■7 群（コンピュータ -ソフトウェア）－3 編（オペレーティングシステム）

4 章 プロセス管理

（執筆者：吉澤康文）［2013 年 2 月 受領］

■概要■

パソコンでは 1 台のコンピュータで複数の仕事を同時に実行するのが一般的になっている．本章では，このように同時に複数の独立した情報処理を実現する原理を説明する．並列で同時に処理するために導入された概念がプロセスである．各々の独立した仕事を複数の処理（プロセス）に分割し，機能分担すると利点が出てくる．これが多重プロセス処理であり，多くの情報システムは，この機能を活用することでプログラム開発を可能とし，高信頼化と高性能化を実現している．本章では，多重プロセス方式ならびに複数の演算装置（CPU）を備えた多重プロセッシングとの関係を説明する．

【本章の構成】

ファイル管理でも述べたように，プロセス管理の基本となる概念，考え方を最初に説明する（4-1 節）．また，プロセス管理を具体的に理解するために，ファイル管理と同じように UNIX におけるプロセス管理の諸機能をとり上げて説明する（4-2 節）．コンピュータシステムの性能目標を維持し向上するには，限りあるコンピュータ資源をプロセスに分配する手段が必要となる．この機能がプロセススケジューリングであり，4-3 節に説明されている．プロセス管理には多くの機能が含まれており 4-2，4-3 節では説明できなかった機能を 4-4 節で説明する．

■7 群 － 3 編 － 4 章

4-1 基本的な考え方

（執筆者：吉澤康文）［2013 年 2 月 受領］

4-1-1 プロセスとは何か

OS においてプロセスは重要な概念である．プロセスは「プログラム実行の抽象化された実体」ということができる．以下，少し具体的に説明する．この本ではプロセスとタスクはほとんど同じ意味で使用することとする．OS によっては異なる意味で使用しているものもある．その定義については 4-1-3 項において説明する．

2 章で述べたように，高性能なコンピュータでは複数のプロセスが同時に実行されている．各プロセスは独立した論理的な処理の流れであるため，入出力要求で一時的に処理が中断しても入出力が完了すれば処理が続行させねばならない．コンピュータは高速で動作するので，多重プログラミング環境で実行されている様子を外部から見ると，1 台のコンピュータであっても複数のプロセスが同時に並列実行されているように見える．そこで，多重プログラミングを疑似的並列処理（Pseudo Parallel）ということがある．

このように，独立した処理を並列に同時実行できるようにする工夫が必要となる．OS では**図 4･1** に示すようなデータ構造を作成し，プロセスを管理している．これをプロセス管理テーブル（PCT：Process Control Table）と呼ぶことがある（詳細は 4-4-1 項にて説明する）．

プロセス管理テーブルには割込みによりプログラムの実行を中断したときの CPU 状態を退避する領域を設けておく．これにより，プロセスが再開したとき各種 CPU レジスタ類に情報を再設定することで処理の連続性が保てる．最初に述べた「プログラム実行の抽象化された実行体」とは，このプロセス管理テーブルにほかならない．

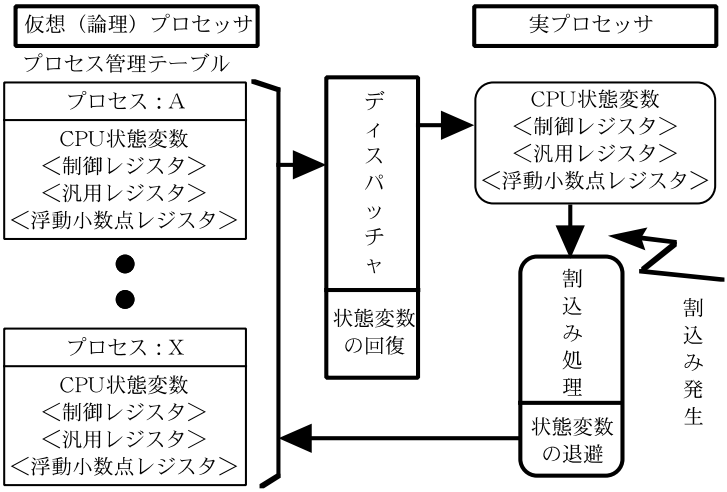

図 4・1　多重プログラミングによる仮想計算機の実現

プロセスの CPU 状態を退避し回復するのは，図 4･1 に示したように割込み処理とディスパッチャ（Dispatcher）である（ディスパッチャはスケジューラとも呼ばれ，4-3 節のスケジューリングの節で詳しく述べる）．これら両者の働きにより，1 台のコンピュータの中に複数の CPU 状態をもつプロセスを作ることができるが，別の角度から見ると，プロセス管理テーブルはコンピュータを論理化した実体ともいえる．

プロセスは，このようにプロセッサという実体がもっている情報を各プロセス管理テーブルに格納しており，1 台のコンピュータ上であたかも複数のCPUが動作しているように見せ掛けられているため，仮想計算機 [*1]（Virtual Machine）ということもある．

4-1-2 プロセススイッチ

多重プログラミング環境でプロセスを切り替えることをプロセススイッチとかコンテックストスイッチ（Context Switch）と呼ぶ．一般的に，プロセススイッチは割込みを契機に行われる．この様子を**図 4･2** に示した．

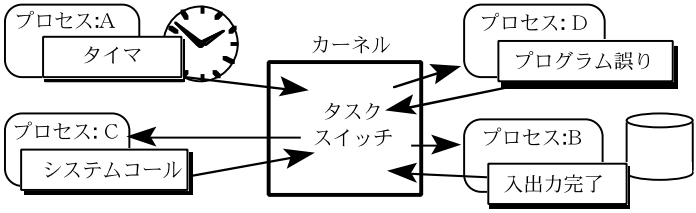

図 4・2　割込みによるプロセススイッチの様子

プロセススイッチの問題点は，切替え時に行わなければならない各種レジスタ類の退避，回復にある．共に主メモリへのアクセスを伴うのでプロセッササイクル（Processor Machine Cycle）を多用する．そこで，過度なプロセススイッチを抑止するなどの工夫がなされている．

