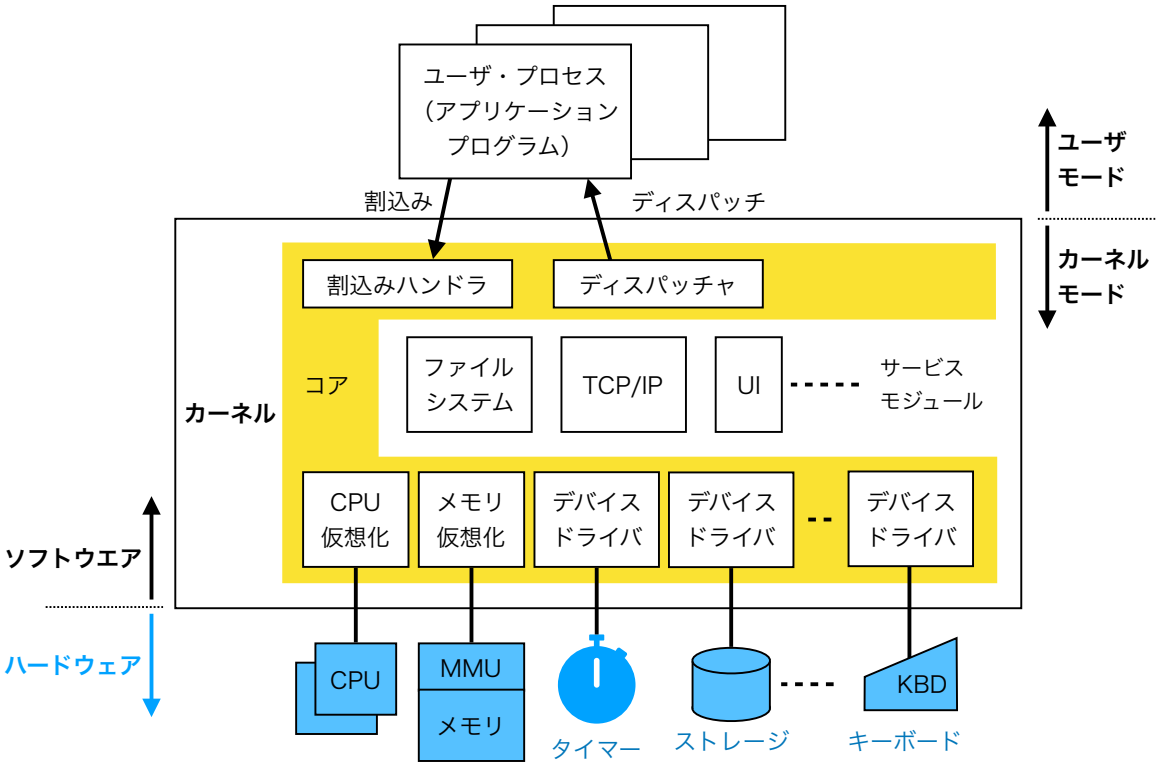


図 2.5　オペレーティングシステムの構造

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．

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．

[*4] CPU によっては TRAP 命令，INT 命令と呼ばれることもある．

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ような手順で処理がされる．

1. 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．
2. 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．
3. 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 と呼ぶことにする．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？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．

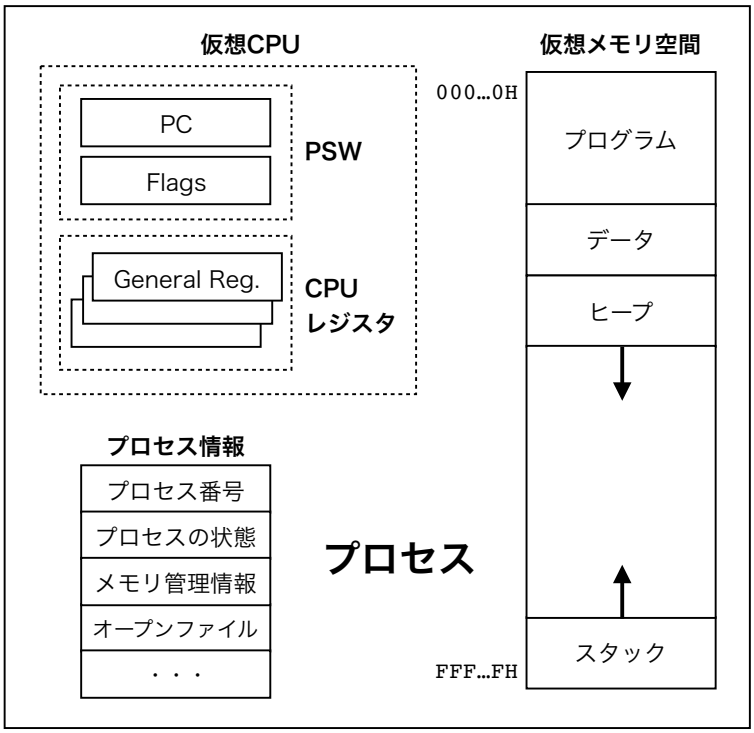


図 2.6　プロセスの構造

(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．

[*5] これらは UNIX の ps コマンドで表示することができる．

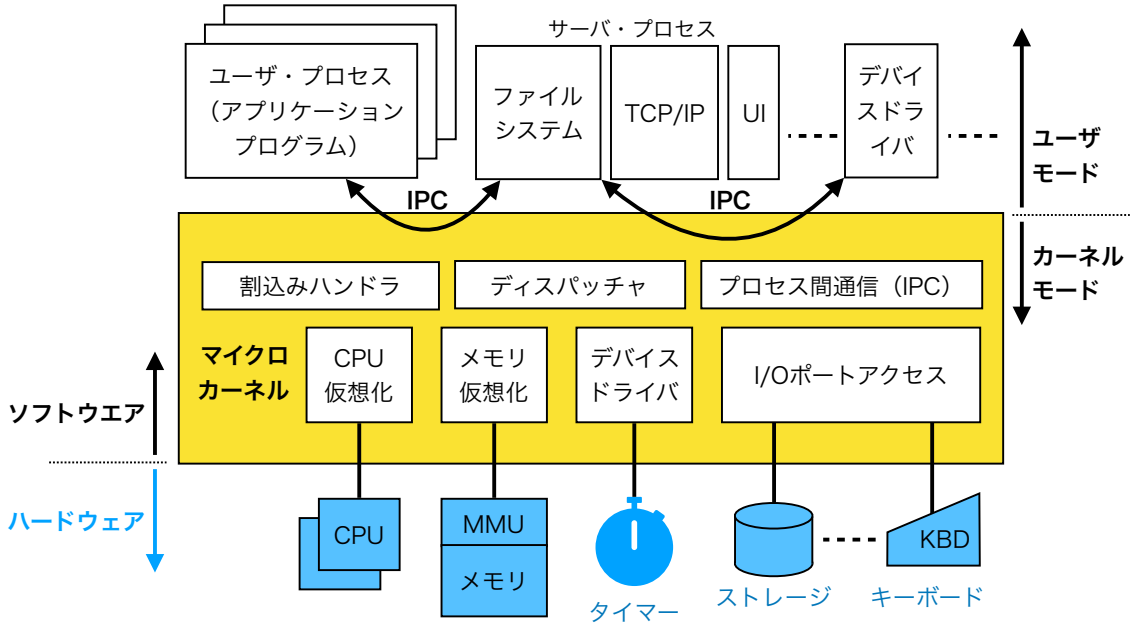


図 2.7　マイクロカーネル方式

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．

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

[*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].
[*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．

