8 プロセス管理

8–1 プロセス管理で何が行われるか

多重プログラミング ···(p.39からの引用)

- 複数のプロセスを並行して走らせる。
- 入出力装置の制御は別の装置 (入出力コントローラという) に任せる。
- 実行中のプロセスの次の動作が入出力なら、
  ① その入出力動作を行うように入出力コントローラに指令を出す、そして、
  ② そのプロセスは一時的に待ち状態にして、別の (実行可能状態になっている) プロセスを (並行して) 走らせる。
- 入出力コントローラは入出力動作が完了した時点で割込み信号を (CPUに対して) 発生する。
- 入出力終了の割込み信号を受けたら、その入出力動作を行ったプロセスを待ち状態から実行可能状態に変更する。

おまけ：

外から見ると複数のプログラムが同時に並列実行されている様にも見えるので、多重プログラミングを疑似的並列 (pseudo parallel) 処理と言うことがある。

多重プログラミング ・・・(p.39からの引用)

- 複数のプロセスを並行して走らせる。
- 入出力装置の制御は別の装置 (入出力コントローラという) に任せる。
- 実行中のプロセスの次の動作が入出力なら、
  ① その入出力動作を行うように入出力コントローラに指令を出す、そして、
  ② そのプロセスは一時的に待ち状態にして、別の（実行可能状態になっている）プロセスを（並行して）走らせる。
- 入出力コントローラは入出力動作が完了した時点で割り込み信号を (CPU に対して）発生する。
- 入出力終了の割込み信号を受けたら、その入出力動作を行ったプロセスを待ち状態から実行可能状態に変更する。

おまけ：

外から見ると複数のプログラムが同時に並列実行されている様にも見えるので、多重プログラミングを疑似的並列処理と言うことがある。

多重プログラミングにおいては 、
入出力動作に入ったプログラム実行は一時的に休止状態にして別のプログラムの実行を進める。

⇨ 複数のプログラム実行を並行して進めるための仕掛けが必要になる。

複数のプログラムを並行して進めるための仕掛け：

- この仕掛けの中で扱うプログラム実行の単位 (i.e. プロセス実行の処理状況を表す) をプロセスまたはタスクと呼び、
  - ◇ プロセスの名前,
  - ◇ プロセスの識別子 (**PID**),
  - ◇ プロセスの動作状態,
  - ◇ プロセスの優先度,
  - ◇ プロセス中断時点での各種レジスタの内容,
  - ◇ プロセスが走らせているプログラムの情報,
  - ◇ プロセスが利用している実メモリ空間や仮想メモリ空間に関する情報,

  などから成る表 (プロセス管理表と言う) で表す。

- 各々の時点でどのプロセスを実際に実行するかを決め、その決定に従って走行プロセスの切替えを行うモジュール (ディスパッチャまたは (CPU) スケジューラと言う) を OS の中に用意する。

- 走行プロセスの切替えは割込みを契機に行われることが多い。

割込み事象が発生すると...：・・・(p.37からの抜粋)

実際に割込み事象が発生すると、次のような手順で処理が切替えられていく。

割込み要求をすぐに受け入れるかどうかの判断：

(1) 生起した割込み事象の割込みレベルと現在の走行レベルを比較し、

(1.1) もし現在の走行レベルの方が (優先度が) 高ければ、割込み要求は待たせて現在の実行を継続する。

(1.2) もし生起した割込みのレベルの方が高ければ、次のステップ **(2)** に移る。

前処理：

(2) レジスタ群の内容を然るべき領域に退避する。

(3) 利用するスタックを切替える。

個別処理：

(4) 走行モードをスーパバイザモードに、走行レベルを生起した割込みのレベルに変更する。

(5) 実行要求のあった割込み処理を起動する。例えば、

(ゼロ除算の場合は)　ゼロ除算に対処するルーチンが用意されていればそのルーチンを実行する。 用意されてなければゼロ除算を起こしたプロセスを強制終了させ、ステップ **(6')** に移る。

(入出力処理終了の報告の場合は)　その装置に対する入出力要求が他にあれば、次の要求に応えるために装置に次の制御メッセージを送る。また、終了報告のあった入出力を行ったプロセスを待ち状態から実行可能状態に変更する。　そして、次にステップ **(6)** または ステップ **(6')** に移る。

ステップ **(6)** と **(6')** のどちらに移るか：
プロセスに **CPU** を割り付けた後そのプロセスが完了するか自ら **CPU** を放棄するまで **CPU** を使用させる、いわゆる **non-preemptive** スケジューリングを採用している場合は、次にステップ **(6)** に移る。

また、現在実行しているプロセスの処理を中断し別のプロセスの処理を開始することがあるスケジュール法をプリエンプティブ・スケジューリングと呼ぶ。この方式を採用している場合は、次にステップ **(6')** に移る。

(カーネルコール呼出しの場合は)　指定された機能を実行する。 入出力要求などの場合には、現在実行しているプロセスを実行中からイベント待ち状態に変更し、ステップ **(6')** に移る。

(割り当てられた **CPU** 時間を使い切った場合は)　現在実行しているプロセスを実行中から実行可能状態に変更し、ステップ **(6')** に移る。

後処理：

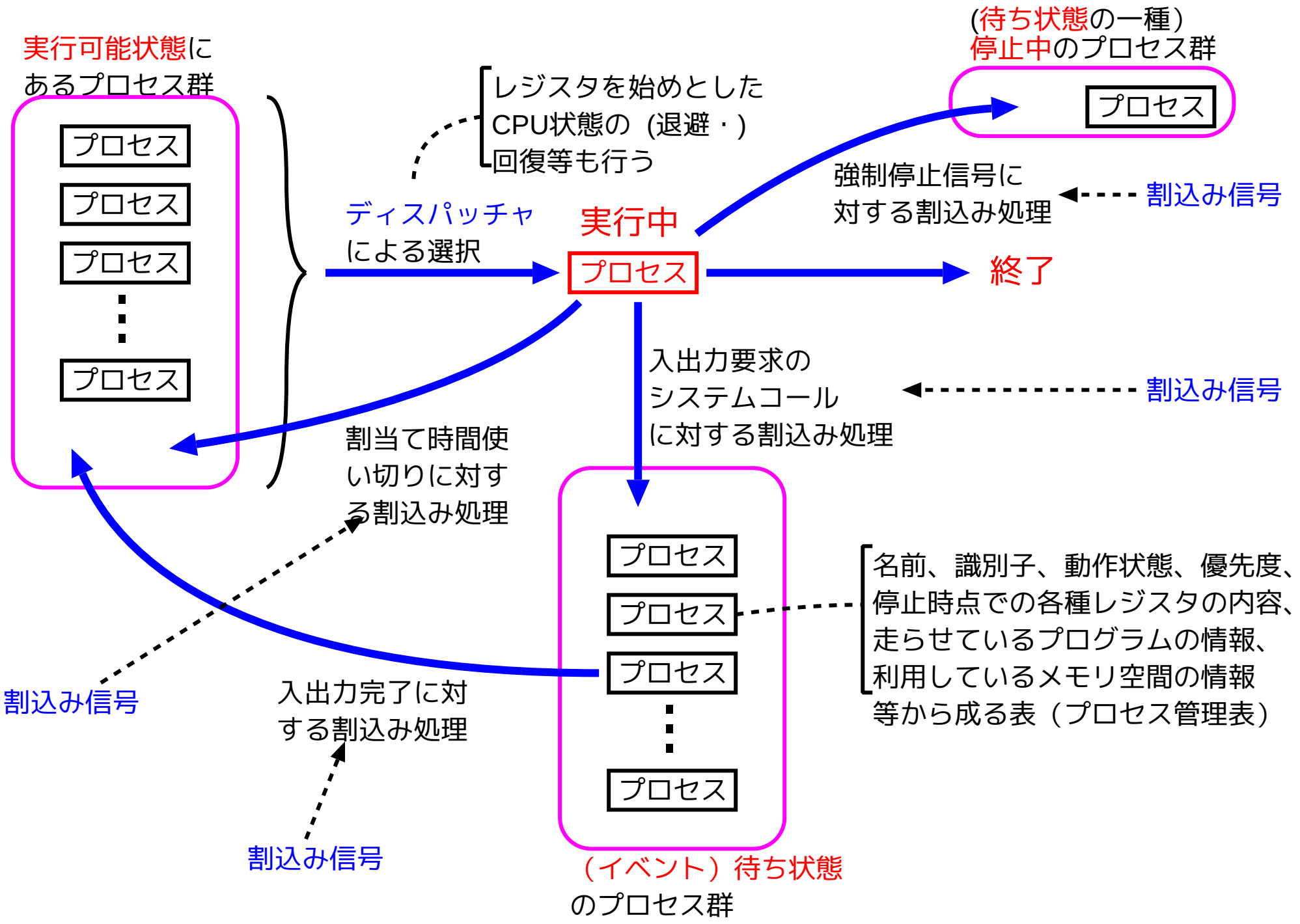
(6) レジスタをステップ **(2)** の前の状態に回復する。

(7) 利用するスタックを切替える。

(8) 走行モード、走行レベルの回復。

(9) 割込み事象発生時に走行していたプロセスを継続して実行する。

(6') 「ディスパッチャ」を呼び出して、その時点で最も優先度の高いプロセスを走らせる。


実行可能状態にあるプロセス群
プロセス
プロセス
プロセス
プロセス
レジスタを始めとしたCPU状態の（退避・）回復等も行う
ディスパッチャによる選択
実行中
プロセス
(待ち状態の一種)
停止中のプロセス群
プロセス
強制停止信号に対する割込み処理
割込み信号
終了
入出力要求のシステムコールに対する割込み処理
割込み信号
割当て時間使い切りに対する割込み処理
プロセス
プロセス
プロセス
プロセス
名前、識別子、動作状態、優先度、停止時点での各種レジスタの内容、走らせているプログラムの情報、利用しているメモリ空間の情報等から成る表（プロセス管理表）
割込み信号
入出力完了に対する割込み処理
割込み信号
（イベント）待ち状態のプロセス群

8–2 プロセス,タスク,スレッド,多重タスキング–

プロセスに関する用語/考え方はOSによっても少し異なる。

⇨ **IBM社MVS, UNIX version7(AT&T), CMU Mach** について比較すると次の通り。

	MVS	UNIX version7	Mach
開発元	IBM	AT&T	CMU
開発年	1974	1978	1986
資源(e.g. アドレス空間)割当ての単位	ジョブ	プロセス	タスク
CPU割当ての単位	タスク	プロセス	スレッド
並列処理方式	多重タスキング	多重プロセス	多重スレッド

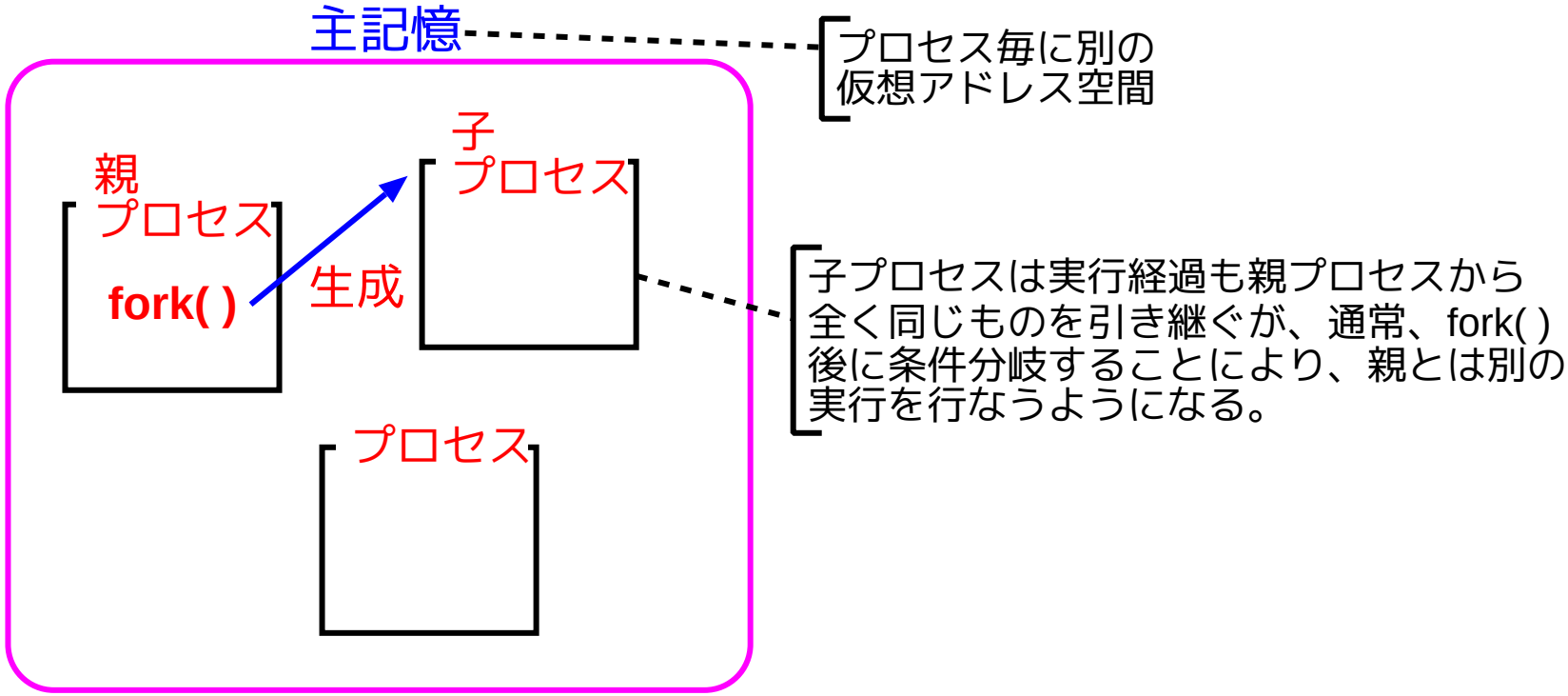
補足：

- 「多重プロセシング」と言ってしまうと
  多重プロセッサにより行われる計算処理
  の意味になってしまうので、ここでは「多重プロセス」と言っている。

- 岩波情報科学辞典によれば、プロセス(タスク)でサブプロセス(サブタスク)を生成できる時、それらのプロセスを並行に実行することを多重タスキング(**multitasking**)と呼んでいる。