4-1-3 プロセスに関する各種の概念

ここではプロセスに関する用語を定義しておく．**表 4･1** には代表的な OS として，IBM 社の MVS，UNIX，そしてカーネギーメロン大学で開発された Mach の各々において使用されている用語を分類して示した．ここでの UNIX は，AT&T 社において 1978 年に開発された

表 4・1　代表的な OS における用語の定義

代表的ＯＳ	資源管理	CPU割当	並列処理方式
IBM/MVS	ジョブ	タスク	マルチタスキング
UNIX	プロセス	プロセス	マルチプロセス
CMU/Mach	タスク	スレッド	マルチスレッド

[*1] 仮想計算機：1 台のコンピュータシステムに複数の OS を稼働させることを目的とした制御プログラムがある．これも仮想計算機と呼んでいる．この場合は，1 台のコンピュータに論理的な CPU を複数台作成し，その各々に仮想的なコンピュータの 3 資源を OS に割り当てるという制御を行っている．

UNIX version 7 である．これをオリジナル UNIX と呼ぶことにする．現在の UNIX では，IEEE/POSIX 仕様においてリアルタイム機能を規定しているが，その中でスレッドが定義されている．

MVS のタスクや CMU/Mach のスレッドは一つのアドレス空間を共有している．つまり，独立したアドレス空間は，タスクやスレッド単位に与えられるのではない．一方，オリジナル UNIX ではプロセスは独立した資源割付けの単位であり，メモリを相互に共有することはない．UNIX で共有メモリの考えが導入されるのは後のことである．

CMU/Mach のスレッドならびに MVS のタスクは，**図 4･3** に示すように，一つのアドレス空間内に複数のスレッドが存在し，各スレッドは一つのアドレス空間内に存在している．つまり，各スレッドはプログラム，データ，スタックなどの実行環境を保有しているが，タスクに割り付けられたアドレス空間を共有しているのである．このようなマルチスレッド（Multi-thread）構成では，スレッド生成におけるカーネルのオーバヘッドが小さく済む点，ならびに，スレッド切替えがアドレス空間切替えにならないため，オーバヘッドが少ないという利点がある．この理由から，スレッドを軽量プロセス（Light Weighted Process）と呼ぶことがある．このように，一つのアドレス空間内に複数のプログラム実行体を作る方法を一般的にはマルチタスキング（Multi-tasking）という．

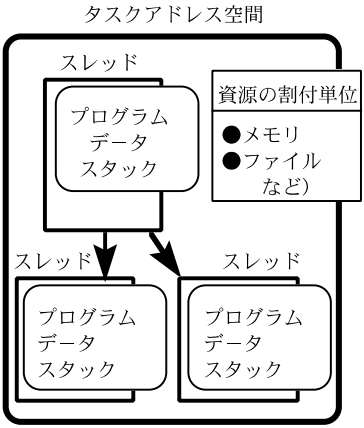

図 4・3　Mach におけるタスク，スレッドの関係

4-1-4 性能向上を目的とする並列処理

図 4･3 にはマルチスレッドのイメージを示した．同様のことを UNIX では**図 4･4** に示すようにシステムコール fork( ) で子プロセスを生成する．CMU/Mach や UNIX がこのような複数のスレッドやプロセスのファミリーを構成する理由は，一つの仕事を並列処理するためである．

並列処理の第 1 の目的は CPU を最大限使用して高速な処理を達成することにある．したがって，一つの仕事をするときに，並列処理できる部分は並列処理するソフトウェア構成に設計するのが基本である．人間社会に例えるならば，プロセスは労働者に相当しており，作業を分担して効率的に仕事を進めるやり方をコンピュータにとり込んだと考えれば，理解が容

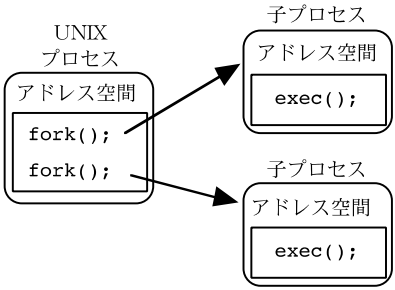

図 4・4　UNIX のプロセスファミリーによる並列処理動作

易である．

並列処理が効果的な例を**図 4･5** に示す．ここでは，ファイルから情報を読み込み，読み込んだ情報を基に大量の計算をする例である．このように二つの子プロセスを生成することにより入出力と計算を各々専用に実行させると CPU と入出力の資源を各プロセスが独立に使用でき，単一プロセスで実行する場合に比べて格段に処理時間が短縮されることになる．この例のように，入出力が処理上のボトルネックになっている場合には，入出力装置を休みなく利用することで最大限のコンピュータシステムの性能を引き出せる．

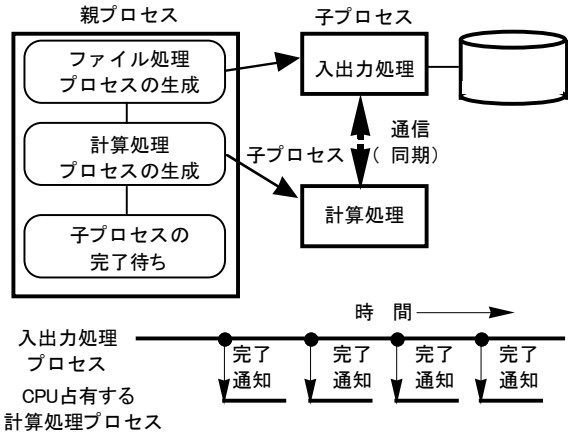

図 4・5　並列処理による高性能処理の例

以上から，ソフトウェア開発においては並列処理の可能性のある部分は極力並列処理できる機構に設計しておくべきである．並列化することで，次に述べる信頼性向上も果たせることになる．また，将来多重プロセッサが利用できる環境になったときに，ソフトウェアを再設計する必要もなく，直ちにハードウェアの投資に見合った性能向上が期待できることになる．