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 とプロセス切り換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．**

2.6　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」と言う用語が，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．

2.6.1　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[*8]．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 2.8 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[*9]，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は製品は，VMware Workstation，VMware Fusion，VirtualBox[*10] 等である．

[*8] 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．

[*9] 完全にソフトウェアで行う場合もある．

[*10] 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．この Windows の中で Xlinix ISE WebPACK が使用できる．

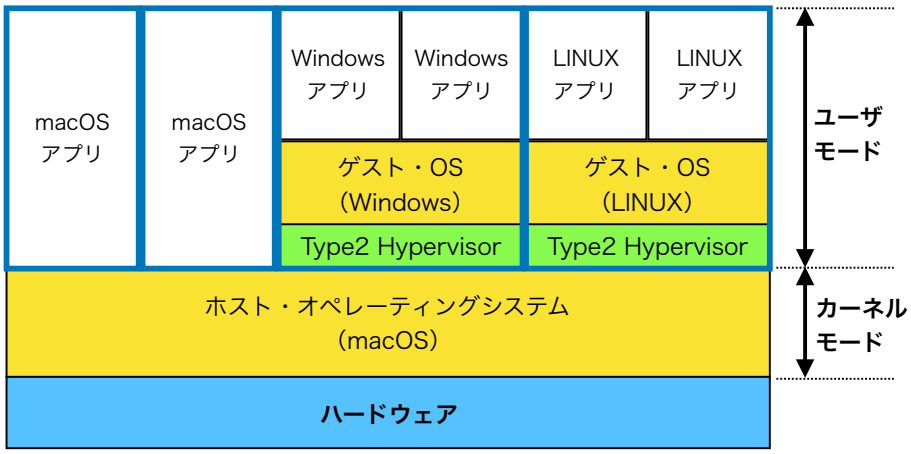


図 2.8　Type 2 ハイパーバイザ

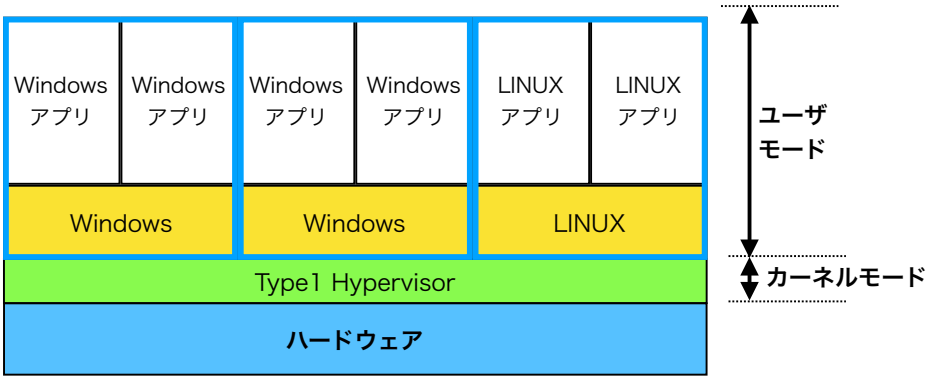


図 2.9　Type 1 ハイパーバイザ

2.6.2　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．

2.6.3　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，VirtulBox 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．

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．

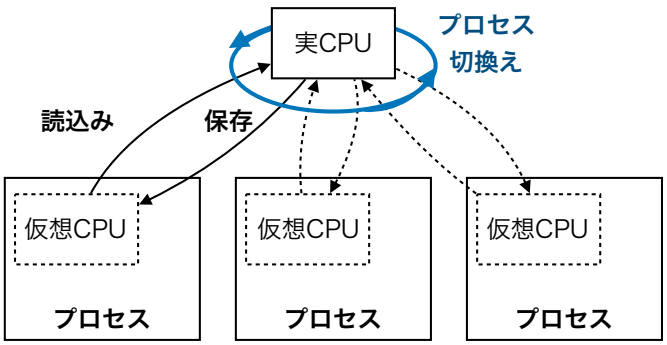


図 3.1　時分割多重による CPU の仮想化

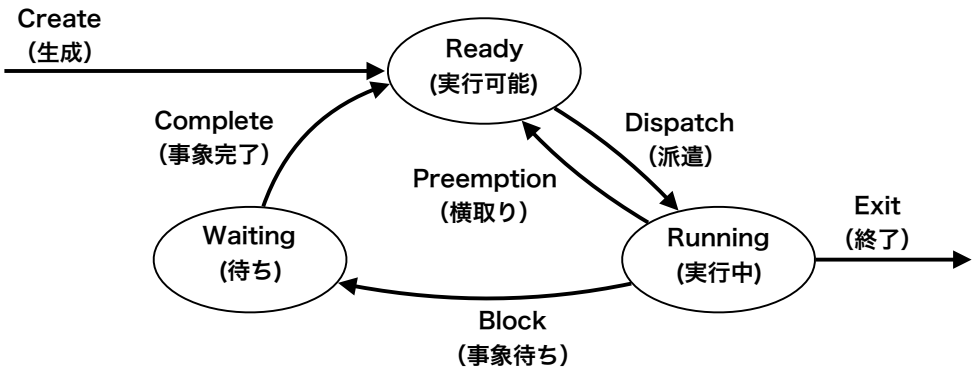


図 3.2　プロセスの状態遷移

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（クオンタムタイム）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 遷移することがある．
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（クオンタムタイム）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．
   (a) CPU の（PC を含む）PSW がスタックに保存される．
   (b) CPU の実行モードがカーネルモードに切り換わる．
   (c) 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. サービスモジュール等
   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させることがある．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ブは 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．