  ⇨ この用語の使い方だと、並列処理方式はどの場合も多重タスキングということになる。

- **UNIX version7 においては**、システムコール fork( ) を用いて1つのプロセス内から別のプロセスを生成して、これらを並行に実行できる様になっている。



しかし、1つのアドレス空間内には1つ のプロセスしか存在しない。

⇨ プロセスはメモリを共有することなく
各々が全く独立な仕事をする。

⇨ アドレス空間もCPUもプロセスに対して割り当てられる。

補足：

その後共有メモリの考えが導入されている。

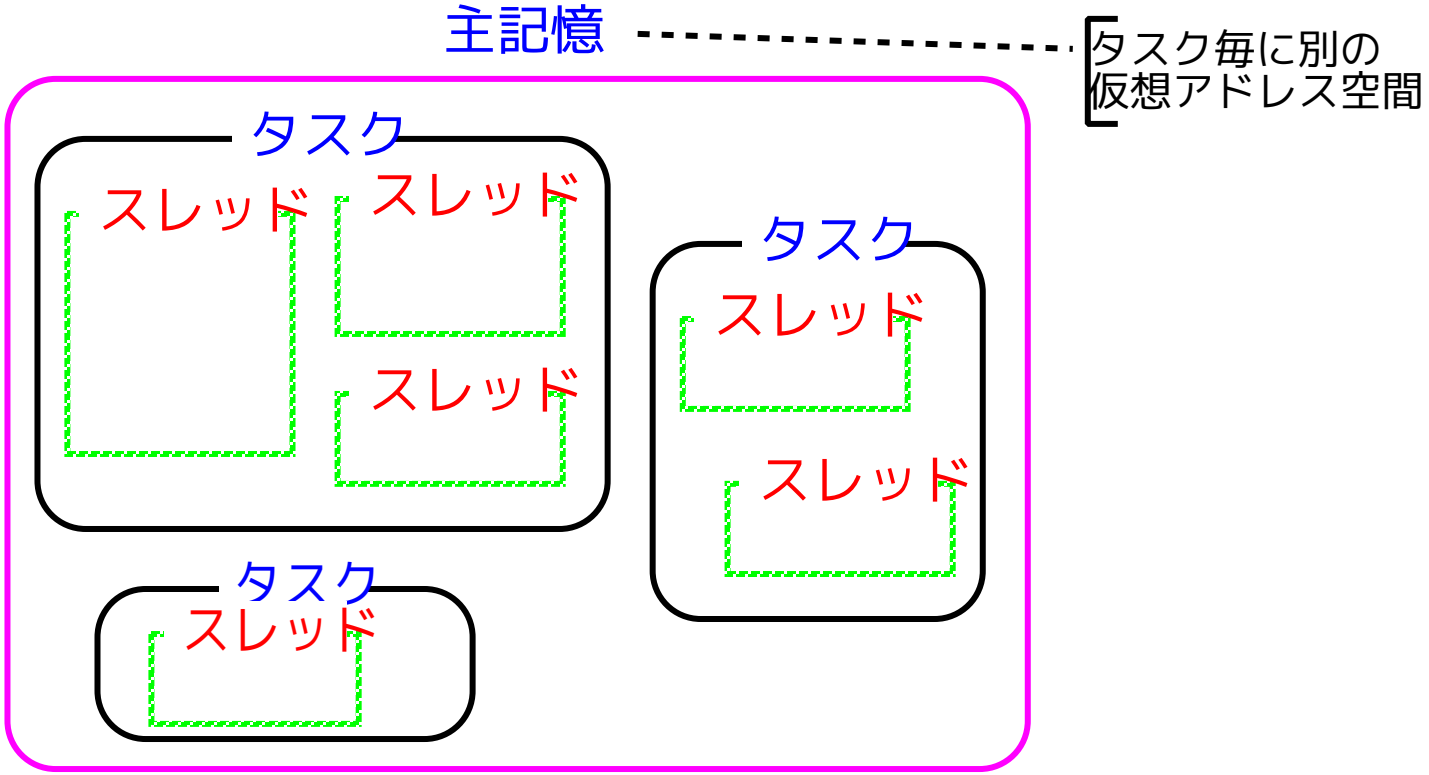
また、現在のUNIXにおいては、IEEE/ POSIX仕様のリアルタイム機能の中で「スレッド」の考えも採り入れられている。

POSIX:

2種類のUNIXを統合するために作られたプロジェクトの総称。最初の3文字は Portable Operating System の意味で、最後の IX は名前をUNIX風にする(unIXish)ために追加された。

- **MachやMVSにおいては**、
  1つのアドレス空間内に複数のプログラム実行の単位が存在し、これらが切替えられながら並行にそして協調/干渉し合って1つのまとまった仕事を行う。

Mach
の場合



⇨ アドレス空間は「1つのまとまった仕事」に対して割り当てられ、CPUは個々の「プログラム実行の単位」に対して割り当てられる。

Machの場合には

プログラム実行の単位をスレッド, まとまった仕事をタスクと呼んでいる。

⇨ 並列処理方式は多重スレッド。

MVSの場合には

プログラム実行の単位をタスク, まとまった仕事をジョブと呼んでいる。

⇨ 並列処理方式は多重タスキング。

- 多重スレッドの利点：
  - ◇ スレッド間での情報共有が容易。
  - ◇ スレッド生成時にはアドレス空間/メモリを新しく用意する必要が無い。
  - ◇ アドレス空間/メモリの切替えが少なくて済む。

補足：
仮想記憶空間の総量を (プロセスベースの場合に比べて) 少なく抑えることが出来るので、OSのオーバヘッドも少なくて済む。OSに負担をあまりかけないプロセスという意味で、スレッドを軽量プロセスと呼ぶことがある。

8–3 多重タスキングの利点

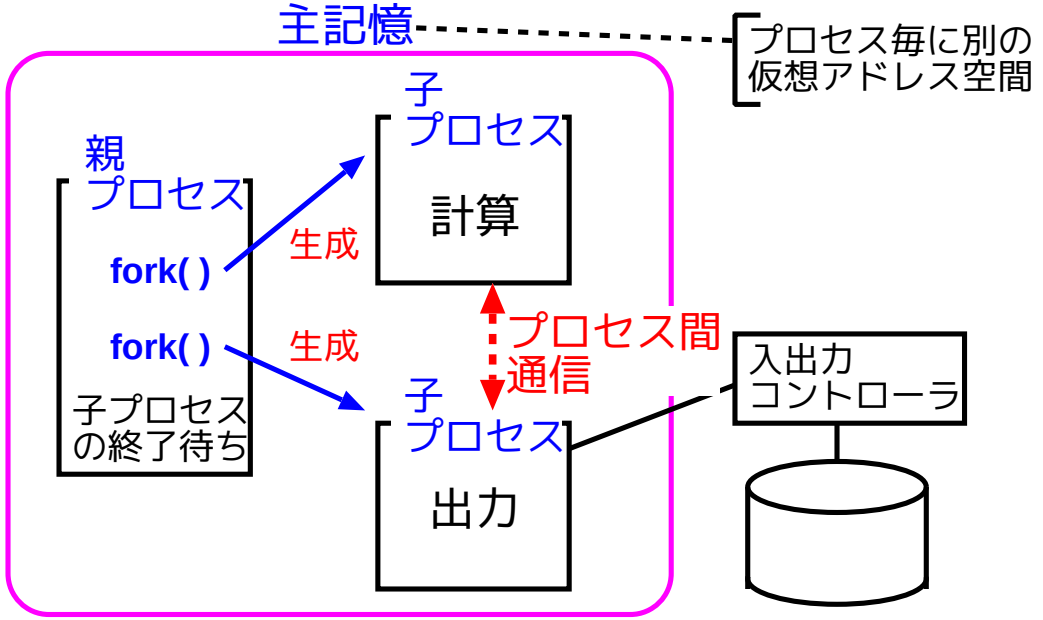
以下では 、
プロセス (やスレッド) が必要に応じて動的に別のプロセスを生成することができる状況下で、
複数のプロセスを並行して実行する処理方式を総称して多重タスキング (マルチタスキング) と呼ぶ。

利点(1) 並行して処理を進めると、適切な処理の切り替えによってシステム全体の処理効率が上がることがある。

例8. 1 ···多重プログラミング

例8. 2 (入出力とプログラム本体) 1つの処理の場合でも、入出力と計算の部分を別々のプロセスとして実行させると、

- CPUと入出力コントローラの並行動作が促進される。
- 2つのプロセスは適切に協調し合うことが期待できる。
- CPUが複数個ある場合、同じプログラムで2つのプロセスは同時に走ることになり、高速な処理が可能になる。



利点 (2) 信頼性向上に繋がる。

- **fail-soft** なシステムを容易に構築できる。
  具体的には、異常が起こっても、異常の起こったプロセスだけを終了させ残りのサービスは続行できる。
- **fault-tolerant** なシステムを容易に構築できる。
  具体的には、同一の仕事をするプロセスを複数用意しておけば、1つが異常終了しても残りの1つが動作してくれる。

補足： ...(岩波情報科学辞典からの引用)

フォールト・トレランス ... 故障の存在によってシステムの性能が低下することはあっても、全面的なシステムの停止に至ることはなく、外部から見る限り、予め定められた全部または一部の機能を正しく遂行する能力をいう。

フェール・ソフト ... システム内に故障が発生しても、システムの全面的な停止には至らないように、その影響をシステムの一部にとどめることの出来るシステムの能力をいう。
(fail が発生しても soft なものにする、ということ。)

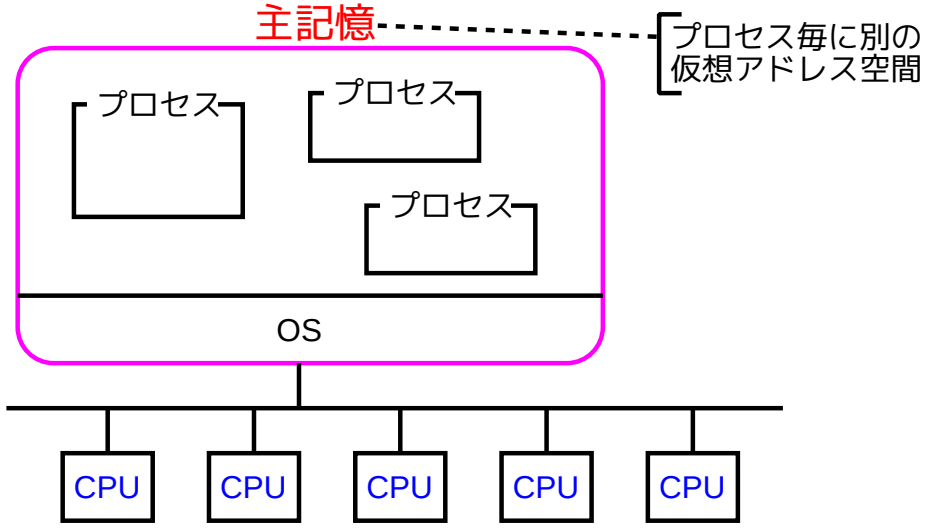
フォールト・アボイダンス ... 故障を予め除去して信頼性を実現しようとする。

フェール・セーフ ... 例. 故障時に赤信号になる交通信号。
(fail が発生しても safe なものにする、ということ。)

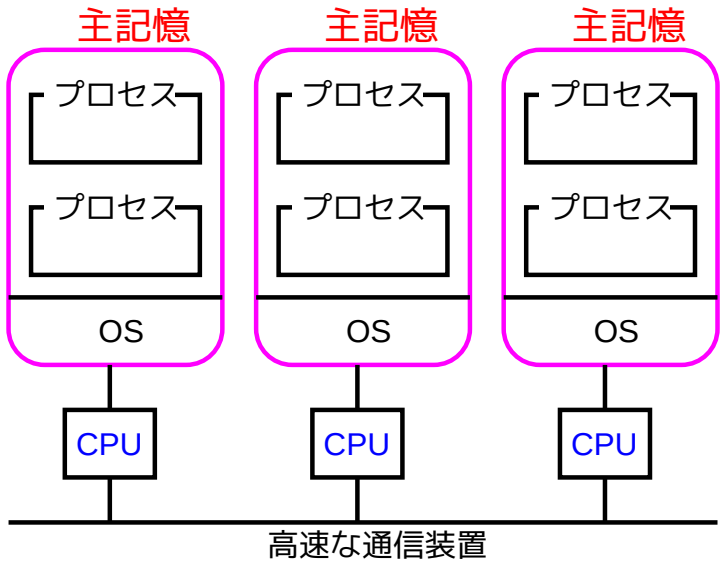
利点 (3) 複数のCPUによる多重プロセシング環境においても、ソフトウェアを再設計することなく直ちにハードウェアに見合った性能向上が期待できる。