図 4･5 では複数のプロセスで一つの仕事を実行する例を示したが，このように複数のプロ

セスで処理を行うとプロセス間相互の通信が必要になる．つまり，このモデルでは，入出力プロセスはファイルを読み込んだ後に計算処理プロセスに対して新しいデータが読み終わったことを知らせる必要がある．このように，並列処理においてはプロセス間通信（IPC : Inter Process Communication）の機能が必要となる．IPC については 4 章に述べる．

4-1-5 信頼性向上を目的とする並列処理

マルチタスキングによる並列処理の第 2 の目的は信頼性の向上にある．単一プロセスでの実行では，プログラムの一つのバグが全処理の異常終了（Abnormal End）につながる可能性が高い．したがって，公共性の高い業務処理や企業の中枢的な情報システムの構築などではこのような状況は避けねばならない．

そこで，マルチタスキングによる機能分担の設計を採用すれば，バグが露呈したプロセスだけを終了させ，一部の機能を縮退（Degradation）してもほかのプロセスの処理を続行可能にすることができる．つまり，バグの影響を最小限に抑えることができる．このような設計思想をフェールソフト（Fail Soft）という．また，同一の仕事を複数のプロセスにより実行可能にしておくなら，その一部がバグでプロセス停止してもほかのプロセスによる処理の続行が可能となる．このように，冗長性をもたせたシステム設計をフォールトトレラント（Fault Tolerant）と呼ぶ．

4-1-6 多重プロセッシングを活かすマルチタスキング

マルチタスキングの利点はもう一つある．現代のコンピュータシステムでは複数の CPU を同時に稼働させることにより計算能力を向上することが可能となった．複数の CPU を使って処理能力を向上する計算機システム構成を一般的に多重プロセッシング（Multiprocessing）と呼ぶ．

多重プロセッシング構成を大きく分けると，主記憶を共有しているタイプと CPU 間を高速な通信装置で結合するタイプが存在する．前者を密結合多重プロセッシング（TCMP : Tightly Coupled Multiprocessing）あるいは UMA（Uniform Memory Access）と呼び，後者を疎結合多重プロセッシング（LCMP : Loosely Coupled Multiprocessing）あるいは NORMA（Non Remote Memory Access）と呼ぶ．**図 4･6** にその概略を示す．

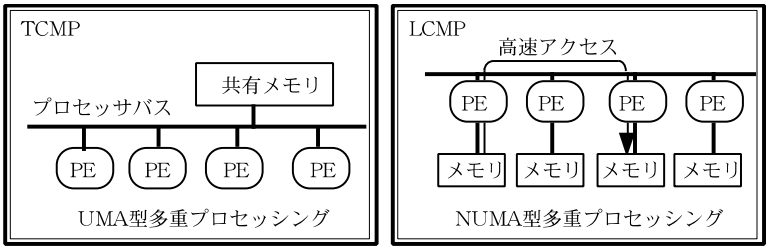

図 4・6　多重プロセッシングの CPU 構成

■7群 － 3編 － 4章

4-2 UNIXにおけるプロセス生成機能

（執筆者：吉澤康文）［2013年2月 受領］

4-2-1 プロセス生成

複数のプロセスによる並列処理ではプロセスを生成する機能が必要となる．UNIX では子プロセスの生成は fork( ) により行う．**図 4･7** に fork( ) システムコールの仕様を示す．返り値（Return Value）は生成された子プロセスに付けられたプロセス識別子（PID：Process Identification）であり，当該コンピュータ（ホストマシン）内でユニークな番号となる．PID はプロセスの識別に利用する．一般的に UNIX のシステムコールでは，返り値が負の場合はエラーである．

```
int fork()    /* 子プロセス生成          */
/*（返り値）= -1 ：生成失敗               */
/*             0 ：子プロセスの場合       */
/*           pid > 0：親プロセスの場合   */
```

図 4･7　UNIX における fork システムコールの仕様

図 4･8 には親プロセスが fork( ) を実行したときのイメージを示す. fork( ) では子プロセスは親プロセスとすべて同一のメモリ内容であり，子プロセスは fork( ) を実行した直後のステートメントから実行開始することになる．したがって，子プロセスは fork( ) の返り値がゼロであることを確認する必要がある．

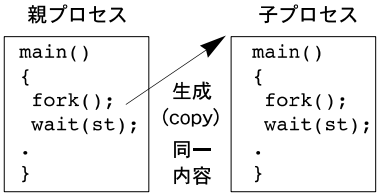

図 4･8　fork 直後の親と子のプロセスのメモリ内容

4-2-2 子プロセスの終了待ち

子プロセスの処理終了を親プロセスは知る必要がある．この機能を果たすのが wait( ) システムコールである．wait( ) の仕様は**図 4･9** のとおりである．

```
int wait(statusp)
int *statusp; /* 完了コード          */
/* 返値： process ID（正常終了時）  */
/*        -1（子プロセスの無いとき）*/
```

status	（返り値）	0x00

図 4･9　wait システムコールの仕様

子プロセスは処理が終了した際の完了情報を親プロセスに知らせることが可能である．これは最も簡単なプロセス間通信ということができる．完了コードは 1 バイトである．

親が wait( ) を実行していないときに子プロセスが exit( ) などで終了すると，ゾンビ（Zombie）プロセスとなって，親の待ち状態になるまでプロセステーブルだけが存在する．ゾンビプロセスは保有している資源をすべて解放されプロセスとしては終了している．これは肉体を失っているが魂だけは残っている様を表している．

4-2-3 プロセス識別子を知る

各プロセスに付けられたプロセス識別子を知るために，getpid（Get Process Identification）が用意されている．getpid( ) は自プロセスの識別子であり，親の識別子を知るには getppid( ) がある．これらのシステムコールを実行すると返り値として PID が得られる．**図 4･10** にその仕様を示す．

```
int getpid()  /* PIDを得る     */
int getppid() /* 親のPIDを得る */
```