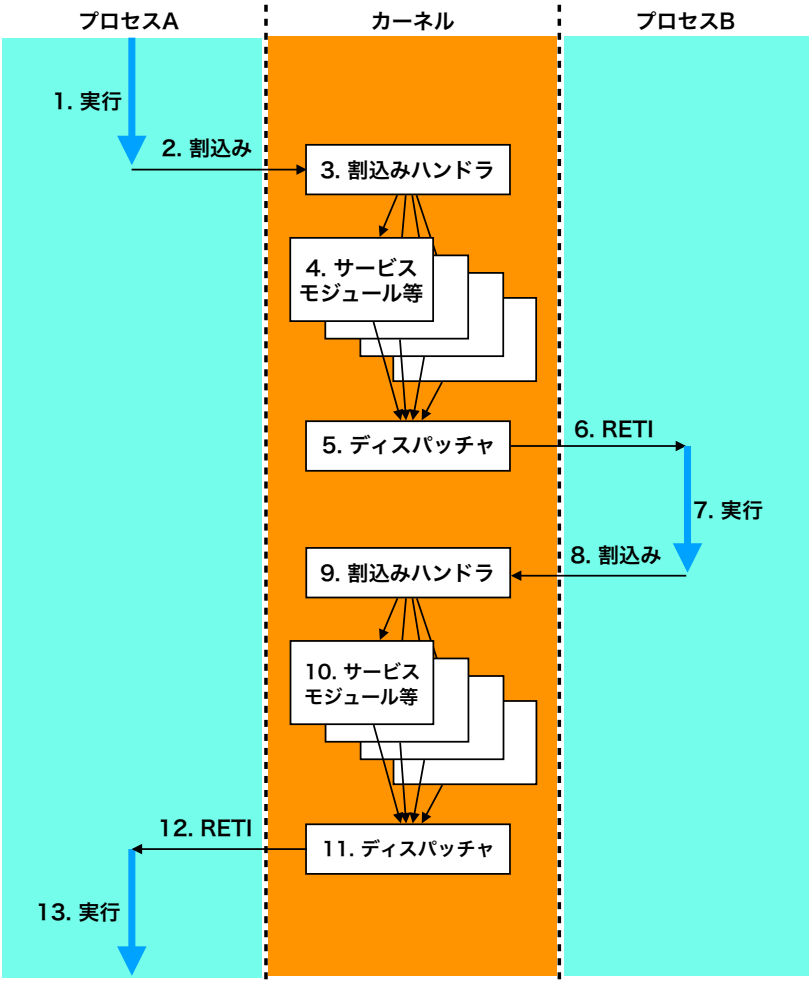


図 3.3　プロセスの切換え

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．

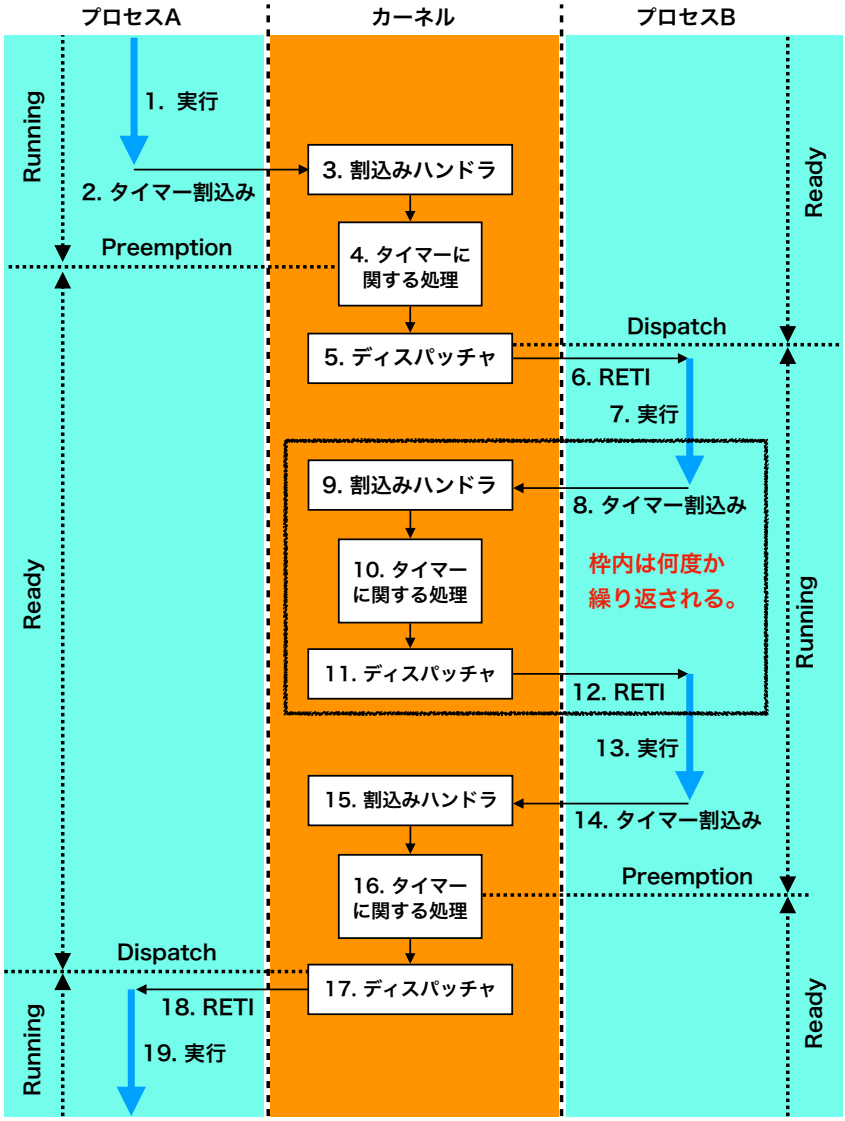


図 3.4　プロセスの切換えの例

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（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 がクオ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，クオンタ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，クオンタ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
    クオンタ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