多重プロセッサの構成は次の2つがある。

- 密結合多重プロセッサ··· CPUが主記憶を共有
- 疎結合多重プロセッサ··· CPUは主記憶を共有しないが、高速な通信装置で結ばれている



密結合



疎結合

8–4 プロセス生成等のシステムコール—プロセス

プロセス生成のシステムコール　pid_t fork(void) :

UNIXにおいては、

- ヘッダファイル <unistd.h> の中で宣言されている。
- fork() が呼ばれると、現在実行中のプロセスと全く同じ内容のメモリ空間とCPU 状態を持つプロセスが新に作られ、元々のプロセス(親プロセスと言う)と新に作られたプロセス(子プロセスと言う)が並行して動作する。
- プロセス生成に成功すると、
  親プロセスの方ではfork()の関数値として 子プロセスのID が返され、子プロセスの方ではfork()の関数値として 0 が返される。
  ⇨ fork()の値を調べて分岐させれば、親プロセスと子プロセスの動作は違ってくる。
- プロセス生成に失敗すると、関数値 −1 を返す。

例8. 3 (プロセス生成,fork( ))

プロセス生成して、2つのプロセスで各々、自分と親プロセスのIDを出力するプログラムを次に示す。

```
[motoki@x205a]$  nl fork-and-run-concurrently.c
 1 /*************************************************************
 2 /* Operating-Systems/C-Programs/fork-and-run-concurrently1.
 3 /*-----------------------------------------------------------
 4 /* fork()システムコール使用例
 5 /* C.オール「例題で学ぶLinuxプログラミング」ピアソン,4.3.1*/
 6 /*************************************************************
 7 #include <stdio.h>
 8 #include <stdlib.h> /* for exit() library function */
 9 #include <unistd.h> /* for fork(), getpid() */
10                     /*  and getppid()       */
                       /*  system calls        */
11 int main(void)
12 {
```

```
13   pid_t childpid;
                                                  大域変数 errno
14   if ((childpid=fork())==-1) { /*forkに失敗  */
15     perror("can't fork");       /*したら、ここ*/
16     exit(EXIT_FAILURE);
17   }else if (childpid==0) {  /* 子プロセスはこちら */
18     printf("I am a child process.\n");
19     printf("  >>(In child) getpid()=%d\n",getpid());
20     printf("  >>(In child) getppid()=%d\n", getppid());
21     exit(EXIT_SUCCESS);
22   }else {                          /* 親プロセスはこちら */
23     printf("I am a parent process.\n");
24     printf("  (In parent) getpid()=%d\n", getpid());
25     printf("  (In parent) getppid()=%d\n", getppid());
26   }

27   return 0;
```

```
28 }
[motoki@x205a]$  gcc fork-and-run-concurrently.c
[motoki@x205b]$  ./a.out
I am a parent process.
I am a child process.
  >>(In child) getpid()=10674
  >>(In child) getppid()=10673
  (In parent) getpid()=10673
  (In parent) getppid()=10307
[motoki@x205a]$
```

2つのプロセスは非同期的に実行

a.outを実行したbashのプロセス番号

```
[motoki@x205b]$  ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TI
root           1  0.0  0.0   1788   592 ?        Ss   08:40   0:
root           2  0.0  0.0      0     0 ?        S<   08:40   0:
     ..........................................................
motoki     10307  0.0  0.0   5120  1560 pts/1    Ss   10:57   0:
```

```
     ..................................................
[motoki@x205b]$
```

子プロセスの終了を待つためのシステムコール

pid_t wait(int *status)：

先に行う処理,下位の処理を子プロセスに割当て、親プロセスの方ではシステムコール wait() を用いて子プロセスの終了を待つ様にすることが出来る。

- ヘッダファイル <sys/types.h> と<sys/wait.h> を必要とする。

- wait() が呼ばれると、子プロセスのどれかが終了する（かシグナルを受け取る）までこのプロセスは停止する。

- 子プロセスのどれかが終了すると、その親プロセスがwait()(またはwaitpid()) システムコールを実行していた場合には、終了した子プロセスのIDがwait()の関数値として親プロセスに返される。 同時に、その終了状態値(exit status)が関数引数で指定された変数 status にセットされる。

- おまけ 親プロセスがwait()(やwaitpid()) システムコールを実行する前に子プロセスが終了してしまった場合は、
(親プロセスからwait()やwaitpid() が出されるまで)終了した子プロセスの終了状態値を引き取るプロセスがないため、終了した子プロセスのプロセス管理表は削除されずに残ったままになってしまう。こういったプロセスをゾンビ(**zombie**)と呼んでいる。

  補足：
  ps aux コマンドを行った時、状態(STATの欄) が Z のプロセスがゾンビである。

- おまけ 子プロセスよりも前に親プロセスが終ってしまい、子プロセスが孤児(**orphan**)になってしまった場合は、
initプロセスが子の親となり終了状態値を引き取るので子プロセスはゾンビにはならない。

- おまけ 子プロセスがない状態では wait() の実行は失敗に終る。

例8. 4 (子プロセスの終了を待つ, wait( )) 例8.3のプログラムで、fork( )を行った直後に親プロセスの方でシステムコールwait( )を実行する様にしてみた。 次の通り。

```
[motoki@x205a]$  nl fork-run-and-wait.c
 1 /*************************************************************
 2 /* Operating-Systems/C-Programs/fork-run-and-wait.c
 3 /*-------------------------------------------------------------
 4 /* fork()とwait()システムコールの使用例
 5 /* C.オール「例題で学ぶLinuxプログラミング」ピアソン,4.3.2*/
 6 /*************************************************************
 7 #include <stdio.h>
 8 #include <stdlib.h>     /* for exit() library function */
 9 #include <unistd.h>     /* for fork(), getpid()        */
10                         /*  and getppid() system calls */
11 #include <sys/types.h>  /* for wait() system call */
12 #include <sys/wait.h>   /* for wait() system call */
```

```
13 int main(void)
14 {
15   pid_t childpid, exited_pid;
16   int status;

17   if ((childpid=fork())==-1) { /* forkに失敗 */
18     perror("can't fork");       /*したら、ここ*/
19     exit(EXIT_FAILURE);
20   }else if (childpid==0) {  /* 子プロセスはこちら */
21     printf("I am a child process.\n");
22     printf("  >>(In child) getpid()=%d\n",getpid());
23     printf("  >>(In child) getppid()=%d\n", getppid());
24     exit(EXIT_SUCCESS);
25   }else {                          /* 親プロセスはこちら */
26     exited_pid=wait(&status);   → 親子の出力が混ざらない
27     printf("I am a parent process.\n");
28     printf("  My child (ID=%d) exited "
```

```
                        "with status %#x just now.\n",
29              exited_pid, status);
30      printf("  (In parent) getpid()=%d\n",getpid());
31      printf("  (In parent) getppid()=%d\n", getppid());
32    }

33    return 0;
34  }
[motoki@x205a]$  gcc fork-run-and-wait.c
[motoki@x205a]$  ./a.out
I am a child process.                          |
  >>(In child) getpid()=11126                  |子
  >>(In child) getppid()=11125                 |
I am a parent process.                                                  |
  My child (ID=11126) exited with status 0 just now.                    |親
  (In parent) getpid()=11125                                            |
  (In parent) getppid()=10307                                           |
```

```
[motoki@x205a]$
```

別のプログラムをoverlayして実行するための システムコール群execv(), ...:

現在実行しているプロセスを別のプログラムで置き換えて実行をすることも出来る。

- 各々の関数プロトタイプは

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
                                        char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[],
                                        char *const envp[]);
```

で、ヘッダファイル <unistd.h> の中で宣言されている。

- これらexecファミリの関数が呼ばれると、
  引数で指定されたプログラムの実行イメージが現プロセスのイメージに完全に置き換わり実行される。

- 関数引数の path は、バイナリ実行モジュールかスクリプトをフルパスで指定した文字列を表す。

```
int execl(const char *path, const char *arg, ...);
int execle(const char *path, const char *arg, ..., ...);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], ...);
```

- 関数引数の file は ファイル名を表す文字列である。ファイル名にスラッシュが含まれない場合はシェルの行動をまねて $PATH を探してバイナリが設定される。

```
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
```

- 関数引数の “arg, ...” の部分は 、プログラム実行を指定したコマンドライン上の空白で区切られた文字列をコンマ(,)で区切って順に並べ、最後にコンマとNULLポインタを付け加えたものである。
  例えば次のように指定する。

```
execl("/bin/cat", "/bin/cat",
                          "/etc/passwd", "/etc/group", NULL);
execlp("cat", "cat", "/etc/passwd", "/etc/group", NULL);
```

- 関数引数の argv は 、プログラム実行を指定したコマンドライン上の空白で区切られた文字列へのポインタとNULLポインタが並んだ配列が別途構成されているとして、そのポインタ配列へのポインタを指定したものである。 (C言語のmain関数の第2引数である argv と同じデータ型、同じ構成になっている。) 例えば次のように指定する。

```
char *argv[] = {"/bin/cat",
                          "/etc/passwd", "/etc/group", NULL};
execv("/bin/cat", argv);
```

- 関数引数の envp を指定することによってoverlayするプログラムに特殊な実行環境を設定することが出来る。実際には envp は

  環境変数名 = 環境変数の値

  という文字列へのポインタとNULLポインタが並んだ配列が別途構成されているとして、そのポインタ配列へのポインタを指定したものである。

  例えば次のように指定する。

```
char *argv[] = {"/bin/cat",
                "/etc/passwd",
                "/etc/group",
                NULL};
char *envp[] = {"PATH=/bin:/usr/bin",
                "USER=motoki",
                NULL};
execve("/bin/cat", argv, envp);
```

例8. 5 (別のプログラムをoverlayして実行,execvp( ))

例8.4のプログラムで、システムコールexecvp( )を用いて子プロセスの方に階乗計算の実行イメージをoverlayする様にしてみた。 次の通り。

```
[motoki@x205a]$  nl fork-overlay-run-and-wait.c
 1 /*************************************************
 2 /* Operating-Systems/C-Programs/fork-overlay-run-and-wait.c
 3 /*-------------------------------------------------
 4 /* fork(),wait()とexecv()システムコールの使用例*/
 5 /* [参考] C.オール「例題で学ぶLinuxプログ*/
 6 /*              ラミング」ピアソン,4.3.1-2節*/
 7 /*************************************************
 8 #include <stdio.h>
 9 #include <stdlib.h>     /* for exit() library function */
10 #include <unistd.h>     /* for fork(), getpid(), getppid()
11                         /*        and execv() system calls
12 #include <sys/types.h>  /* for wait() system call */
13 #include <sys/wait.h>   /* for wait() system call */
```

```
14 int main(void)
15 {
16   pid_t childpid, exited_pid;
17   int status;
18   char *argv[]={"fundamentals-factorial", NULL};

19   if ((childpid=fork())==-1) { /* forkに失敗 */
20     perror("can't fork");      /*したら、ここ*/
21     exit(EXIT_FAILURE);
22   }else if (childpid==0) {  /* 子プロセスはこちら */
23     printf("I am a child process.\n");
24     printf("  >>(In child) getpid()=%d\n",getpid());
25     printf("  >>(In child) getppid()=%d\n", getppid());
26     printf("  >>I will overlay "
                 "fundamentals-factorial and run.\n");
27     if(execvp("./fundamentals-factorial",argv)==-1){
```

```
28        perror("can't execvp");
29        exit(EXIT_FAILURE);
30      }
31      printf("ここには来ない。\n");   実行されることはない
32      exit(EXIT_SUCCESS);             実行されることはない
33    }else {                          /* 親プロセスはこちら */
34      exited_pid=wait(&status);
35      printf("-----\n"
36             "I am a parent process.\n");
37      printf("  My child (ID=%d) exited "
               "with status %#x just now.\n",
38             exited_pid, status);
39      printf("  (In parent) getpid()=%d\n", getpid());
40      printf("  (In parent) getppid()=%d\n", getppid());
41    }