図 4・10　PID を得るシステムコール

```
/****************************************/
/***  sample program using fork()      ***/
/****************************************/
#include <stdio.h>
 main()
{
 int  childpid;
 int  status;
 if ( (childpid = fork()) == -1) {
    perror("can't fork");
    exit(1);
 }
 else {
    if ( childpid == 0 ) {/* child process*/
                          /* will run here*/
       printf("child: child pid = %d, parent
          pid = %d\n", getpid(), getppid());
       exit(0);
    }
    else {     /* parent process will run */
       printf("parent: child pid = %d,
          parent pid =  %d\n", childpid,
                                  getpid());
       wait(&status);
    }
 }
}
```

図 4・11　fork システムコールの使用例

UNIX の PID にはあらかじめ決められたプロセスがある．プロセスのメモリ空間を 2 次記憶に退避・回復するスワッパ（Swapper）は 0，全プロセスの親であるイニット（init）プロ

セスは 1，そして仮想記憶を実現するために必要なページデーモン（Page Daemon）と呼ばれるプロセスは 2 である．

図 4･11 に，上記に説明したシステムコールを使った子プロセスの生成と，そのプロセス識別子を表示するプログラムの例を示す．親プロセスは fork( ) を実行し，fork( ) の返り値が負とゼロでないことを確認し，自プロセスの識別子を得るために getpid( ) を実行する．そして，PID 値を出力し子プロセスの完了を wait( ) で待つ．一方，子プロセスは自分が子プロセスであることを fork( ) の返り値ゼロであることから判別し，自プロセスと親プロセスの PID を得てそれらを出力し完了する．

4-2-4 自プロセスを終了させる

プロセスが全処理を完了したときに実行するシステムコールが exit( ) であり，その仕様は **図 4･12** に示すとおりである．カーネルはプロセスに割り付けられている資源を解放処理する．例えば，メモリ領域，使用されているファイルの使用完了処理などである．引数としてプロセス終了コードを指定できる．このコードを親プロセスは wait( ) によって知ることができる．

```
void exit(status) /* プロセス終了 */
int  status;      /* 終了コード   */
```

図 4・12　exit システムコールの仕様

4-2-5 別のプログラムを実行する

子プロセスに別のプログラムを実行させたい場合がある．そのようなときには exec( ) システムコールを実行する．**図 4･13** に示すように子プロセスが exec( ) を実行することによりプログラムを主記憶に読み込み（Loading）実行する．

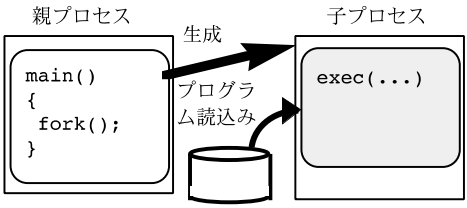

図 4・13　exec システムコールによる別プログラムの実行

4-2-6　init とプロセス生成過程

UNIX ではブート過程において init プロセスが生成される．init は**図 4･14** に示すようにプロセス PID が 1 である．init は/etc/rc に指定されたシェルスクリプトを実行する．ここには，図に示すようなスワッパ，プリンタ，ページングなどのデーモンプロセスを起動することが書かれている．

その後，/etc/ttys に指定された端末に対応するだけのプロセスを fork( ) により生成する．実行するプログラムは/etc/init である．この後の処理を次に述べる．

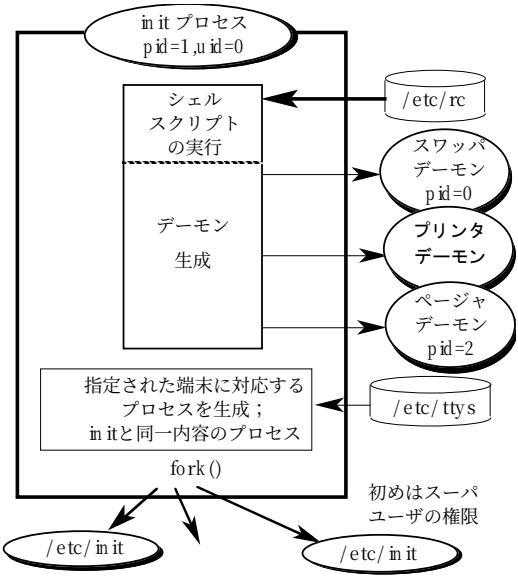

図 4・14　init プロセスのブート時動作

4-2-7 端末のログイン

図 4・15 のように fork( ) で生成された PID は 1 ではない．/etc/init のプログラムの重要な仕事は，正しいログインユーザを認証することである．そこで，端末に対してログインを促す

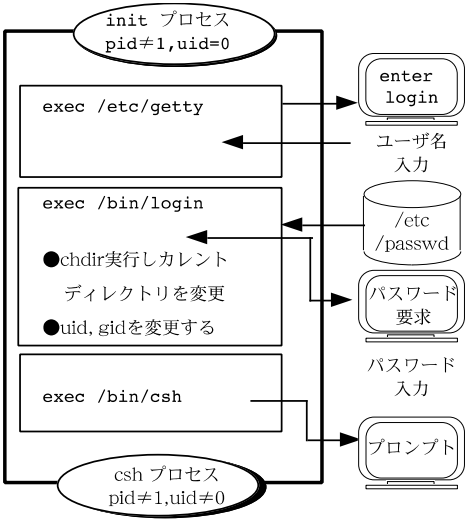

図 4・15　/etc/init の実行過程

プロンプトメッセージを表示する．ユーザはこのメッセージにより自分の登録してあるユーザ名を入力すると，/etc/init は/etc/passwd ファイルから該当のログイン名を探し，パスワード入力を端末に要求する．

パスワードの認証が正しく行われると，該当ユーザのファイルポジションをカレントワーキングディレクトリ（Current Working Directory：ファイルシステムの章を参照）とし，指定されたシェルを起動する．