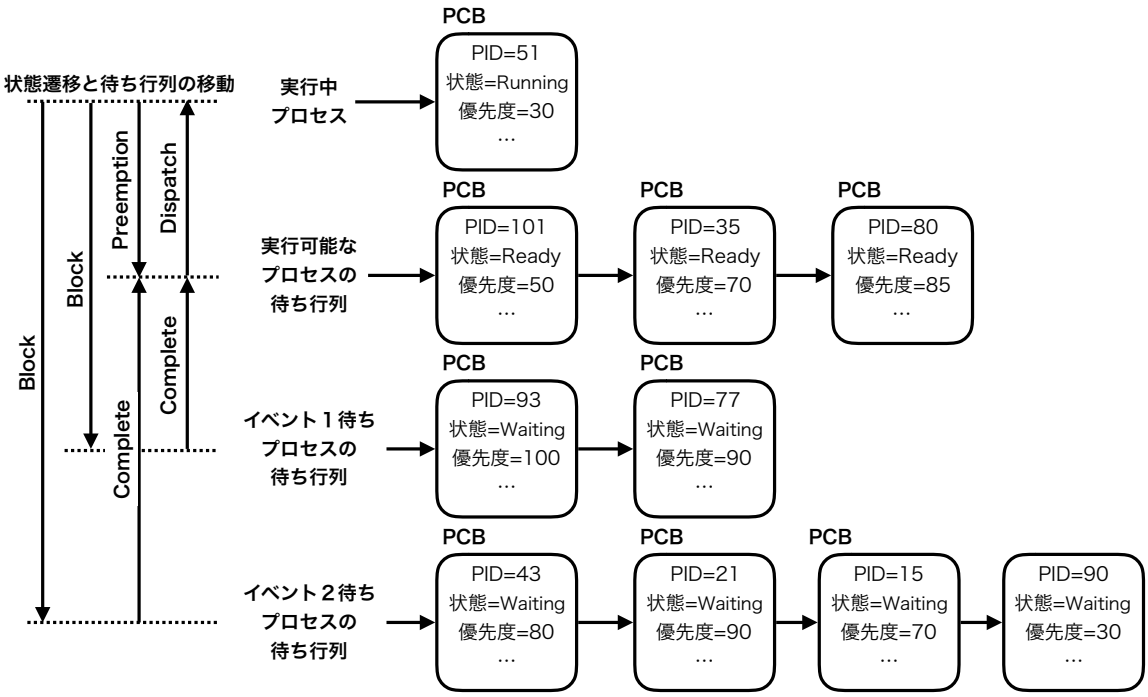


図 3.5　PCB のリスト

- 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
- 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も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`）

[*3] TacOS の詳細は https://github.com/tctsigemura/TacOS を参照のこと．
[*4] 図 3.6 は TacOS のソースコード https://github.com/tctsigemura/TacOS/blob/master/os/kernel/process.h の一部である．

```
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

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）
  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

[*5] 詳しくは後の章で解説する．
[*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．

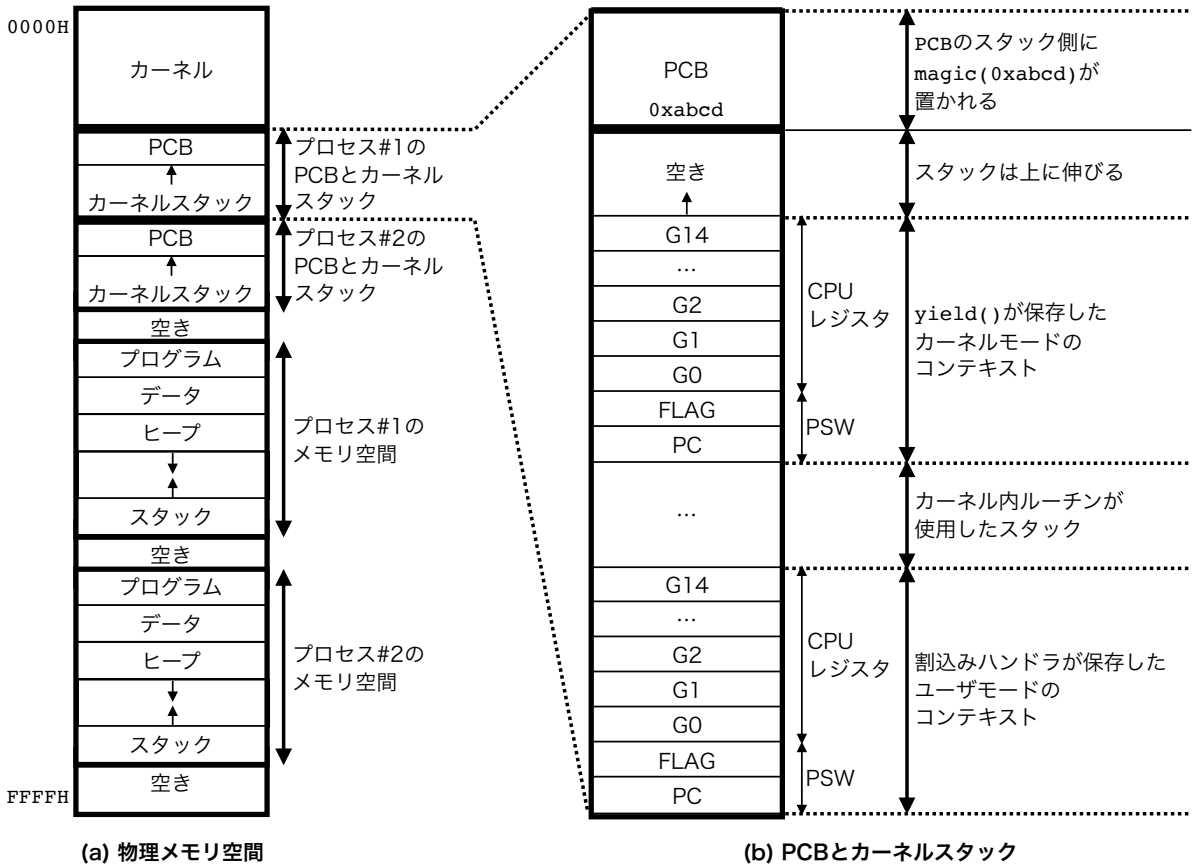


図 3.7　TacOS のメモリ配置

プロセス 1 の 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
  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

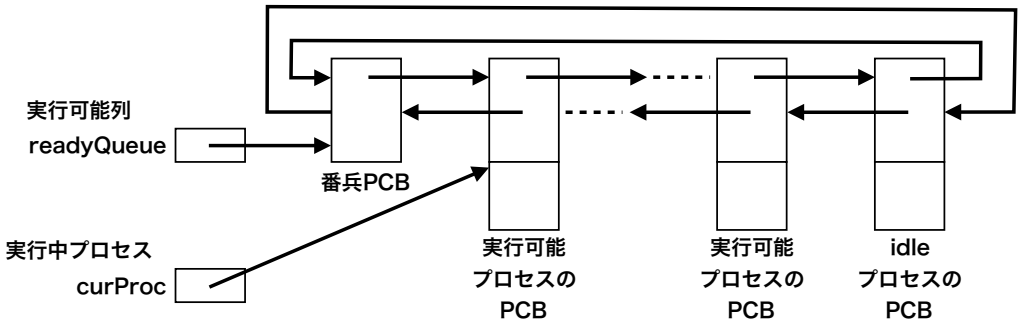


図 3.8 TacOS の実行可能列

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．

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．ここまでの処理でコンテキストの保存が完了した．

[*7] 図 3.9 は TacOS のソースコード https://github.com/tctsigemura/TacOS/blob/master/os/kernel/dispatcher.s の一部である．プログラムは TaC のアセンブリ言語で記述してある．

[*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．

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**」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（6 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される．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