42    return 0;
```

```
43 }
[motoki@x205a]$  gcc fork-overlay-run-and-wait.c
[motoki@x205a]$  ./a.out
I am a child process.
  >>(In child) getpid()=20347
  >>(In child) getppid()=20346
  >>I will overlay fundamentals-factorial and run.
何の階乗を求めますか?: 5
5! = 120.000000 (approximation)
-----
I am a parent process.
  My child (ID=20347) exited with status 0 just now.
  (In parent) getpid()=20346
  (In parent) getppid()=10307
[motoki@x205a]$
```

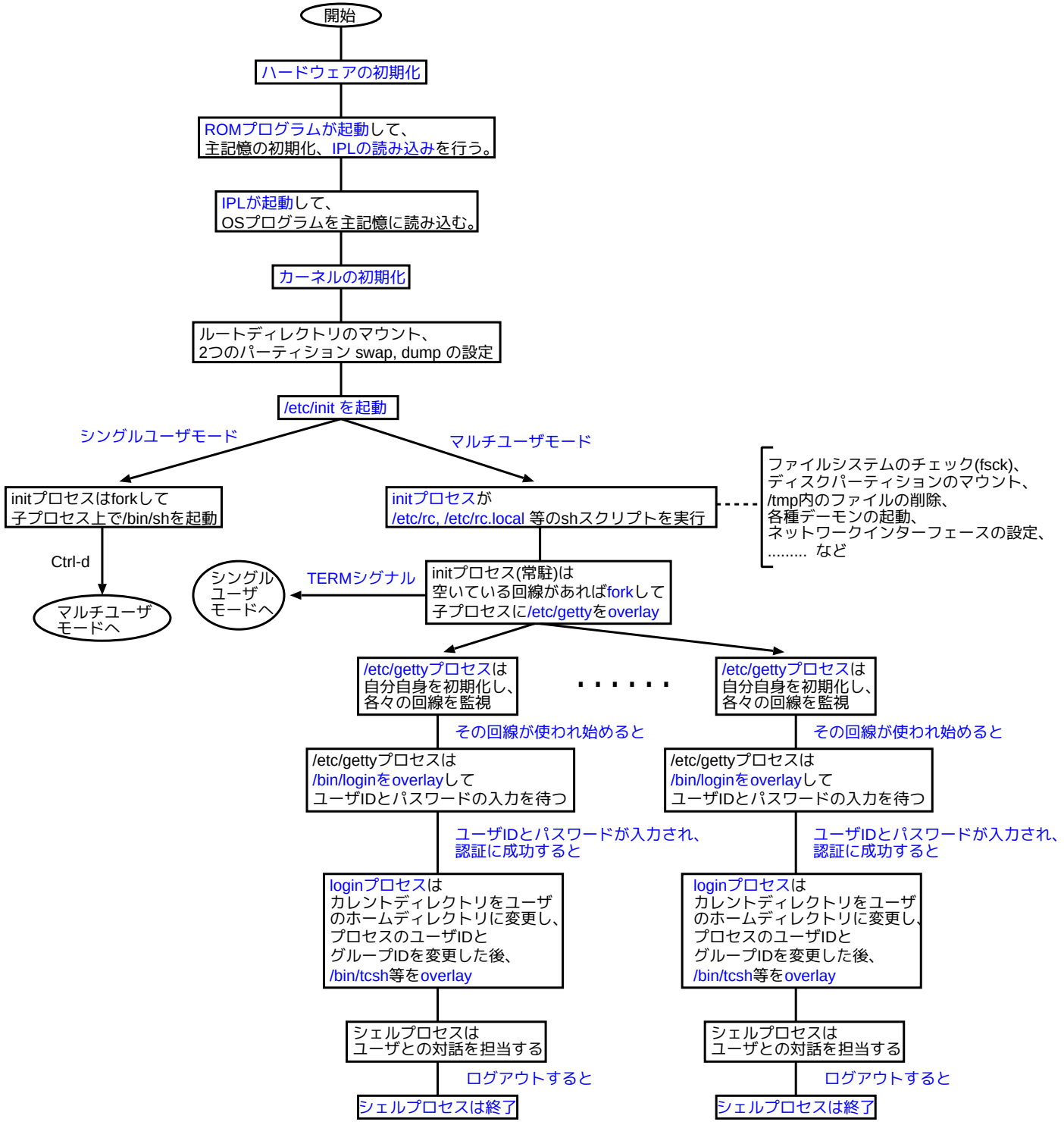
8–5 UNIX起動時に生成されるプロセス群—プロ

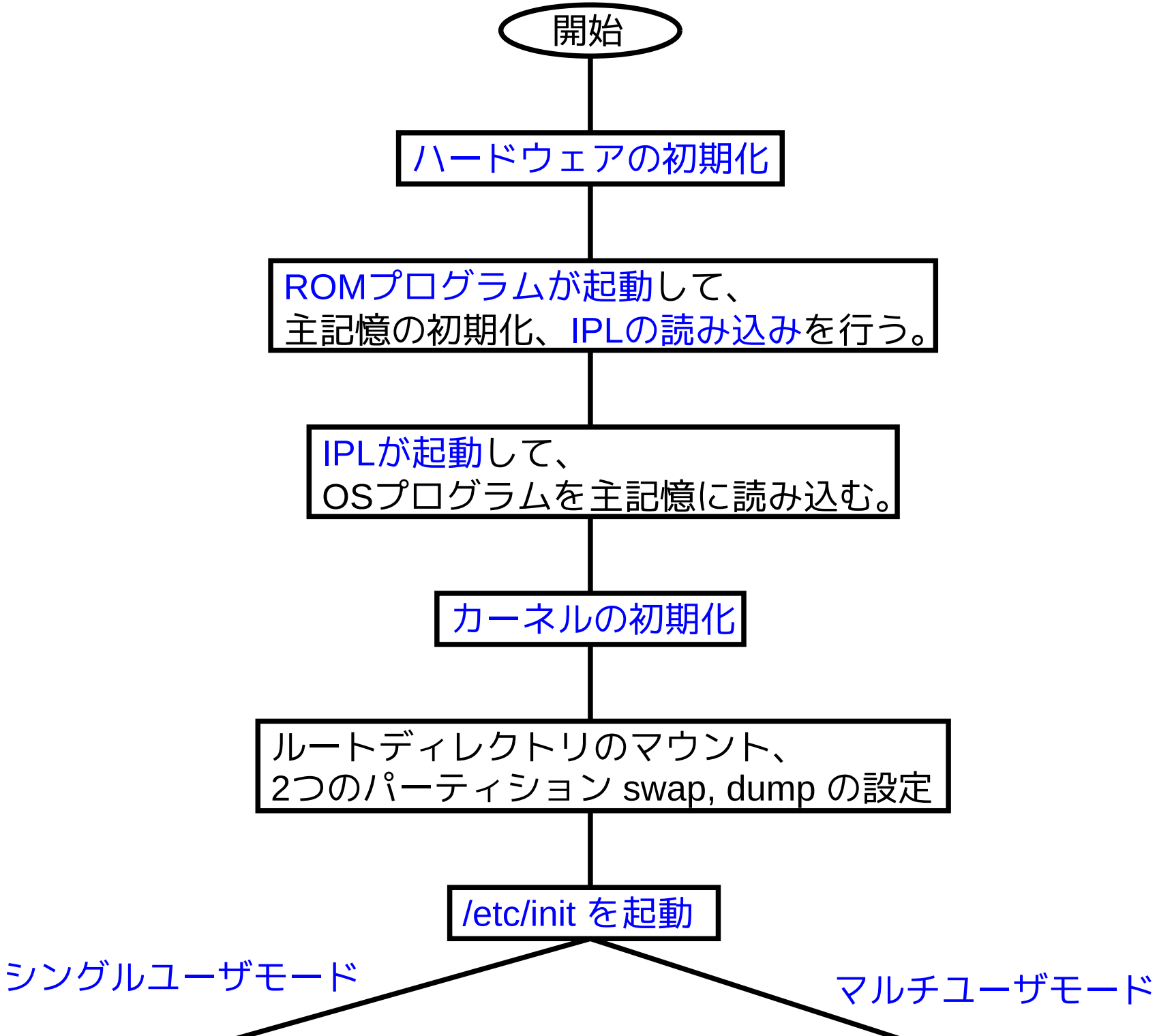
補足： ···(p.15からの引用)
谷口2.1.1節によれば、ブートストラップの部分の処理を詳しく書くと次のようになる。
(1.1) ハードウェアの初期化。
(1.2) ROMプログラムが起動して、主記憶の初期化、IPL (Initial Program Loader)の読み込みが行われる。
(1.3) IPLが起動し、OSプログラムを主記憶に読み込む。
(1.4) OSの初期化。

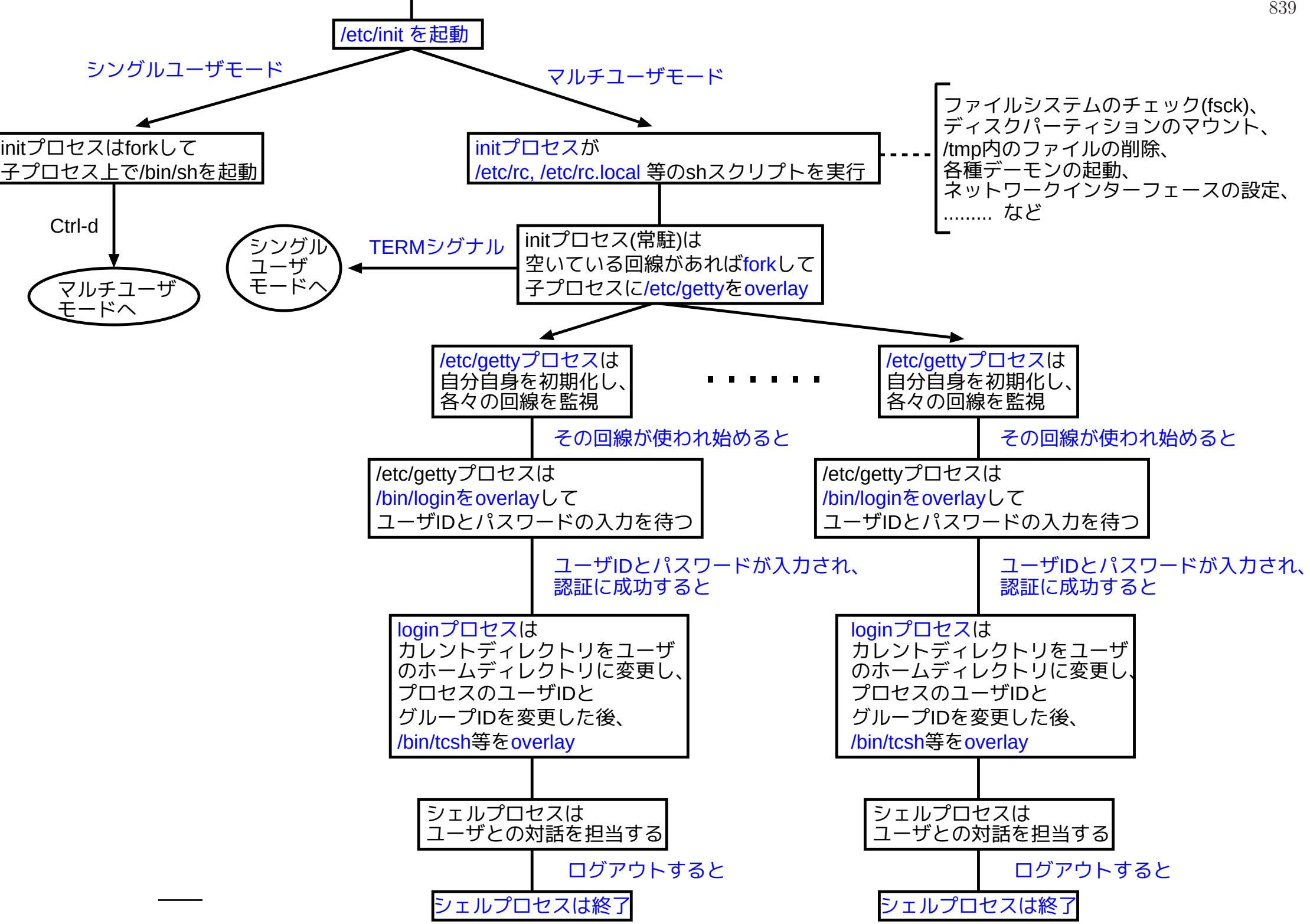
UNIXにおける起動プロセス：

(1) ハードウェアの初期化。
(2) ROMプログラムが起動して、主記憶の初期化、IPLの読み込みが行われる。
(3) IPLが起動し、OSプログラムを主記憶に読み込む。
(4) カーネルの初期化。

. . . . . .


開始
ハードウェアの初期化
ROMプログラムが起動して、
主記憶の初期化、IPLの読み込みを行う。
IPLが起動して、
OSプログラムを主記憶に読み込む。
カーネルの初期化
ルートディレクトリのマウント、
2つのパーティション swap, dump の設定
/etc/init を起動
シングルユーザモード
マルチユーザモード
initプロセスはforkして
子プロセス上で/bin/shを起動
initプロセスが
/etc/rc, /etc/rc.local 等のshスクリプトを実行
ファイルシステムのチェック(fsck)、
ディスクパーティションのマウント、
/tmp内のファイルの削除、
各種デーモンの起動、
ネットワークインターフェースの設定、
..... など
Ctrl-d
マルチユーザ
モードへ
シングル
ユーザ
モードへ
TERMシグナル
initプロセス(常駐)は
空いている回線があればforkして
子プロセスに/etc/gettyをoverlay
/etc/gettyプロセスは
自分自身を初期化し、
各々の回線を監視
/etc/gettyプロセスは
自分自身を初期化し、
各々の回線を監視
その回線が使われ始めると
その回線が使われ始めると
/etc/gettyプロセスは
/bin/loginをoverlayして
ユーザIDとパスワードの入力を待つ
/etc/gettyプロセスは
/bin/loginをoverlayして
ユーザIDとパスワードの入力を待つ
ユーザIDとパスワードが入力され、
認証に成功すると
ユーザIDとパスワードが入力され、
認証に成功すると
loginプロセスは
カレントディレクトリをユーザ
のホームディレクトリに変更し、
プロセスのユーザIDと
グループIDを変更した後、
/bin/tcsh等をoverlay
loginプロセスは
カレントディレクトリをユーザ
のホームディレクトリに変更し、
プロセスのユーザIDと
グループIDを変更した後、
/bin/tcsh等をoverlay
シェルプロセスは
ユーザとの対話を担当する
シェルプロセスは
ユーザとの対話を担当する
ログアウトすると
ログアウトすると
シェルプロセスは終了
シェルプロセスは終了


開始
ハードウェアの初期化
ROMプログラムが起動して、
主記憶の初期化、IPLの読み込みを行う。
IPLが起動して、
OSプログラムを主記憶に読み込む。
カーネルの初期化
ルートディレクトリのマウント、
2つのパーティション swap, dump の設定
/etc/init を起動
シングルユーザモード
マルチユーザモード


/etc/init を起動
シングルユーザモード
マルチユーザモード
initプロセスはforkして
子プロセス上で/bin/shを起動
initプロセスが
/etc/rc, /etc/rc.local 等のshスクリプトを実行
ファイルシステムのチェック(fsck)、
ディスクパーティションのマウント、
/tmp内のファイルの削除、
各種デーモンの起動、
ネットワークインターフェースの設定、
..... など
Ctrl-d
シングル
ユーザ
モードへ
TERMシグナル
initプロセス(常駐)は
空いている回線があればforkして
子プロセスに/etc/gettyをoverlay
マルチユーザ
モードへ
/etc/gettyプロセスは
自分自身を初期化し、
各々の回線を監視
/etc/gettyプロセスは
自分自身を初期化し、
各々の回線を監視
その回線が使われ始めると
その回線が使われ始めると
/etc/gettyプロセスは
/bin/loginをoverlayして
ユーザIDとパスワードの入力を待つ
/etc/gettyプロセスは
/bin/loginをoverlayして
ユーザIDとパスワードの入力を待つ
ユーザIDとパスワードが入力され、
認証に成功すると
ユーザIDとパスワードが入力され、
認証に成功すると
loginプロセスは
カレントディレクトリをユーザ
のホームディレクトリに変更し、
プロセスのユーザIDと
グループIDを変更した後、
/bin/tcsh等をoverlay
loginプロセスは
カレントディレクトリをユーザ
のホームディレクトリに変更し、
プロセスのユーザIDと
グループIDを変更した後、
/bin/tcsh等をoverlay
シェルプロセスは
ユーザとの対話を担当する
シェルプロセスは
ユーザとの対話を担当する
ログアウトすると
ログアウトすると
シェルプロセスは終了
シェルプロセスは終了

(5) ルートディレクトリをマウントし、swap，dumpという2つのパーティションを設定。

但し、これは古い Sun SPARCstationの場合。　swapはスワップ領域、dumpはカーネルが異常終了した時にカーネルのプログラムをデバッグするための情報を保存する領域。

(6) /etc/initを起動。これが最初のUNIXプロセスで、プロセスIDは1、所有者はroot(user id=0)である。