4-2-8 デーモンプロセス

図 4･14 に示したように init はスワッパ，プリンタ，ページャなどのプロセスを生成する．これらは，独立したプロセスとして動作するが，一般のプロセスとは異なり，デーモン（Daemon）と呼ばれている．本来はカーネルの機能を担っているが，自律的に動作した方が効率的であるために独立したプロセスになっている．まさに，カーネルの守護神的な位置付けになっており，プロセスとして独立して任務を分担する．

デーモンは，fork( ) で生成されたときにキーボードやディスプレイを保有していない．つまり，ユーザとのインターフェースをもっていないこと，ならびに，カーネル同様に高い特権モードで実行されるのが一般的である．以下，代表的なデーモンについて説明する．

(1) **スワッパ**：多くの UNIX では仮想記憶を実現しているが，仮想記憶の容量が実記憶の容量に比べて大きくなるとページングが多発し，性能低下に陥る．この状況を回避するために，プロセスの多重度（マルチプログラミングの多重度）を低下させるために，プロセスに割り付けられたメモリ領域を 2 次記憶に退避する．この操作をスワップアウトという．逆に，実記憶に空きが多くなり，スワップアウトされているプロセスがあるならば，それを実記憶に読み込む必要がある．これをスワップインという．このような操作を行うのがスワッパである．

(2) **ページャ**：仮想記憶において，利用度の低いメモリ領域を 2 次記憶に書き出したり（これをページアウトという），参照されたが主記憶になく 2 次記憶に存在するようなとき読み込む（これをページインという）ことを行う．ページャはこれらの機能を分担している．

(3) **プリンタデーモン**：一般的に，プリンタは低速な装置である．そこで，プリントアウトするときは，磁気ディスク上にスプール（Spool）ファイルを用意し，プリンタ出力情報をファイルにいったん格納しておく．その後に，ファイルからプリンタへの出力を専用に行うプロセスがプリンタデーモンである．**図 4･16** にはプリンタデーモンとスプール処理の関連を示す．

こうすることにより，プロセスがプリンタへの出力時に生じるプリンタ占有の問題を避けられる．つまり，プロセスが 1 行出力した後に長い計算を行うと，次の出力までほかのプロセスはプリンタを使用することができない．そこで，プリンタデーモンはプリンタへの出力要求が生じると，仮想的プリンタであるスプールファイル内にプリンタイメージ情報を出力しておく．このようにすると，各プロセスは出力時間が大幅に短縮されるので，計算が高速になるばかりでなく，プリンタを占有することを避けることができる．

プリンタデーモンは，プロセスが完了したときに各プロセスのスプールファイルを閉じ（close），プリンタに出力を行う．このようにプリンタデーモンは，一般のプロセスに対し

てスプールファイルによりプリンタ機能をシミュレートしておき，各プロセスのプリンタ占有問題を解決する．そして，プリンタを占有管理し休ませることなくその能力を 100 %活用することが可能となる．

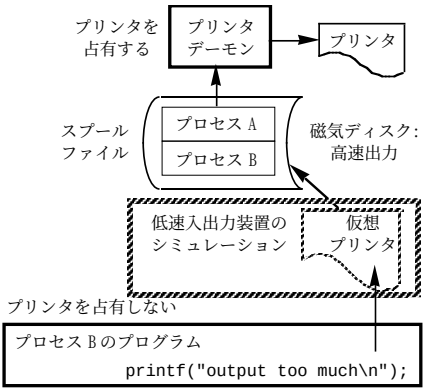

図 4・16　プリンタデーモンによるスプール処理の概要

■7群 － 3編 － 4章

4-3 プロセススケジューリング

（執筆者：吉澤康文）［2013年2月 受領］

4-3-1 プロセスの状態と遷移

基本的な三つの状態をプロセスはもつ．それらは，以下の状態である．

・実行状態（Running State）
・実行可能状態（Ready State）
・待ち状態（Waiting State）

(1) **実行状態**

プロセスはCPUを割り付けられてプログラムを実行している状態である．コンピュータシステム内に存在するCPUの数だけこの状態のプロセスが最大限存在する．つまり，2台のCPU構成のコンピュータシステムならば，この状態になり得るプロセス数は最大2である．

プロセスが実行状態になるのは，実行可能状態にあるプロセスの中から優先順位に従って選ばれたときである．このようにCPUを割り付ける操作をディスパッチ（Dispatch）あるいはスケジュール（Schedule）と呼び，OS重要な役割である．

(2) **実行可能状態**

CPUの割付けを待っている状態であり，各プロセスは相互にCPUの割付けを競っているとみなされる．この状態のプロセスは，一般的に実行優先順位が付けられている．fork( )で生成されると，最初はこの状態になる．

(3) **待ち状態**

プログラムが何らかのイベント（Event）を待つことになり，命令を実行できない状態をいう．UNIXではブロック（Blocked）状態ということがある．代表的な例は，ファイルの読込み時で，その完了まで待たされる状態，あるいは，端末からユーザの入力を待つ状態などである．したがって，待ち状態の解除は，ほかのプロセスあるいはカーネルからのイベント通知であり，一般的にプロセス間通信で実現される．次章で説明するが，排他制御によるイベントの通知もこの中に入る．

プロセスは上記の3種類のいずれかの状態にあるが，これらの状態を遷移して最後は仕事を完了し消滅する．この状態遷移を示したのが**図4･17**である．プロセスは生成されると実行可能状態に入る．そして，CPUの割当て順番を待ち，実行状態に入る．CPUだけを利用する