```
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

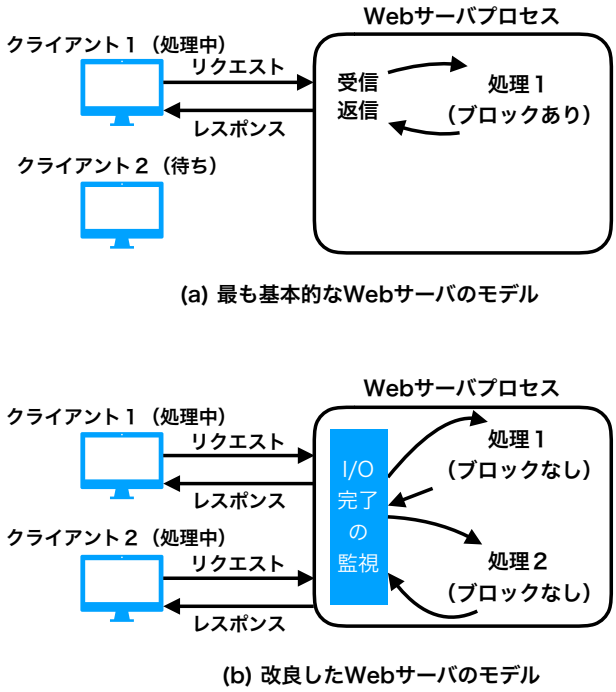


図 3.10　マルチプログラミングを用いない Web サーバ

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，同時には一つの CPU しか働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

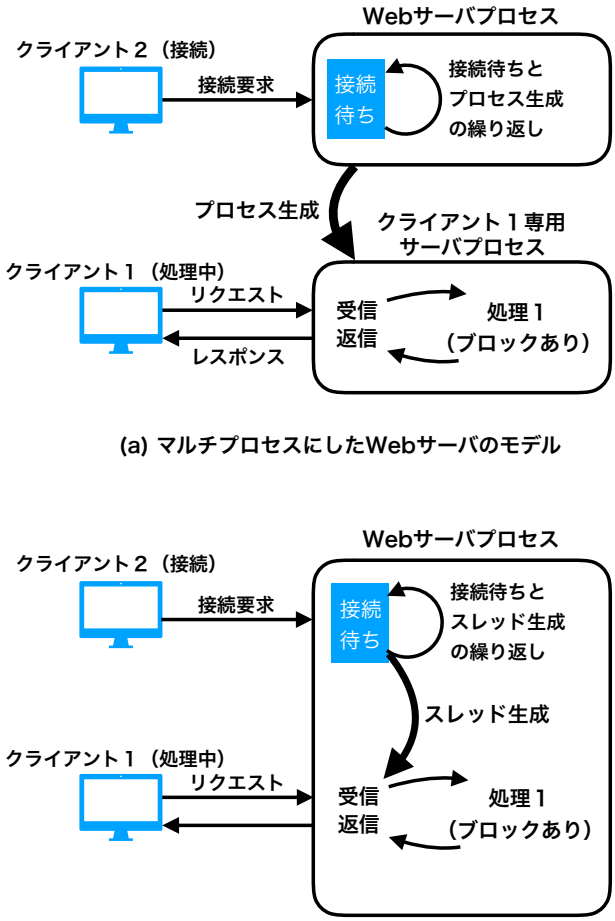


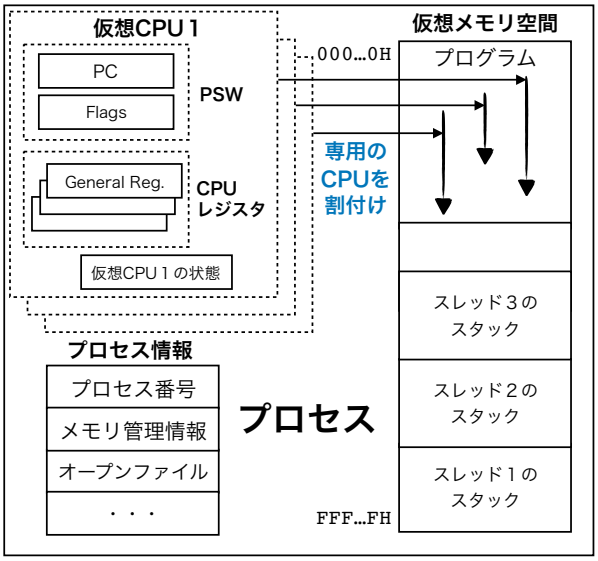
図 3.11　マルチプログラミングを用いる Web サーバ

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．

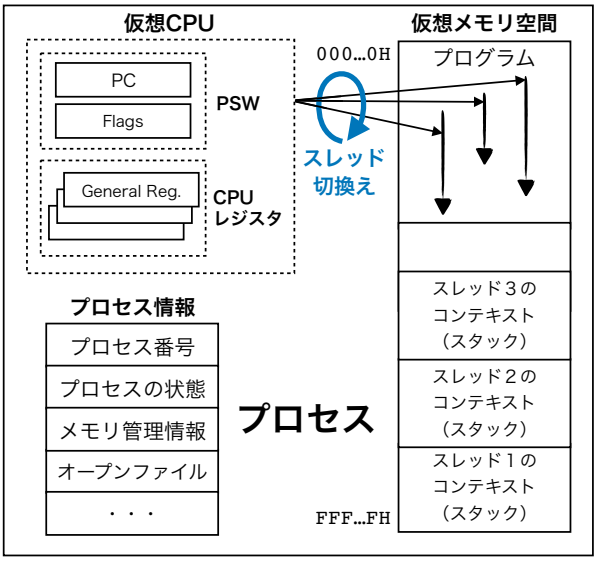
3.6.2　スレッドの形式

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



(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

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．

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．

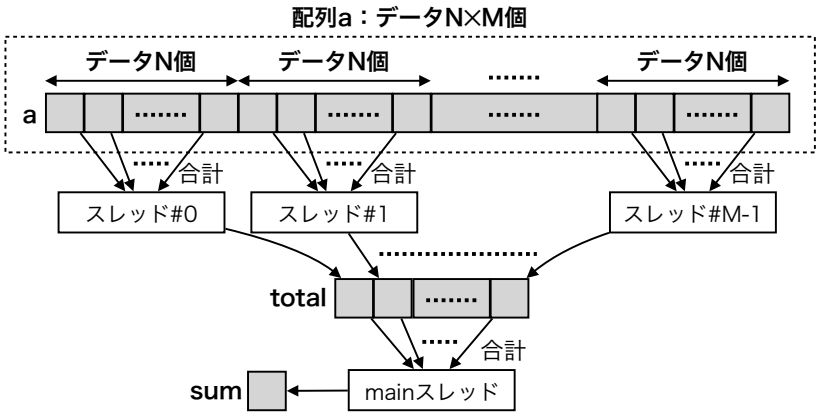


図 3.13　M 個のスレッドで手分けして合計を計算する様子

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）の工夫で単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，真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．