但し、これは古い Sun SPARCstationの場合。　VineLinux 2.1.5だと/sbin/init。

UNIXがマルチユーザモードで動作している場合は、

(6.1) initプロセスは/etc/rc というshシェルスクリプトを実行。これによって各種デーモンプロセスが起動される。

(6.2) 引き続いて、 initプロセスは
/etc/ttysに指定された端末の個数分だけforkして、各々の子プロセス上に端末制御の/etc/gettyプログラムを overlay する。

(6.2.1) 各/etc/gettyプロセスは
自分自身を初期化し、各々のコンソールや端末に繋がる回線を監視する。

(6.2.2) ユーザIDが回線から入力されると、その端末を担当する/etc/gettyプロセスは
ユーザID を引数として/bin/loginというプログラムを overlay・実行してパスワードの入力を要求する。

(6.2.3) パスワードが入力されると、 /bin/login プロセスは
システム内の/etc/ passwd ファイルを使ってユーザIDの照合を行う。

—

(6.2.4) ユーザ認証に成功すれば、 loginプロセスは
カレントディレクトリをユーザのホームディレクトリに変更し、
プロセスのユーザID,グループIDを変更した後、指定された
シェルプログラム(e.g./bin/tcsh)を**overlay**して実行する。

(6.2.5) ユーザは指定したシェルと会話しながら自分の作業を進
める。

(6.2.6) ユーザがログオフするとそのシェルのプロセスは終了す
る。

(6.2.7) /etc/gettyに始まるプロセスが終了すると制御がinit(プ
ロセスid=1)に戻り、 目を覚ましたinitプロセスは
再びforkして端末ポート上に/etc/gettyを起動する。

補足：

これは古い **Sun SPARCstation** の場合であるが、大体の流れは今も変わっていない。

VineLinux2.1.5 だと

`/sbin/init` の動作を設定するための`/etc/inittab` というファイルが用意され、この設定に従った動作の中で各種デーモンの起動や端末制御のプログラムの起動が行われる。

まず、上記ステップ **(6.1)** に相当する手続として、`/etc/rc.d/rc.sysinit`, `/etc/rc.d/rc` という **2** つの **sh** スクリプトが実行される。`rc.sysinit` はハードウェアのチェックと初期化を行うのが主な仕事であるが、システムクロックの調整、スワップの有効化、キーボードマップのロード、ファイルシステムのマウントといったシステム設定も行っている。また、`/etc/rc.d/rc` は `/etc/rc.d/rc`ランレベル`.d` というディレクトリ内のスクリプトを実行することによって各種デーモンの起動制御や `/etc/rc.d/rc.local` という **sh** スクリプトの実行を行う。

ステップ **(6.2)** に関しては、`/etc/ttys` というファイルはなく、端末制御のプログラムとして `/sbin/mingetty` が **overlay**・起動される。

デーモンプロセス：

OSを運用するために裏で常に働いているプロセスのことを**UNIX**では
デーモン(**daemon**)と呼ぶ。

- 大半のデーモンは `/etc/rc` 等の、ブート直後に実行される**sh**スクリプトの中で起動され、システムの稼働中は実行し続ける。

- カーネルの一部と考えられるものから、
  インターネットを通じてサービスを提供するもの
  (インターネットデーモンという)
  まで様々のものがある。

- カーネルの一部と考えられる代表的なデーモン
  [これらの内 **pagedaemon** と **swapper** は実際にはカーネルの一部であるがスケジューリングの都合上プロセスの様に見せかけているものである。]

  (init) ··· 全てのプロセスの親 (または先祖) となる最も重要なデーモンで、**PID = 1** である。

  (pagedaemon) ··· **BSD** 系のシステムにおけるページャと呼ばれるデーモンで、**PID = 2** である。アクセスされた仮想記憶のページが主記憶上にない時に呼び出される。

  (swapper) ··· **BSD** 系のシステムにおけるデーモンで、**PID = 0** である。ページフォールトの起きる頻度を常に監視し、ページフォールトが頻繁に起こりすぎる場合、スワップ領域に追い出してしまうプロセスを選び出す。

  (update) ··· **BSD** 系のシステムにおけるデーモン。**30** 秒おきに sync システムコールを実行して、...

- システム管理のための代表的なデーモン

  (syslogd) $\cdots$ システムのログファイルを書き出すデーモン。

  (cron) $\cdots$ 指定した日時に指定したプログラムを起動するデーモン。

  (sendmail) $\cdots$ メールを送ったり受け取ったりするデーモン。

  (X) $\cdots$ **X Window System**のサーバ。

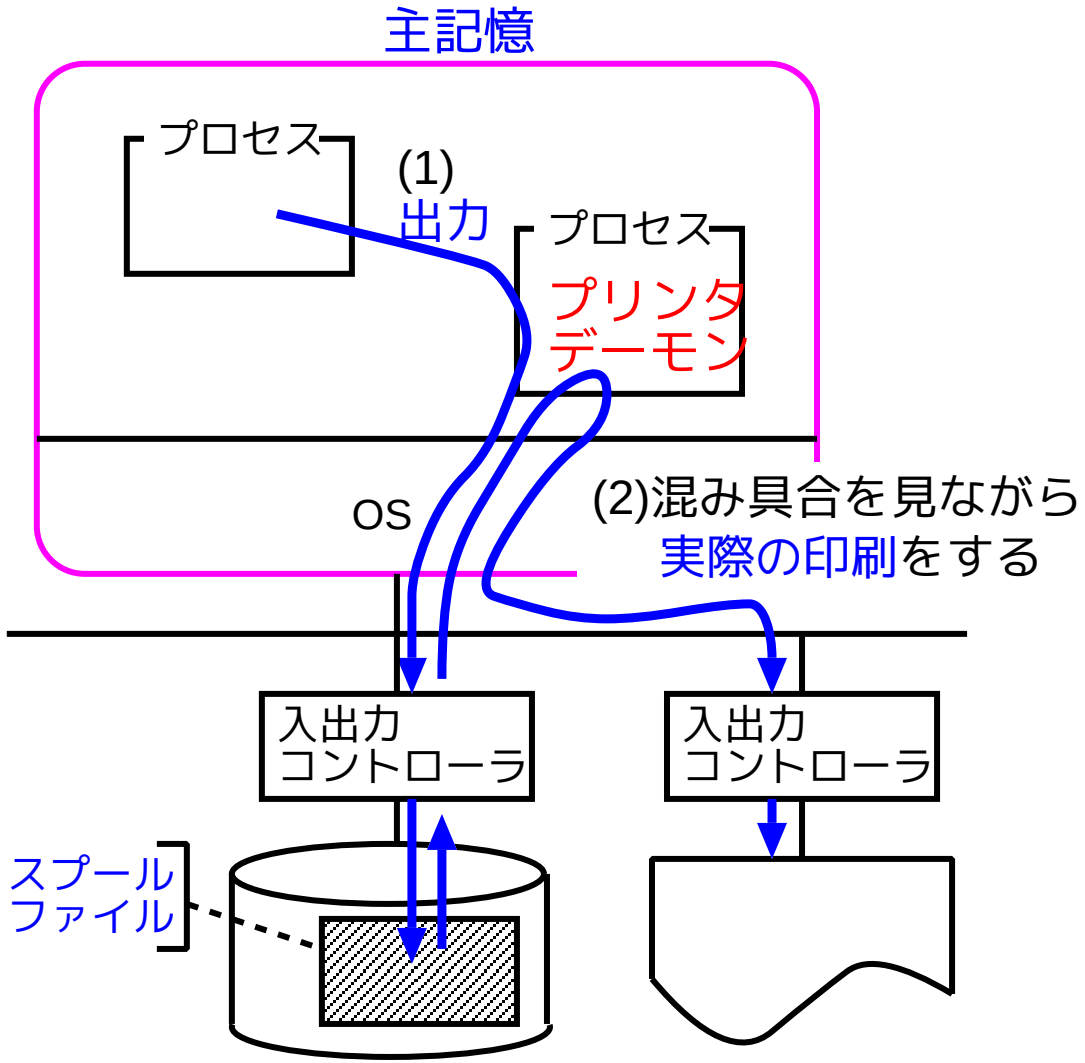
  (jserver) $\cdots$ **Wnn**のかな漢字変換サーバ。

  (cannaserver) $\cdots$ かな漢字変換サーバ**(Canna)**。

  (lpd) $\cdots$ プリンタデーモン。 **.....**

(lpd) $\cdots$ プリンタデーモン。印刷要求に応じて印刷データを磁気ディスク上のスプール(spool)ファイルに入れ、プリンタの状況を見ながら書き込みが完了したスプールファイルのデータを実際にプリンタに送ったりしている。

⇨ 印刷後に長い計算に入り実際に印刷はしないが(印刷の連続性を保つために) プリンタを占有してしまうプロセスの問題を解決することが出来る。



- 代表的なインターネットデーモン

  (inetd) ··· インターネットスーパーサーバと呼ばれるデーモン。多くのネットワークデーモンを常時起動しておく代わりにこのデーモンだけを常時起動しておき、このデーモンが要求に応じて適宜必要なネットワークデーモンを起動する。

  (httpd) ··· **Web**サーバ。

  (ftpd) ··· **ftp**サーバ。

例8. 6 (VineLinux2.1.5起動直後のプロセス群) VineLinux2.1.5を起動してランレベル=5 としてemacs と xdvi を立ち上げた直後にシステム内に存在しているプロセスの情報を以下に示す。

プロセスが生成された順にPIDが割り振られ並んでいるので、/etc/rc.d/rc や/etc/rc.d/rc5.d のshスクリプトによってプロセスが生成されている様子が観察できる。