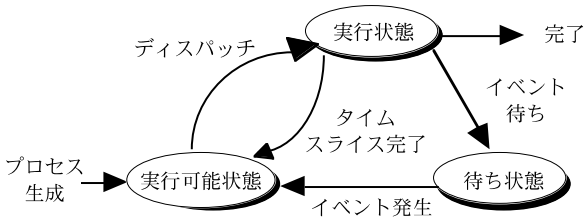

図4･17　プロセス状態の遷移

プログラムの場合はこれで完了する場合もある．しかし，一般的には何らかのイベントを待つ操作を行うので，待ち状態に入る．

イベントが発生すると当該プロセスは他プロセスやカーネルにより待ち状態が解除される．これは，プロセスの状態が待ち状態から実行可能状態に移ることを意味する．このとき，実行可能状態のプロセスは実行優先順位に基づき整列することになる．整列の方法により実行優先度が決定されるので，これをプロセススケジューリング（Process Scheduling）と呼び，各種のアルゴリズムが考案されている．

一般的に，プロセスは実行可能状態からスタートして実行状態，待ち状態，そして再び実行可能状態を繰り返して完了する．しかし，この過程において，図 4･17 に示してもあるが，実行状態から再び実行可能状態に遷移する場合がある．これは，タイムスライス（Time Slice）完了という事象である．

タイムスライスとは時間を一定のサイズに分割した単位のことであり，タイムクオンタム（Time Quantum）と呼ぶこともある．このように時間を小さな単位に区切り，各プロセスに与える CPU 時間を制限すると，実行可能状態にあるプロセスに CPU を割り当てる機会が増えることになる．

タイムスライスの考え方はタイムシェアリングシステム（TSS）の出現が契機になった．つまり，一台の CPU のもとに，複数の端末から対話形式でコンピュータを同時利用させる環境を考えると，応答性能が重要になる．初期の TSS の環境では，プログラム開発を端末から行うことが重要な使命であり，テキスト編集（Text Editor）を多くの端末ユーザは使っていた．テキスト編集は，通常少ない CPU の消費で処理ができる．したがって，**図 4･18** に示すタイムスライス内で処理が完了し，次のユーザの端末入力を待つことになる．このため，CPU を多用するプロセスを積極的に排除することが必要になり，CPU の割付け時間の上限値がタイムスライスとなったのである．

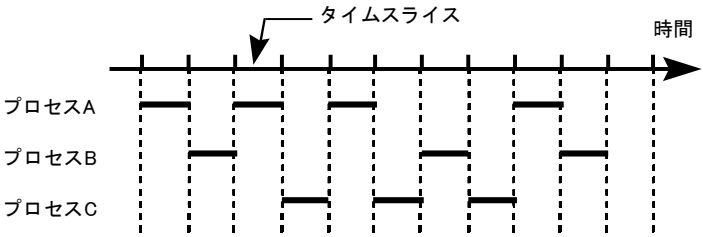

図 4・18　TSS におけるタイムスライス制御の考え方

4-3-2 スケジューリングの基本的な考え

ある目標達成のために限られた資源の利用順位と使用量を決めることをスケジューリングと呼ぶ．コンピュータにおける性能目標は，単位時間当たりのジョブ処理件数であるスループットならびに対話処理における応答時間（Response Time）の保障にある．ここで応答時間とは，端末からの処理要求に対して最初のメッセージが端末に出力されるまでの時間である．

上記のスループットと応答時間という性能目標に対して有限のコンピュータ資源の割当て方針を立てるのはコンピュータ運用者の役割である．そこで，OS に対してその方針を指示

できるようになっていることが望ましい．この種の優れた機能をもつ OS の代表に IBM 社の MVS の SRM（System Resource Manager）があるが，ここでは，CPU 資源に限定したプロセススケジューリングについてのみ説明する．

プロセススケジュールとは，実行可能状態にあるプロセスに対して CPU 割付け順位を決定することである．各プロセスはプロセステーブル（Process Table）により管理されている（4-4-1 項で説明）ので，プロセススケジューリングは，プロセステーブルのチェイン（Chain）を各アルゴリズムに基づき作成する．**図 4･19** にはカーネル制御テーブルとプロセステーブルへのチェインの一部を示した．

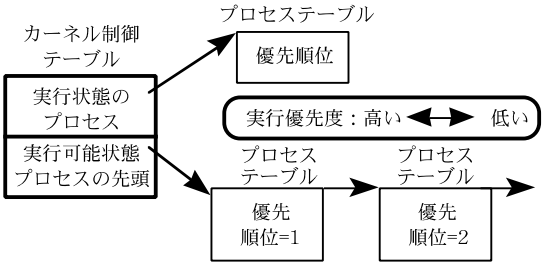

図 4・19　実行可能状態のプロセステーブル

プロセススケジューリングの使命は，スループットの向上と応答時間の保障にある．したがって，マルチタスキング環境において，ある特定のプロセスが資源を独占的に使用するような状況が生じると，この目標を達成することが困難になる可能性が高い．**図 4･20** に示すような単一 CPU システム（Single Processor System）では，CPU 資源はコンピュータシステム内で重要な位置を占めており，CPU をあるプロセスが独占してしまうとほかの入出力装置が働けなくなるのである．

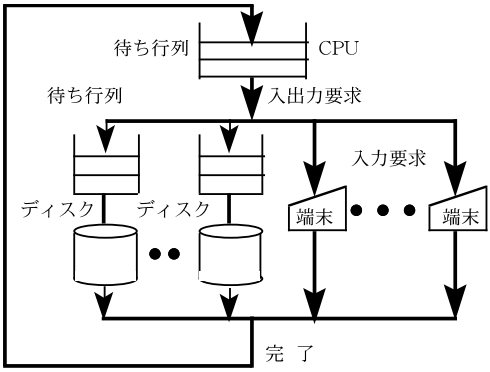

図 4・20　プロセスの処理過程における資源要求の循環

このため，CPU のスケジューリングの基本的な考え方は，CPU を独占するプロセスを排除する機能が必須であること，ならびに積極的にプロセスが入出力装置を使用できるように

CPU 資源を割付ける努力をすることである．この結果として，スループットと応答性能を共に満足するスケジューリングが可能となる．以下，CPU スケジューリングの説明を行うが，各々に対する評価は上記の観点からなされる．