[*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

	スレッド数 (M)・データ件数 (M*N)									
M	1	2	3	4	5	6	7	8	9	10
N	10,000	5,000	3,333	2,500	2,000	1,666	1,428	1,250	1,111	1,000
M*N	10,000	10,000	9,999	10,000	10,000	9,996	9,996	10,000	9,999	10,000
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

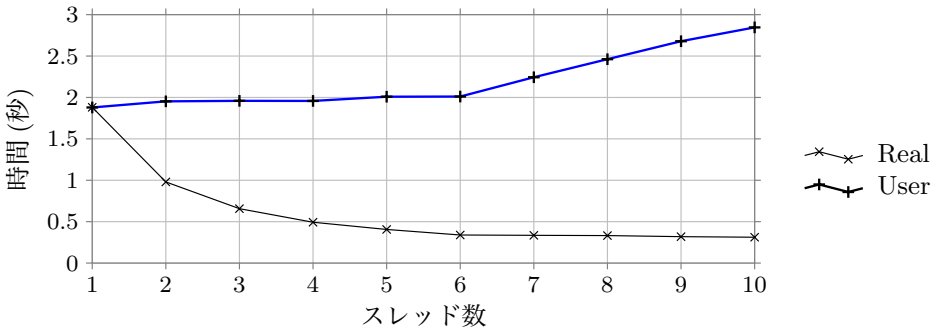


図 3.15　スレッド数による実行時間の変化

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．

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，グラフにしたものを図 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

[*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．

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[*16]．

[*16] この計測結果からは，ハイパースレッディング・テクノロジーは，性能の改善に余り寄与していないように見える．

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ために**省エネルギー**が評価基準になり得る．

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．

- メインフレームで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**

[*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．

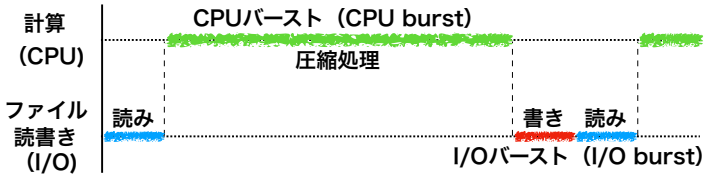
表 4.1　スケジューリングの目標

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

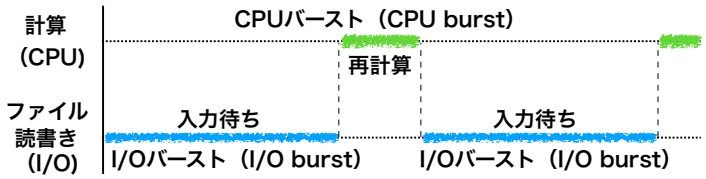
を優先する．例えば，コンテキストスイッチにも処理時間が必要なので，プリエンプションを行わないスケジューリング方式を採用し，コンテキストスイッチの回数を少なくすること等が考えられる．また，ユーザが結果を早く受取ることができるように，**ターンアラウンド時間**にも気を使う必要がある．

- ネットワークサーバは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は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．
- ノートパソコンやスマートフォンのようなモバイルデバイスでは，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の場合，**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．



(a) CPUバウンド（CPU-bound）プロセス



(b) I/Oバウンド（I/O-bound）プロセス

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．

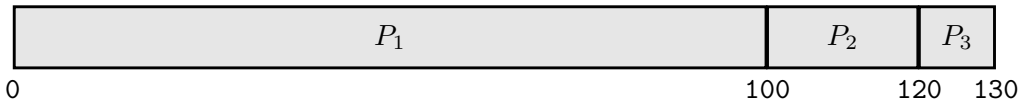
4.4.1　First-Come, First-Served（FCFS）スケジューリング

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．

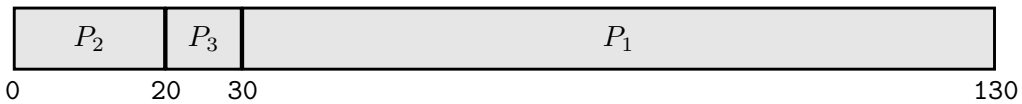
FCFS 方式は実行可能列を FIFO にするだけで実現できるが性能は良くない．例えば次の三つのプロセスが時刻 0 で，P_1，P_2，P_3 の順に Ready 状態になったとする．

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，$(100 + 120 + 130)/3 = 117$ 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，$(20 + 30 + 130)/3 = 60$ 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．

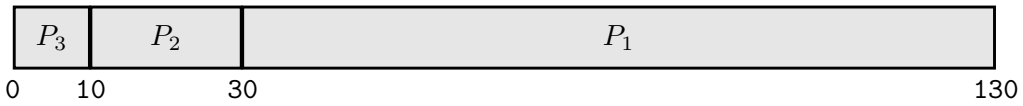
4.4.2　Shortest-Job-First（SJF）スケジューリング

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，$(T_1 + (T_1 + T_2) + (T_1 + T_2 + T_3))/3 = T_1 + T_2 * 2/3 + T_1/3$ 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．

[*3] 皆さんの教科書では SPT のこと．

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 $(10+30+130)/3 = 57$ 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，

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

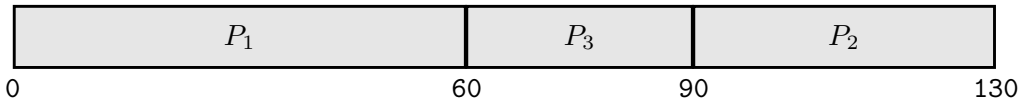
となる．この式は，過去の CPU バースト時間を，最近のものほど大きな重みを付けて平均したものになっている．つまり，次回の CPU バースト時間は，過去の CPU バースト時間と同程度であろうとの仮定に基づいた予測値を計算している．

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 状態になるとき，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．

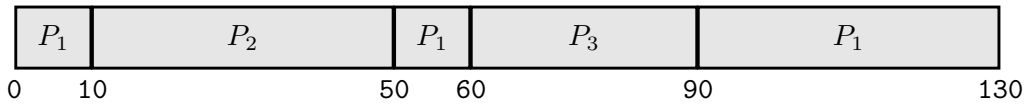
プロセス	到着時刻	CPU バースト時間 (ms)
P_1	0	60
P_2	10	40
P_3	60	30