```
[motoki@x205b]$  ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT   START    TIME COMMAND
root         1  3.2  0.2   1064   480 ?        S      23:16    0:05 init [5]
root         2  0.0  0.0      0     0 ?        SW     23:16    0:00 [kflushd]
root         3  0.0  0.0      0     0 ?        SW     23:16    0:00 [kupdate]
root         4  0.0  0.0      0     0 ?        SW     23:16    0:00 [kpiod]
root         5  0.0  0.0      0     0 ?        SW     23:16    0:00 [kswapd]
root         6  0.0  0.0      0     0 ?        SW<    23:16    0:00 [mdrecoveryd]
root         7  0.0  0.0      0     0 ?        SW     23:16    0:00 [khubd]
bin        360  0.0  0.2   1164   408 ?        S      23:16    0:00 portmap
root       383  0.0  0.2   1104   572 ?        S      23:16    0:00 rpc.statd
root       397  0.0  0.2   1048   468 ?        S      23:16    0:00 /usr/sbin/apmd
```

```
root      441  0.0  0.3  1168   580 ?        S      23:16   0:00 /usr/sbin/autc
root      494  0.1  0.2  1120   528 ?        S      23:16   0:00 syslogd -m 0
root      503  0.0  0.4  1492   876 ?        S      23:16   0:00 klogd
nobody    517  0.0  0.3  1232   632 ?        S      23:17   0:00 identd -e -o
nobody    520  0.0  0.3  1232   632 ?        S      23:17   0:00 identd -e -o
nobody    521  0.0  0.3  1232   632 ?        S      23:17   0:00 identd -e -o
nobody    523  0.0  0.3  1232   632 ?        S      23:17   0:00 identd -e -o
nobody    524  0.0  0.3  1232   632 ?        S      23:17   0:00 identd -e -o
daemon    535  0.0  0.2  1092   504 ?        S      23:17   0:00 /usr/sbin/atd
root      549  0.0  0.3  1288   616 ?        S      23:17   0:00 crond
root      580  0.0  0.3  1188   620 ?        S      23:17   0:00 /sbin/cardmgr
root      595  0.0  0.2  1064   452 ?        S      23:17   0:00 usbmgr
root      610  0.0  0.2  1092   492 ?        S      23:17   0:00 inetd
root      619  0.1  0.5  2296  1084 ?        S      23:17   0:00 sshd
root      635  0.0  0.2  1152   536 ?        S      23:17   0:00 lpd
root      706  0.0  0.3  1592   736 ?        S      23:17   0:00 /usr/lib/postf
postfix   709  0.0  0.3  1580   700 ?        S      23:17   0:00 pickup -l -t f
postfix   710  0.0  0.4  1700   800 ?        S      23:17   0:00 qmgr -l -t fif
root      722  0.0  0.2  1092   440 ?        S      23:17   0:00 gpm -t ps/2
root      736  0.0  0.7  2624  1396 ?        S      23:17   0:00 httpd
```

```
nobody   739  0.0  0.7  2832 1484 ?      S    23:17   0:00 httpd
nobody   740  0.0  0.7  2832 1484 ?      S    23:17   0:00 httpd
nobody   741  0.0  0.7  2832 1484 ?      S    23:17   0:00 httpd
nobody   742  0.0  0.7  2832 1484 ?      S    23:17   0:00 httpd
nobody   743  0.0  0.7  2832 1484 ?      S    23:17   0:00 httpd
nobody   744  0.0  0.7  2832 1484 ?      S    23:17   0:00 httpd
nobody   745  0.0  0.7  2832 1484 ?      S    23:17   0:00 httpd
nobody   746  0.0  0.7  2832 1484 ?      S    23:17   0:00 httpd
nobody   762  0.0  0.4  1808  928 ?      S    23:17   0:00 proftpd (accep
wnn      776  0.0  2.0  4836 3872 ?      S    23:17   0:00 /usr/bin/jserv
xfs      791  1.4  4.7  9980 9064 ?      S    23:17   0:01 xfs -droppriv
bin      801  0.0  0.6  1964 1340 ?      S    23:17   0:00 /usr/sbin/cann
root     823  0.0  0.2  1036  392 tty1     S    23:17   0:00 /sbin/mingetty
root     824  0.0  0.2  1036  392 tty2     S    23:17   0:00 /sbin/mingetty
root     825  0.0  0.2  1036  392 tty3     S    23:17   0:00 /sbin/mingetty
root     826  0.0  0.2  1036  392 tty4     S    23:17   0:00 /sbin/mingetty
root     827  0.0  0.2  1036  392 tty5     S    23:17   0:00 /sbin/mingetty
root     828  0.0  0.2  1036  392 tty6     S    23:17   0:00 /sbin/mingetty
root     829  0.0  0.6  3536 1204 ?      S    23:17   0:00 /usr/X11R6/bin
root     862  2.8  5.1 16548 9932 ?      R    23:17   0:03 /etc/X11/X -au
```

```
root     863  0.0  0.9  4404 1864 ?      S    23:17   0:00 -:0
motoki   879  0.3  1.9  6568 3712 ?      S    23:17   0:00 /usr/bin/gnome
motoki   948  0.1  1.1  3876 2244 ?      S    23:17   0:00 kinput2 -canna
motoki  1049  0.0  0.2  1300  444 ?      S    23:17   0:00 esd -terminate
motoki  1051  0.0  0.9  5368 1884 ?      S    23:17   0:00 gnome-smproxy
motoki  1061  0.6  1.8  5440 3512 ?      S    23:17   0:00 /usr/bin/sawfi
motoki  1064  0.0  0.8  3160 1668 ?      S    23:17   0:00 xscreensaver -
motoki  1066  0.6  2.8  8680 5404 ?      S    23:17   0:00 panel --sm-con
motoki  1068  0.5  3.3  9100 6352 ?      S    23:17   0:00 gmc --sm-confi
motoki  1070  0.5  2.2  7080 4296 ?      S    23:17   0:00 /usr/bin/gnome
motoki  1072  0.0  1.5  6228 2912 ?      S    23:17   0:00 magicdev --sm-
motoki  1075  0.0  0.7  3024 1520 ?      S    23:17   0:00 /usr/X11R6/bin
motoki  1080  0.0  0.6  2600 1240 ?      S    23:17   0:00 gnome-name-ser
motoki  1082  0.0  0.2  1096  524 ?      S    23:17   0:00 gnome-pty-help
motoki  1083  0.0  0.5  1804 1076 pts/0  S    23:17   0:00 bash
motoki  1104  0.2  2.0  7612 3916 ?      S    23:17   0:00 tasklist_apple
motoki  1106  0.1  1.8  6716 3572 ?      S    23:17   0:00 deskguide_appl
motoki  1110  0.1  1.8  6696 3564 ?      S    23:17   0:00 battery_applet
motoki  1118  5.6  4.3 10600 8448 pts/0  S    23:18   0:04 emacs sec-proc
motoki  1119  0.0  0.1  1044  360 ?      S    23:18   0:00 /usr/lib/emacs
```

```
motoki   1120  0.4  2.6 10976 5092 pts/0  S     23:18   0:00 xdvi.bin -name
motoki   1125  0.2  0.3  1860  680 pts/0  S     23:19   0:00 script immedia
motoki   1126  0.1  0.3  1864  708 pts/0  S     23:19   0:00 script immedia
motoki   1127  0.5  0.5  1804 1076 pts/1  S     23:19   0:00 bash -i
motoki   1145  0.0  0.3  2300  732 pts/1  R     23:19   0:00 ps aux
[motoki@x205b]$
```

このリスト中の主要プロセスについて簡単な説明を次に与える。

(PID=1) init ··· システムのブートプロセスを引き継ぐプロセスで、全プロセスの親(もしくは祖先)プロセスとなる。

(PID=2~ 7) kflushd, kupdate, kpiod, kswapd, mdrecoveryd, khu ··· カーネルが自動的に立ち上げるデーモン群。 kupdateはBSD系のupdateデーモン(30秒おきにsyncシステムコールを実行)に相当するもの？ kswapdはBSD系のスワッパに相当？

(PID=360) portmap ··· **Portmap is a server that converts RPC program numbers into DARPA protocol port numbers.** (man portmapより)

(PID=383) rpc.statd ··· **The rpc.statd server implements the NSM (Network Status Monitor) RPC protocol.** (man rpc.sta より)

(PID=397) /usr/sbin/apmd ··· APM(電源管理)デーモン。

(PID=441) /usr/sbin/automount ··· ファイルシステムにアクセスした時にそのファイルシステムをNFSマウントし、しばらくアクセスしなければマウントを解除するシステムのデーモン。

(PID=494) syslogd ··· システムのログファイルを書き出すデーモン。

(PID=503) klogd ··· カーネルまわりのメッセージを記録するデーモン。

(PID=517,520~ 524) identd ··· **TCP/IP IDENT protocol server.** (man identdより)

(PID=535) /usr/sbin/atd ··· **run jobs queued for later execution.** (man atdより)

(PID=549) crond ··· 指定した日時に指定したプログラムを起動するデーモン。

(PID=580) /sbin/cardmgr ···

(PID=595) usbmgr ···

(PID=610) inetd ··· インターネットスーパーサーバ。

(PID=619) sshd ··· セキュアシェル(slogin, ssh, scpコマンド)のサーバ。

(PID=635) lpd ··· プリンタデーモン。

(PID=706) /usr/lib/postfix/master ··· **Postfix** マスタープロセス。 (man masterより)

(PID=710) qmgr ··· Postfix キューマネージャ。 (man qmgrより)
(PID=722) gpm ··· a cut and paste utility and mouse server for virtual consoles. (man gpmより)
(PID=736,739~ 746) httpd ··· Webサーバ。(デーモン)
(PID=762) proftpd ··· ftpサーバ。(デーモン)
(PID=776) /usr/bin/jserver ··· Wnnのかな漢字変換サーバ。
(PID=791) xfs ··· X フォントサーバ。
(PID=801) /usr/sbin/cannaserver ··· かな漢字変換サーバ。(Canna)
(PID=823~ 828) /sbin/mingetty ··· ログインのプロンプトを表示するデーモンgetty。
(PID=862) /etc/X11/X ··· X Window Systemのサーバ

例8. 7 (プロセス間の親子関係)

プロセス間の親子関係は pstree コマンドによって図示することが出来る。

例えば 、例8.5で表示されたプロセス群の親子関係は次の様に表示される。

```
[motoki@x205b]$  pstree
init-+-apmd
     |-atd
     |-automount
     |-battery_applet
     |-cannaserver
     |-cardmgr
     |-crond
     |-deskguide_apple
     |-esd
     |-gmc
```

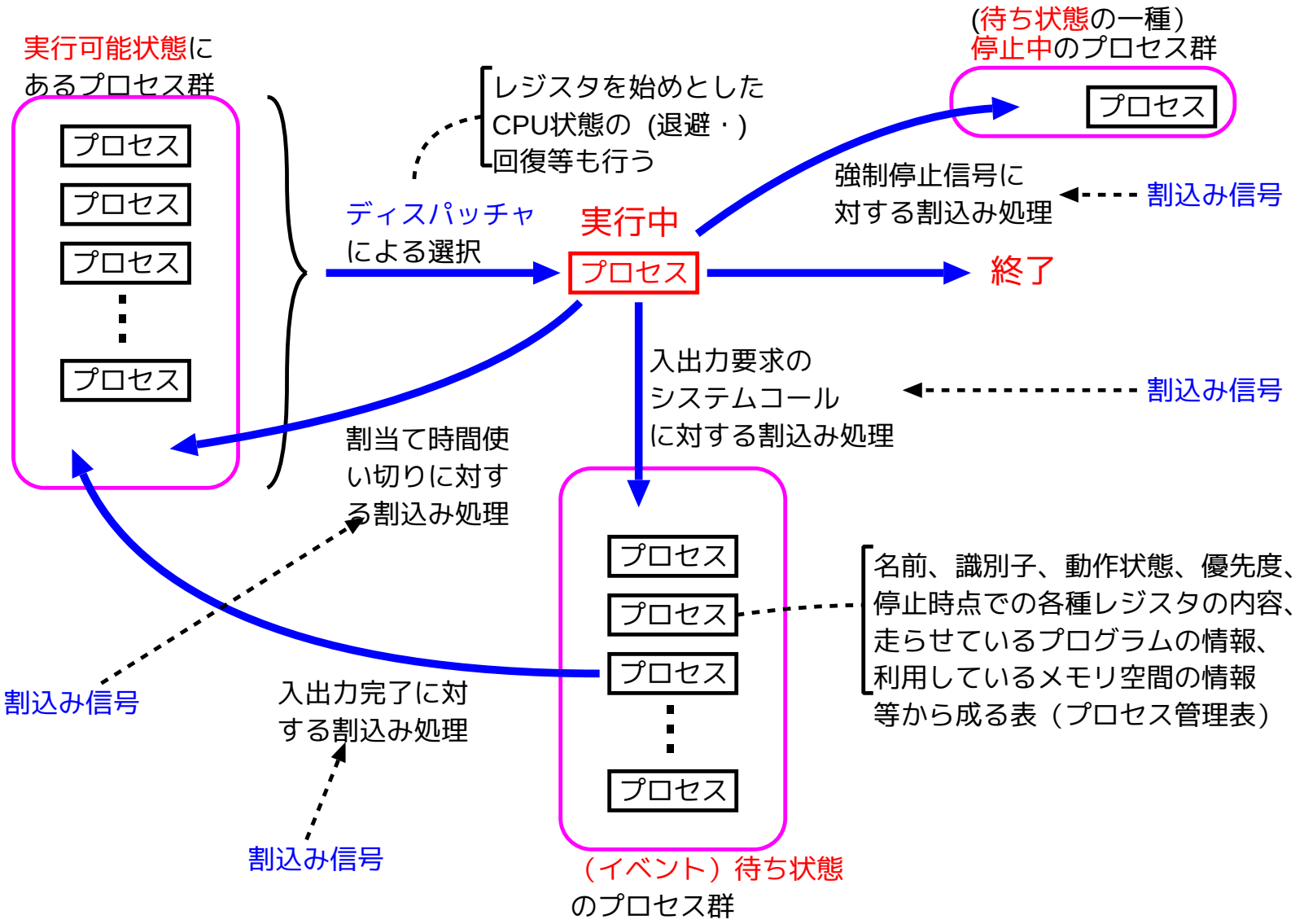
```
|-gnome-name-serv
|-gnome-smproxy
|-gnome-terminal-+-bash-+-emacs---emacsserver
|                |      |-script---script---bash---pstree
|                |      `-xdvi.bin
|                `-gnome-pty-helpe
|-gpm
|-httpd---8*[httpd]
|-identd---identd---3*[identd]
|-inetd
|-jserver
|-kflushd
|-khubd
|-kinput2
|-klogd
|-kpiod
|-kswapd
```

```
|-kupdate
|-lpd
|-magicdev
|-master-+-pickup
|        `-qmgr
|-mdrecoveryd
|-6*[mingetty]
|-panel
|-portmap
|-proftpd
|-rpc.statd
|-sawfish
|-sshd
|-syslogd
|-tasklist_applet
|-usbmgr
|-wdm.bin-+-X
```