これらのスケジューリングを大別すると，プロセスに CPU を割り付けた後に，そのプロセスが完了するか，あるいは CPU の使用を自ら放棄しない限り CPU を使用させる方法をノンプリエンプティブスケジューリング（Non-preemptive Scheduling）と呼ぶ．逆に，OS の方針により実行可能なプロセスであってもプロセスの意に反して CPU の使用を中断し他のプロセスに切換えるスケジューリングがある．これをプリエンティブスケジューリングという．プリエンプティブは横取りスケジューリングと呼ぶこともある．FIFO や SPTF はノンプリエンプティブスケジューリングであり，ラウンドロビンや優先度スケジューリングはプリエンプティブスケジューリングである．また，現代の OS では，一つのスケジュール方式だけを採用していることは少なく，ここに説明された方式を複合して採用していることが多い．

4-3-3 FIFO

初期のバッチ処理においては，ジョブがコンピュータに投入されると，その順序にジョブが実行された．これが，先入れ先出し，つまり FIFO（First In First Out）あるいは FCFS（First Come First Serve）である．この方法は実現が簡単であるが CPU を独占するプロセスが一つでもあればほかのプロセスがすべて CPU 待ち状態に陥り，性能の低下となる．この方式は典型的なノンプリエンプティブスケジューリングといえる．

4-3-4 優先度スケジューリング

プロセスに処理優先順位を与える方法である．この方法も FIFO 同様の欠点を備えている．しかし，優先的に処理を行いたいプロセスのためには性能がある程度保障されるという利点がある．この方式では，入出力完了割込みが生じたとき，優先度の高いプロセスの入出力が完了したのであれば，その時点でプロセススイッチが生じるので，プリエンプティブスケジューリングとなる．

4-3-5 ラウンドロビン

タイムシェアリングシステムの出現により考案されたスケジューリング方式であり，タイムスライスを導入している．この方式は，図 4･18 に示したように，実時間を短いタイムスライスに区切り，CPU 割付け時間の上限値とする方法である．したがって，タイムスライスを使い切ったプロセスは**図 4･21** に示すように，処理が中断されて実行可能待ち行列にプロセスは並ぶことになる．

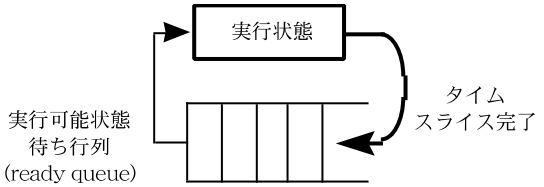

図 4・21　ラウンドロビン法によるスケジュール

ラウンドロビン法（Round Robin）は CPU を独占使用する FIFO や優先度スケジューリング法の欠点を解決している．このため，CPU を短く使い，入出力を多用するようなプロセスは処理が進み，スループットや応答性能の向上が図れる利点がある．ラウンドロビン法は典型的なプリエンプティブスケジューリングである．

欠点としては，最適なタイムスライス値の設定にある．タイムスライスが短いとタイマ割込みやスケジューリングのオーバヘッドが大きくなる．逆に，長いタイムスライスではラウンドロビンの効果を失うことになる．

4-3-6 SPTF

コンピュータシステムでは十分なメモリがあればプロセスは CPU と入出力装置を使用することで仕事を進められる．CPU は数の少ない装置であるため CPU がシステムボトルネック（System Bottleneck）になりやすい．そこで，複数のプロセスがシステム内に存在するとき，CPU を多用するプロセスを CPU バウンドプロセス（CPU Bound Process），入出力を多用するプロセスを I/O バウンドプロセスと分けるならば，CPU バウンドプロセスの処理優先度は低くした方がよいと考えられる．

上記の考えからすると，ラウンドロビン法はその考え方に基づいた一つのスケジューリング方式といえる．図 4･20 に示したプロセスの資源要求の循環が進めば，スループット，応答時間の向上が図れることになる．そこで，ラウンドロビン法を発展させる方式を考える．ラウンドロビン法は図 4･21 に示したように，タイムスライス完了時にプロセスを実行可能待ち行列の最後に並べるが，SPTF（Shortest Processing Time First）ではその際に優先順位を計算する．つまり，CPU バウンドプロセスほど優先度を低くするのである．逆の言い方をすれば，I/O バウンドプロセスほど処理優先度が高くなり，実行可能待ち行列の前の方につながれることになる．

このような方法を採用するためには，OS は各プロセスが連続して CPU を使用する時間間隔を記録する必要がある．この時間間隔が長いほど CPU バウンドプロセスと判断される．各プロセス動作のモニタリング（Monitoring）情報は，プロセスが生成された時点からの蓄積ではなく，過去のある一定期間の動作を測定するのが一般的である．つまり，最近の資源使用動向を観察することで，近い将来の動作を予測しようという考えに基づいているといえる．

4-3-7 デッドラインスケジューリング

アプリケーションプログラムによっては，限られた時間内に処理を完了しなければならないものがある．時間の制約が厳しいリアルタイム処理（Real Time Processing）などが代表的応用例であり，動画像や音声処理を行う場合なども含まれる．これらの処理は実時間に正しく処理されねばならない．

これらの処理は，各プロセスの処理がある時間帯内に完了する必要があり，目標となる時間帯の中でスケジュールされねばならない．つまり，処理の完了すべき目標時間が設定されるのである．このようなプロセスに対しては，目標時間（Deadline）が近づくに従って CPU 割当の優先順位を高くすることが望ましい．デッドラインスケジューリングは目標の時間帯内に処理を完了することを目的に処理の優先順位を動的に決定する方式のことである．

■7群 － 3編 － 4章

4-4 その他

（執筆者：吉澤康文）［2013年2月 受領］

4-4-1 プロセス管理テーブル

プロセスは「プログラム実行の抽象化された実行体」として OS に管理されている．複数のプログラムを生成しマルチタスキング機能を提供することでコンピュータの性能，信頼性向上を実現している．ここでは，図 4･1 にて説明したプログラム実行体としてのプロセス管理テーブル（PCT と略す）について説明する．