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．



[*4] 皆さんの教科書では SRPT のこと．

平均ターンアラウンド時間を計算すると，$((60-0)+(90-10)+(130-60))/3 = 70$ 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．



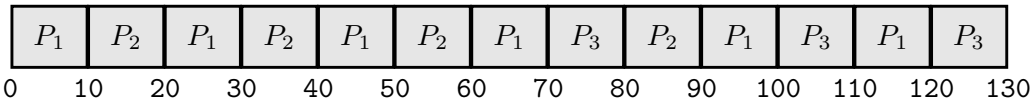
P_2 が到着した時, P_2 の CPU バースト時間(40 ms)の方が P_1 の残り CPU バースト時間(60−10 = 50 ms）より短いので，P_1 はプリエンプションし P_2 が先に実行される．P_3 が到着した時も同様である．平均ターンアラウンド時間を計算すると，$((130-0)+(50-10)+(90-60))/3 = 67$ ms となり，SJF よりも改善されている．

4.4.4　Round-Robin（RR）スケジューリング

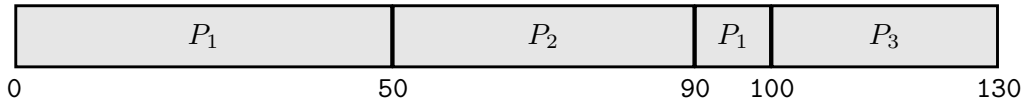
タイムシェアリングシステム (TSS) で使用された方式である．**クォンタム時間（time quantum）**，または，**タイムスライス（time slice）**と呼ばれる 10 ms 〜 100 ms 程度の一定の時間が予め決められている．実行可能列は FIFO になっている．実行可能列の先頭のプロセスに CPU が割り付けられて Running 状態になる．プロセスの実行が**クォンタム時間**連続するとプリエンプションが発生し，プロセスは実行可能列の最後尾に付け加えられる．

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，$q = 10$ 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（$q = 10$ 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，$((120-0)+(90-10)+(130-60))/3 = 90$ ms となる．次に $q = 50$ 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，$((100-0)+(90-10)+(130-60))/3 = 83$ ms となる．$q = 50$ 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，$q = 10$ ms はクォンタム時間が短すぎると言える．

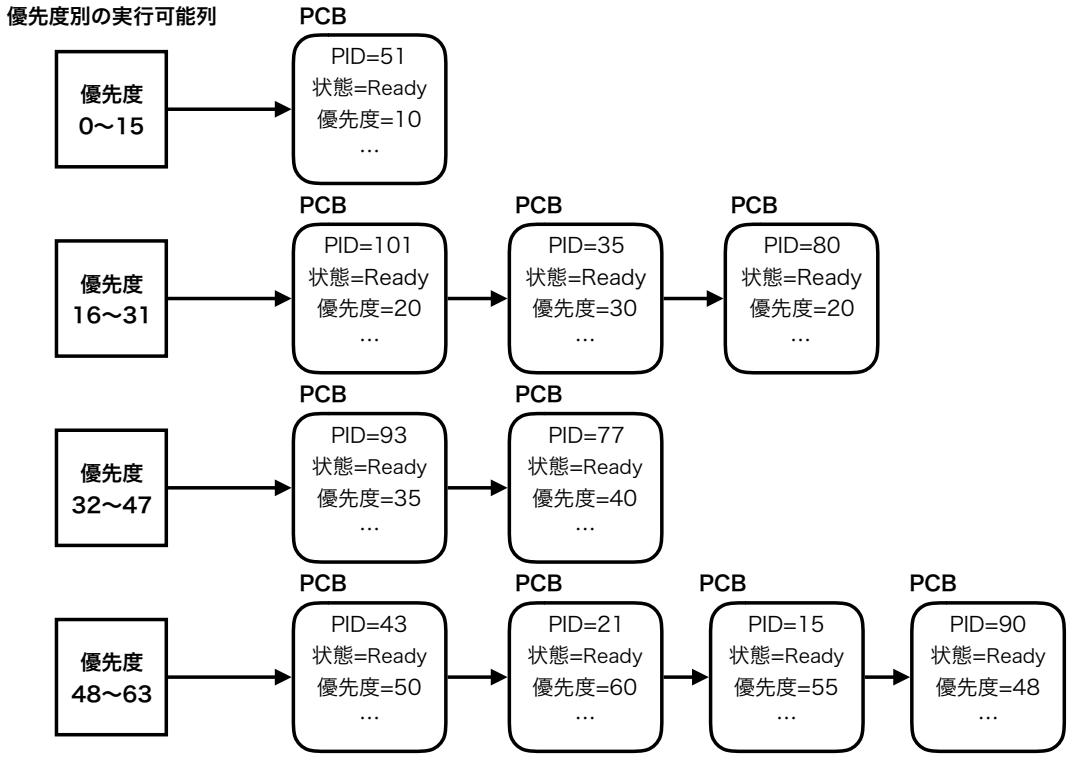


図 4.2　Multilevel Feedback Queue

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（stavation）**が発生することである．これの対策として，実行可能列に留まるプロセスの優先度を徐々に高くしていく**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**

[*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．

```
1  // プロセスキューで 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

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．

[*6] 図 4.3 は TacOS のソースコード https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm の一部である．

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．このような状況

```
                                    // 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

を**競合（Race Condition または Competition）**と呼ぶ.

5.2　クリティカルセクション（Critical Section）

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（Critical Section）**，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　相互排他（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**相互排除**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他

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 Interrrupt）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 Interrrupt）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 が割込みを受付けない[*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他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ても，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他の目的に使用できる．

---

*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      G0, 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場合は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      G0, #0
        ST      G0, 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
        ...

// フラグ
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（Test and Set）命令

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. if (R==0)　$z \leftarrow 1$;　else $z \leftarrow 0$;
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 Zero フラグの値を決定する．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．

フラグのクリアは通常の ST 命令でできる[*3]．TS 命令を用いる場合もクリティカルセクションは割込み禁止で実行する必要がある．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（Busy Waiting）**を始め**デッドロック**に陥る可能性があるからである．「0」をロードしたプロセス（スレッド）は，クリティカルセクションでプリエンプションしてはならない．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 命令のものとよく似ているので解説は省略する．

CAS（Compare And Swap）命令

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. `if` $(T == R0)$ { $[M] \leftarrow R1;\ z \leftarrow 1;$ } `else` { $R0 \leftarrow T;\ z \leftarrow 0;$ }
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，何らかのロックを掛けていた．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．

[*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      G0, #1  // フラグに書き込む値
        SW      G0, 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     G0, #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場合は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      G0, #0
        ST      G0, 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
        ...