```
      |          `-wdm.bin---gnome-session
      |-xconsole
      |-xfs
      `-xscreensaver
[motoki@x205b]$
```

8–6 プロセスの状態と状態遷移

p.70からの引用

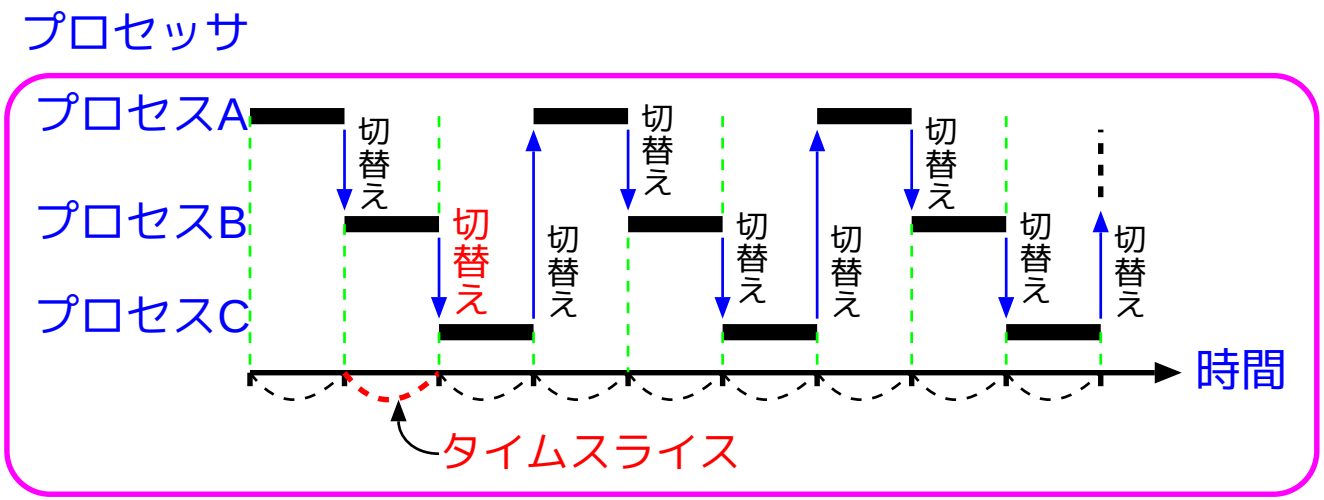


プロセスの状態：

システム内で生きているプロセスは次の3つのいずれかの状態にある。

実行中 ··· 実行可能な状態にあるプロセスの中からスケジューラ(またはディスパッチャという)が選ぶ。

- 無事終了するとプロセスは消滅する。
- 続けてCPUを使用できる時間に上限値(タイムスライスと言う)を設定することもある。



- 入出力要求のシステムコールを発すると、「入出力完了」という事象(イベント)を待つ状態に移る。
- 強制停止の割込み信号が入ると、やはりイベント待ち状態に。

実行可能 ··· CPUの割り付けを待っている状態。

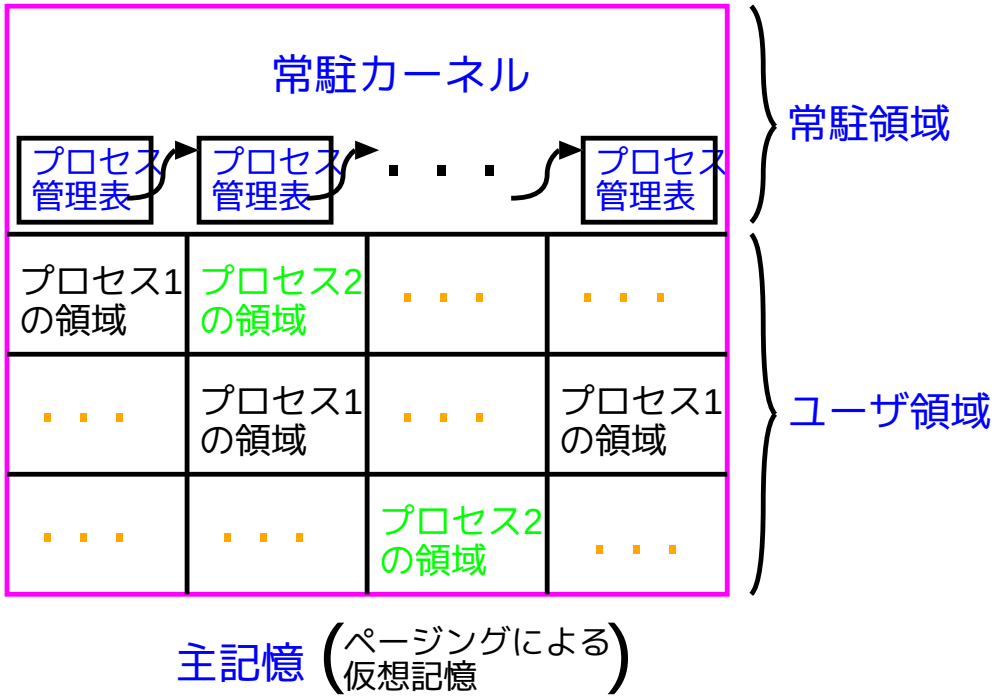
- `fork()`で生成された直後の子プロセスはこの状態になる。

イベント待ち ··· すぐには実行できず、「入出力完了」等の事象(event)を待っている状態。

- 待っている事象が起こると、OSにより実行可能状態に移される。

プロセス管理表：

プロセスの管理／**seamless**な切替えのために**OS**が必要とする情報は全てプロセス管理表(プロセス表,プロセス制御ブロック,...とも言う)に保持され、主記憶内の常駐領域に置かれる。



プロセス管理表には次の様な情報が入っている。

- 名前 ··· 人間がプロセスを識別するために付けた名前
- 識別子 ··· プロセスの識別子(**PID**), 親プロセスの識別子(**PPID**), プロセスが属するグループの識別子

- 所有者情報 ··· プロセスの所有者のID,グループID
- 動作状態 ··· プロセスの動作状態,イベント待ちの要因
- シグナル情報 ··· イベント待ちマスク
- 時間情報 ··· プロセス起動時の時間,これまでに消費したCPU時間,子プロセスの消費したCPU時間
- プロセスの優先度 ··· スケジューリングの参考にされる。
- 保護情報 ··· 他のどのプロセスに資源操作やプロセス間通信を許すかを明らかにする。
- 走行情報 ··· プロセス中断時点でのプログラムカウンタ,プログラム状態語を始めとした各種レジスタやスタックポインタの内容
- プログラム情報 ··· プロセスがどのロードモジュールを使って実行しているかを示す。
- 空間情報 ··· プロセスが利用している実メモリ空間や仮想メモリ空間に関する情報
- 資源情報 ··· ファイルや入出力装置、セマフォなどの、操作している資源を管理する表へのポインタなど

8–7 プロセススケジューリングの目標

目標： ..... コンピュータシステム全体の性能目標でもある。

(1) スループットの向上
　　... 単位時間当たりのジョブの処理件数を上げる。

(2) 応答時間の保証
　　... 端末からの処理要求に対して、あまり待たせることなく反応がある。

⇨ 特定のプロセスが資源を独占的に使用するのは避ける。

- CPUを独占するプロセスが出ない様にする。
- 入出力装置を独占するプロセスが出ない様にCPU資源を割り付ける。

補足：
機種によっては、資源割当ての方針をOSに指示できる様になっている。

8–8 プロセススケジューリングの方法

non-preemptive vs. preemptive:

- ノンプリエンプティブ・スケジューリング ...

  プロセスにCPUを割り付けた後そのプロセスが完了するか
  自らCPUを放棄するまでCPUを使用させる方法。

- プリエンプティブ・スケジューリング (横取りスケジューリング) ...

  現在実行しているプロセスの処理を中断し別のプロセスの処理を開始することがある方法。

preemptive a.
先買の, 先買権のある, ...

スケジューリングのためのデータ構造：

CPUを使用するためのプロセスの待ち行列が出来ていると考えて良い。

これがスケジューリングの基本。

—

スケジューリングの方式：

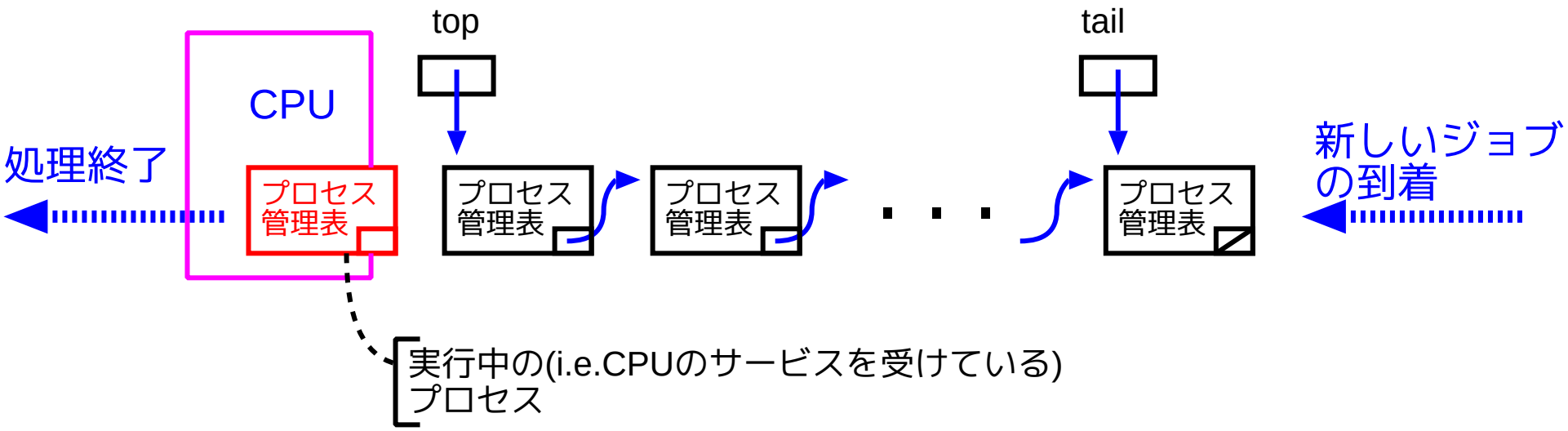
基本的な方式としては次の7つがある。
一般には、これらを組み合わせた複合方式