プロセスを管理するためにはいくつかの管理項目がある．**図 4･22** には主たる項目を示した．PCT をそのシステムで生成可能な数だけシステムブートストラップの時点でカーネル領域に作成しておく．これらの PCT は最初は未使用状態であり，未使用状態の PCT を管理するためにリスト（チェイン）にしておく必要がある．これらの情報が先頭の 2 項目である．PCT のチェインはプロセスが実行可能状態やイベント待ち状態のときにも作成される．

pctの状態：未使用／使用
pct のチェイン（ポインタ）：空きpct, 実行可能状態pct, イベント待ちpct, など
プロセスID
プロセスの保有するレジスタ： 中断時点で保存
プログラム状態語(Program Status Word) ：再実行時のアドレス、プロセッサ使用状態
メモリ割付情報：テキスト，データ，スタック などメモリオブジェクト単位の管理
プロセスの状態：イベント待ちの要因
シグナル情報：イベント待ちマスク
時間に関する情報：自プロセス、 子プロセスの使用した計算機時間
プロセス間通信バッファ／メッセージ情報
ファイル記述子テーブルポインタ

図 4･22　プロセス管理テーブル（pct）の主たる構成要素

プロセスが生成されると，空きの PCT が取り出され，プロセスにユニークな番号としてプロセス ID が付与され，PCT に入る．プロセスが仮想計算機と呼ばれる所以は，CPU の状態を保持している点にある．プロセスが待ち状態や実行可能状態で処理を中断している場合には，中断時点のレジスタ類，CPU の状態を表すプログラム状態語（PSW：Program Status Word）などが必要となる．PSW の中には，プロセスを再開したとき，次に読出す命令（Instruction Fetch）のアドレスや CPU の各種動作モードが含まれている．

その他，メモリ，ファイルなどの資源割付け管理の情報を格納しておく必要がある．プロセスあるいはその子プロセス類が使用した CPU に関する統計量も必要となる．また，シグナ

ルのようなソフトウェア割込みは，各プロセスごとに設定された機能であるため，これらもプロセス管理テーブルに格納しておく必要がある．

4-4-2 タイマ機能

時間に関するカーネルのサービスには以下のようなものがある．

(1) **使用した CPU 時間**：**stimes( )**

性能を測定し料金計算をするために利用される場合が多い．そのために UNIX では，システムコール stimes( ) が用意されている．仕様は**図 4･23** に示すとおりであり，そこでは四つの情報を含む構造体へのポインタを引数とする．これらには，自プロセスの使用した CPU 時間だけでなく，カーネルの使用した時間も分離されて得ることが可能である．また，自分の生成した子プロセスの使用した CPU 時間も同時に得られる．

```
#include  <sys/types.h>
#include  <sys/times.h>

struct tms {
    time_t  tms.utime;  /* ユーザ使用CPU時間    */
    time_t  tms_stime;  /* システム使用CPU時間    */
    time_t  tms_cutime; /* 子プロセスのユーザ使用CPU時間 */
    time_t  tms_cstime; /* 子プロセスのシステム使用CPU時間 */
}
   long  times(struct tms *tbuf)
/* 返り値：エラーのとき -1 */
```

図 4・23　CPU 時間取得システムコール times の仕様

times( ) で返される値の単位はクロックティック（Clock Tick）値であり，1/100 秒であることが多い．したがって，秒で値を得るには CLK_TCK の値で除算する必要がある．

(2) **時刻を設定，取得する**：**stime( ), time( )**

システムの時刻を設定するシステムコールが stime( ) である．システムワイドな資源に値を入れるため，スーパユーザにだけ許されるシステムコールである．stime( ), time( ) の仕様を**図 4･24** に示す．

```
long  time(long *timep)
/* 返り値：エラーのとき -1 */
```

図 4・24　経過時間を取得するシステムコール time の仕様

UNIX では，1970 年 1 月 1 日午前 0 時グリニッチ標準時（GMT：Greenwich Mean Time）からの秒経過時間で内部の時（カレンダ時間）を刻んでいる．stime( ) ではパラメータをカレンダ時間に変換しておく必要がある．逆に，time( ) では，GMT をライブラリ関数 ctime( ) を使ってローカルタイム（Local Time）に変換した方が理解しやすい．

(3) **時間の経過でソフトウェア割込みを発生させる**：**alarm( )**

時間を指定して事象をとらえる場合に使用する．例えば，端末からの入力打切り時間の設定などに使用する．本機能は 7 章のプロセス間通信において詳しく説明する．

■7 群 － 3 編 － 4 章

4-5 演習問題

（執筆者：吉澤康文）［2013 年 2 月 受領］

(1) プロセス管理の機能を列挙せよ．
(2) スループット，応答時間を説明せよ．
(3) スプールとは何か説明し，プリンタへの出力にスプールを利用している利点は何か述べよ．
(4) 複数のプロセスで処理を実行する利点，欠点を述べよ．
(5) プロセスの三つの状態とそれらの状態遷移の条件を示す図を示してみよ．
(6) プロセススケジューリングの方法を 4 種類あげ，その特徴を述べよ．
(7) 横取りのあるスケジューリングの利点を述べよ．
(8) タイムシェアリングシステムの目的を述べどのようなスケジューリングを行っているかを説明せよ．
(9) UNIX のシステムコール fork( ) の機能を述べ，欠点が何かを説明せよ．
(10) UNIX のシステムコールに fork( )，wait( )，exit( ) があるが，それらの関係を説明せよ．
(11) 子プロセスを生成し，子プロセスが作成されたことを以下のプログラムを作成することで確認せよ．
　・親，子プロセスのプロセス ID を表示する．
　・子プロセスの終了を親プロセスは知り，子のリターンコードを親プロセスが表示する．
(12) プロセス制御テーブルに含ませるべき情報にはどのようなものがあるか列挙せよ．
(13) 現実のコンピュータシステムの応用において，並列処理を行っている例を示し，その理由，利点などを述べよ．