// フラグ
FLG     DC      0       // 初期値ゼロ(TS 命令により 1 に書き換えられる)
```

図 5.4　SW 命令の使用例

```
// 口座 account に足し込む
        LD      G0,account
L1      LD      G1,G0
        ADD     G1,receipt
        CAS     G0,G1,account
        JNZ     L1
```

```
// 口座 account から差し引く
        LD      G0,account
L2      LD      G1,G0
        SUB     G1,payment
        CAS     G0,G1,account
        JNZ     L2
```

図 5.5　CAS 命令を用いた口座管理プログラムの例

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）

```
                // 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;                                   flag[1] = true;
turn = 0;                                         turn = 1;
while (turn==0 && flag[1]==true)                  while(turn==1 && flag[0]==true)
   ; // ビジーウェイティング                       ; // 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] = flase;

// 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

5.4　セマフォ（Semaphore）

これまでに紹介してきた相互排他は，主に**ビジーウェイティング（Busy Waiting）**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他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．

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他は，セマフォを実現するためにも使用される．

5.4.1　セマフォの概要

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．セマフォ型（Sempahore）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

[*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] カウンタの値は 0 以上の整数値である．

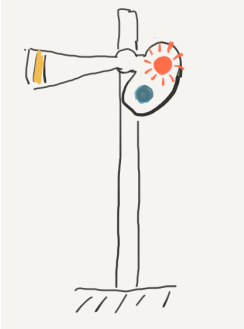


図 5.7　腕木式信号機

```
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

ロセス（スレッド）の状態を**待ち (Waiting) 状態**に変える．**ビジーウェイティング (Busy Waiting) 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　セマフォの使用例

セマフォを用いてプロセス間の同期問題の解を示すことができる．

相互排除問題

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

```
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();       // ネットワークから入金を受信する
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

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．

生産者と消費者の問題（Producer-Consumer problem）

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 にしている．

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込み位置を表すローカル変数である．0,1,2,...,N-1,0,1,2,... 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．**

```
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() {           // 生産者スレッド
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

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．**

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．

複数生産者と複数消費者の問題（Producer-Consumer problem**）**

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．

```
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() {             // 生産者スレッド(複数のスレッドで並列実行する)
8    for ( ; ; ) {                     // 生産者スレッドは以下を繰り返す
9      Data d = produce();             // 新しいデータを作る
10     P( &emptySem );                 // リングバッファの空き数をデクリメント
11     P( &inSem );                    // in にロックを掛ける
12     buffer[ in ] = d;               // リングバッファにデータを格納
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() {            // 消費者スレッド(複数のスレッドで並列実行する)
22   for ( ; ; ) {                     // 消費者スレッドは以下を繰り返す
23     P( &fullSem );                  // リングバッファのデータ数をデクリメント
24     P( &outSem );                   // out にロックを掛ける
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

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 procucerThread 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 procucerThread 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機構を追加してある．

```
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      writeRecores( d );              // データを書換える
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() {               // リータスレッド(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

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

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る．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．

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．

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他される．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 セマフォデータ構造

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

[*6] 図は https://github.com/tctsigemura/TacOS/blob/master/os/kernel/process.h の一部である．

```
#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

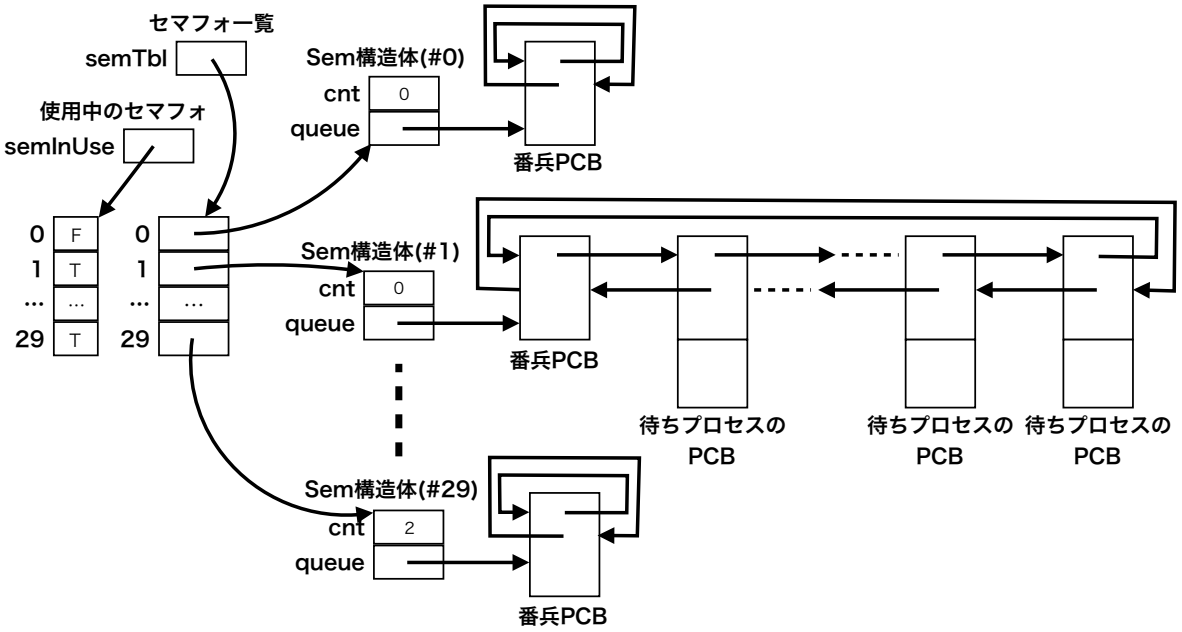


図 5.14　TacOS のセマフォ関連データ構造

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．

```
1  #incude <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() {                   // プロセスの初期化ルーチン
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

5.5.2 セマフォ使用例

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 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．

```
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

5.5.3　セマフォ割当

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他」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

[*7] 図は https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm の一部である．
[*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

(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 行でそれの番号を返す．

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，単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．

---

*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 図は https://github.com/tctsigemura/TacOS/blob/master/os/kernel/kernel.cmm の一部である．

*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．

```
1  // プロセスキュー（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

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

[*14] 図は 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．

```
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

`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．

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．

[*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 ;;  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

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出し側に返している．

[*18] 図は https://github.com/tctsigemura/TacOS/blob/master/os/util/crt0.s の一部である．
[*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

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)（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 Generaton(1965–1980):ICs and Multiprogu-ramming”，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」のためのパーソナルコンピュータ，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」のためのパーソナルコンピュータ」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）

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