(1) 到着順 (FIFO,first-in-first-out, FCFS,first-come-first-served):

… プロセスをその到着の順序に従って処理する。

⇨ 典型的なノンプリエンプティブ方式。
特定のプロセスによるCPU独占の可能性。

実装：



(2) 処理時間順 (SPT, shortest-processing-time first) :

… 処理時間の短いプロセスから処理する。

⇨ 処理時間を予め知る方法は一般に存在しないので、
プリエンプティブ方式と組み合わせると有効。

実装：

自分で考えてみて下さい。

(3) 優先度順スケジューリング (priority scheduling) :

… 各プロセスに割り当てられた優先度の高い順に処理する。実行中のものより優先度の高いプロセスが到着すると、即座にこれがディスパッチされる。

⇨ 典型的なプリエンプティブ方式。

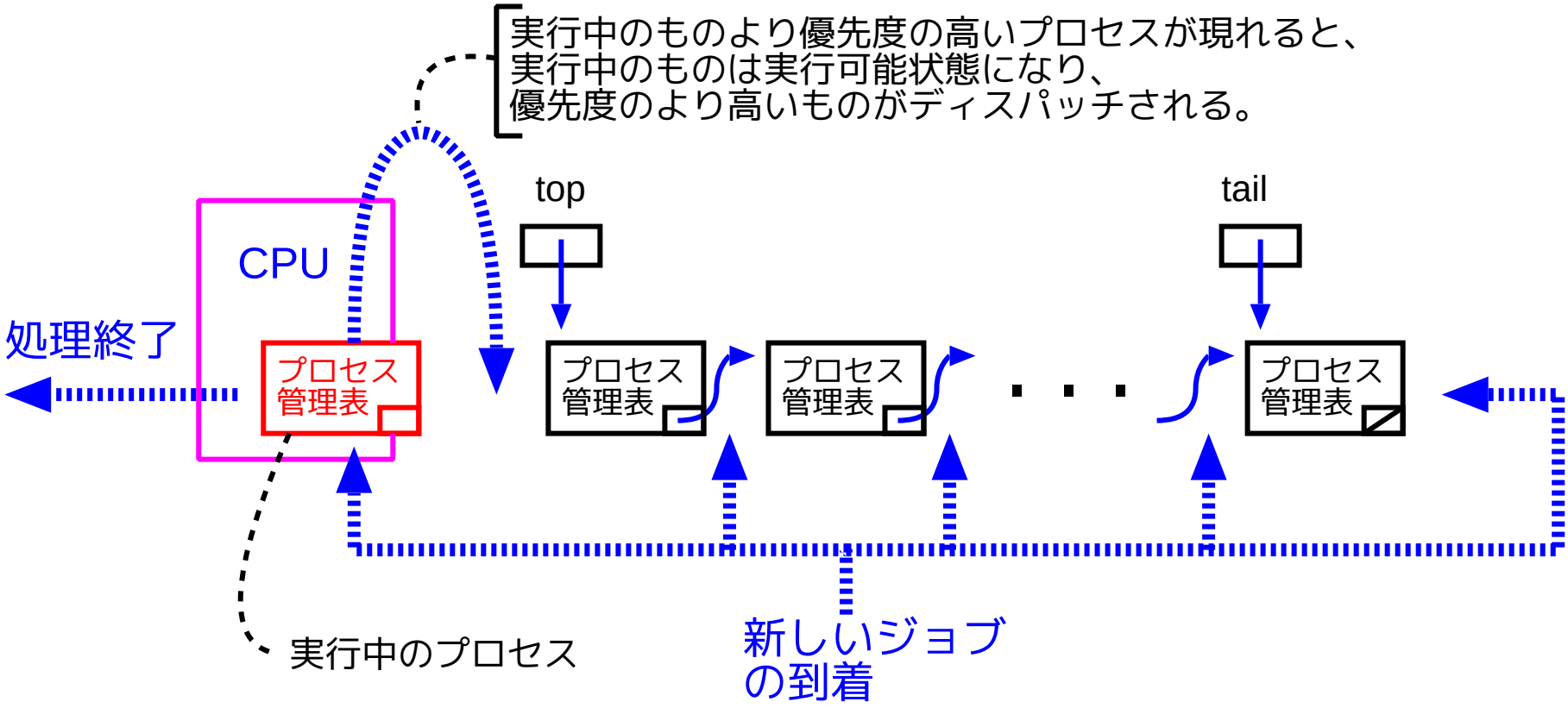
補足：

一旦指定された優先度が変化しない静的優先度方式と、実行中に優先度を変える動的優先度方式の2種類がある。

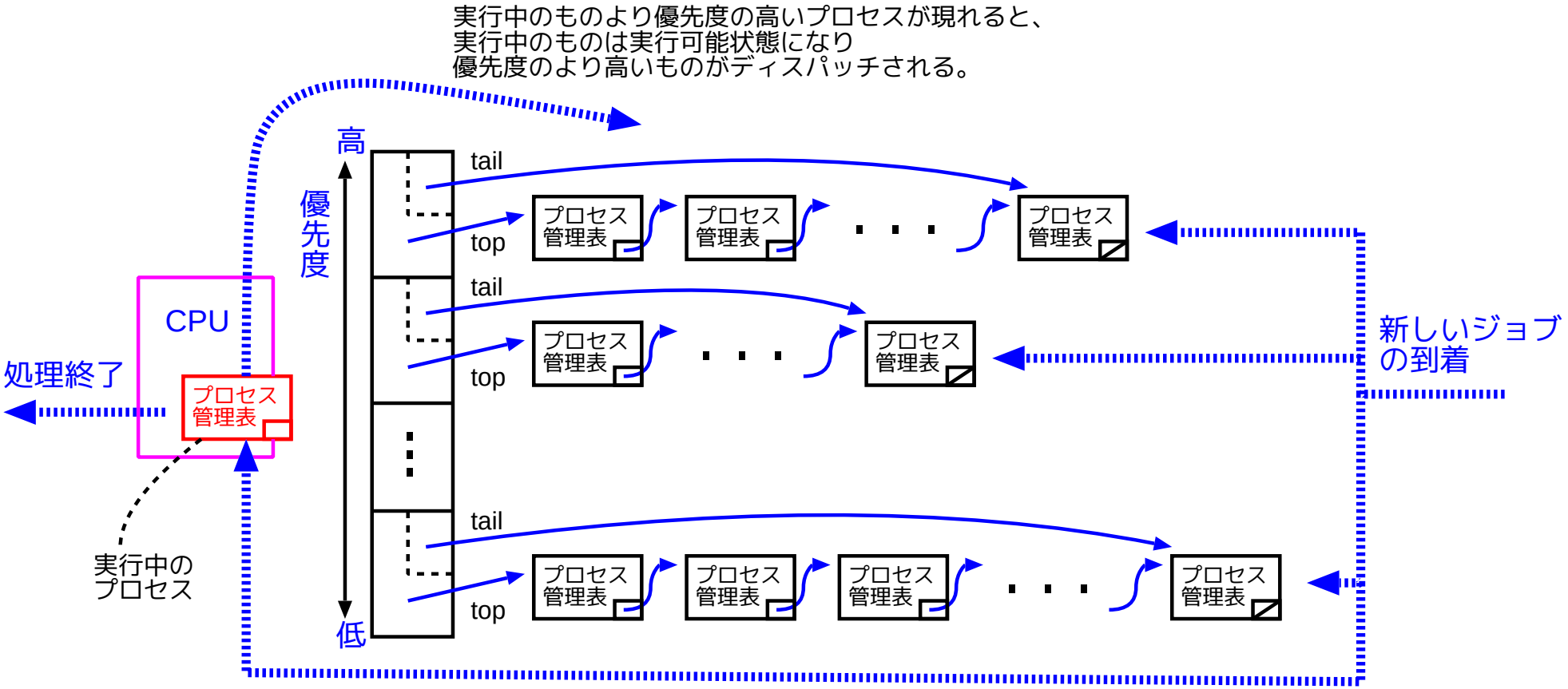
動的優先度方式の例：

優先度の低いプロセスになかなかサービスの順番が回って来ない問題を解決するために、待ち時間に比例して優先度を上げる、エージング (aging) と呼ばれる方法が取られることがある。

実装：


実行中のものより優先度の高いプロセスが現れると、
実行中のものは実行可能状態になり、
優先度のより高いものがディスパッチされる。
top
tail
CPU
処理終了
プロセス
管理表
プロセス
管理表
プロセス
管理表
プロセス
管理表
実行中のプロセス
新しいジョブ
の到着

または、

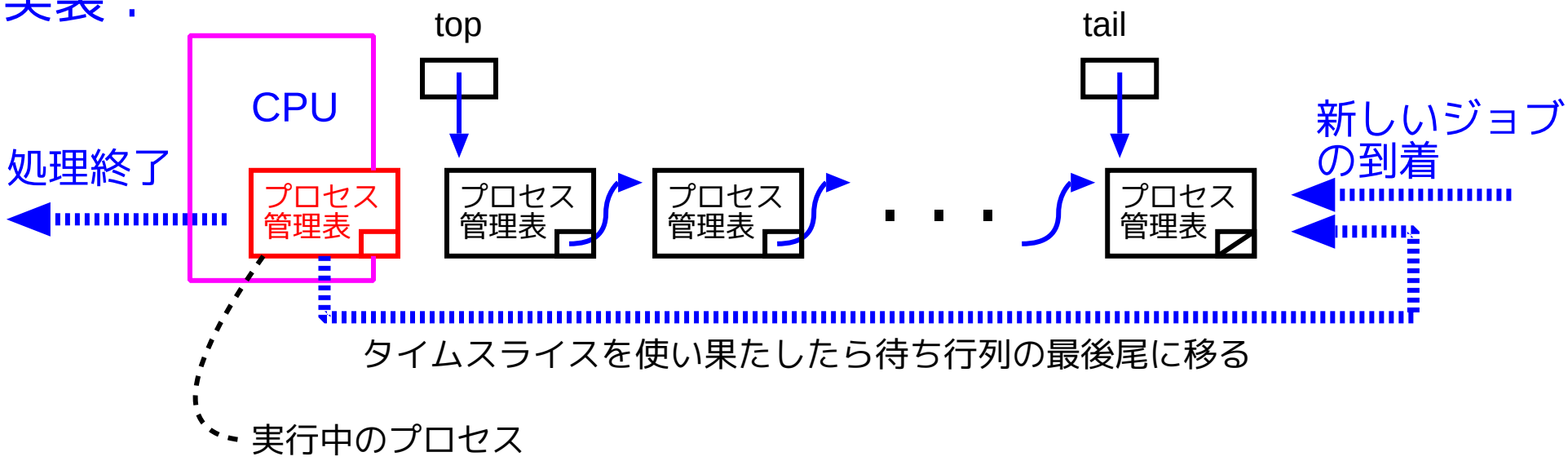

実行中のものより優先度の高いプロセスが現れると、
実行中のものは実行可能状態になり
優先度のより高いものがディスパッチされる。
高
優先度
低
tail
top
tail
top
tail
top
プロセス管理表
CPU
処理終了
実行中のプロセス
新しいジョブの到着

(4) ラウンドロビン (RR,round robin) :

… 実時間を短いタイムスライスに区切り、走行プロセスが与えられたタイムスライスを使い切る度に最も長時間待っているプロセスに切り替える。これによって、多数のプロセスを切り替えながら並行に少しずつ実行する。

⇨ 典型的なプリエンプティブ方式。
タイムスライス値をどう決めるか？

実装：



(5) 動的ディスパッチング (dynamic dispatching, heuristic dispatching) :

··· **I/O**バウンドな(i.e.入出力を多用する)プロセスを**CPU**バウンドな(i.e.入出力をあまり使わない)プロセスより優先して処理する。

補足：
この方式を用いると多重プログラミング方式の計算機システムの処理能力(スループット)が最も高くなることが、実験・経験的にも理論的にも示されている。
この方式を採用する場合は、どのプロセスが**I/O**バウンドでどれが**CPU**バウンドなのかを判断するために、最近のある一定期間の入出力の頻度を測定するのが一般的である。

実装： 自分で考えてみて下さい。

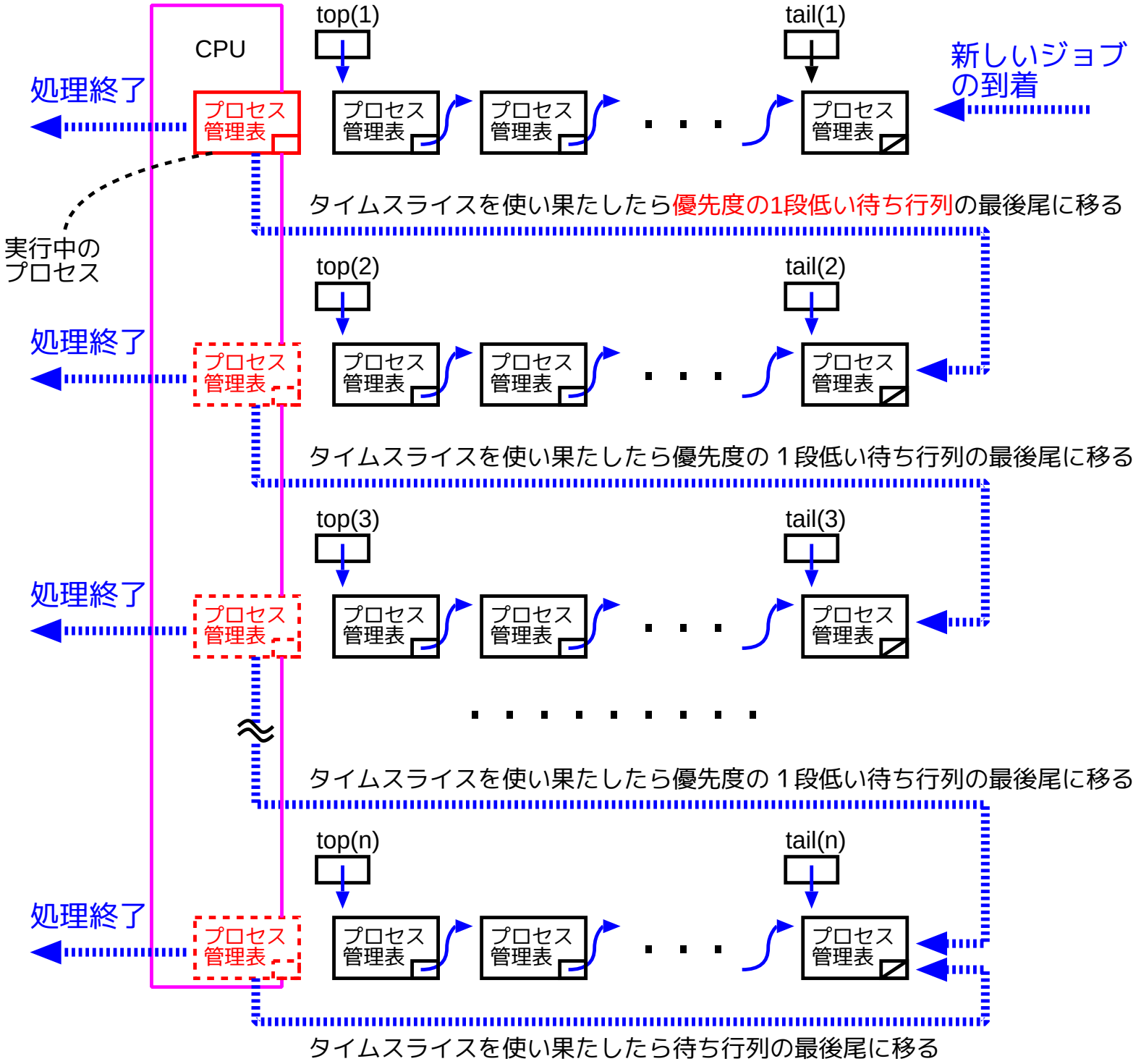
(6) フィードバック待ち行列方式 (FB,feedback queue) :

… プロセスを多段のラウンドロビン方式で処理する。

補足：
優先度の低いプロセスになかなかサービスの順番が回って来ない問題を解決するために、待ち時間に比例して優先度を上げる、エージング (aging) と呼ばれる方法が取られることがある。

実装： ⇨ 次ページ

実装：


CPU
top(1)
tail(1)
新しいジョブの到着
処理終了
プロセス管理表
実行中のプロセス
タイムスライスを使い果たしたら優先度の1段低い待ち行列の最後尾に移る
top(2)
tail(2)
タイムスライスを使い果たしたら優先度の１段低い待ち行列の最後尾に移る
top(3)
tail(3)
top(n)
tail(n)
タイムスライスを使い果たしたら待ち行列の最後尾に移る

(7) デッドラインスケジューリング (deadline scheduling) :

· · · 処理完了の目標時間が迫っている順に処理する。

実装：

自分で考えてみて下さい。