オペレーティングシステム2004
プロセス (2) および
カーネルモード・システムコール

2004年10月14日

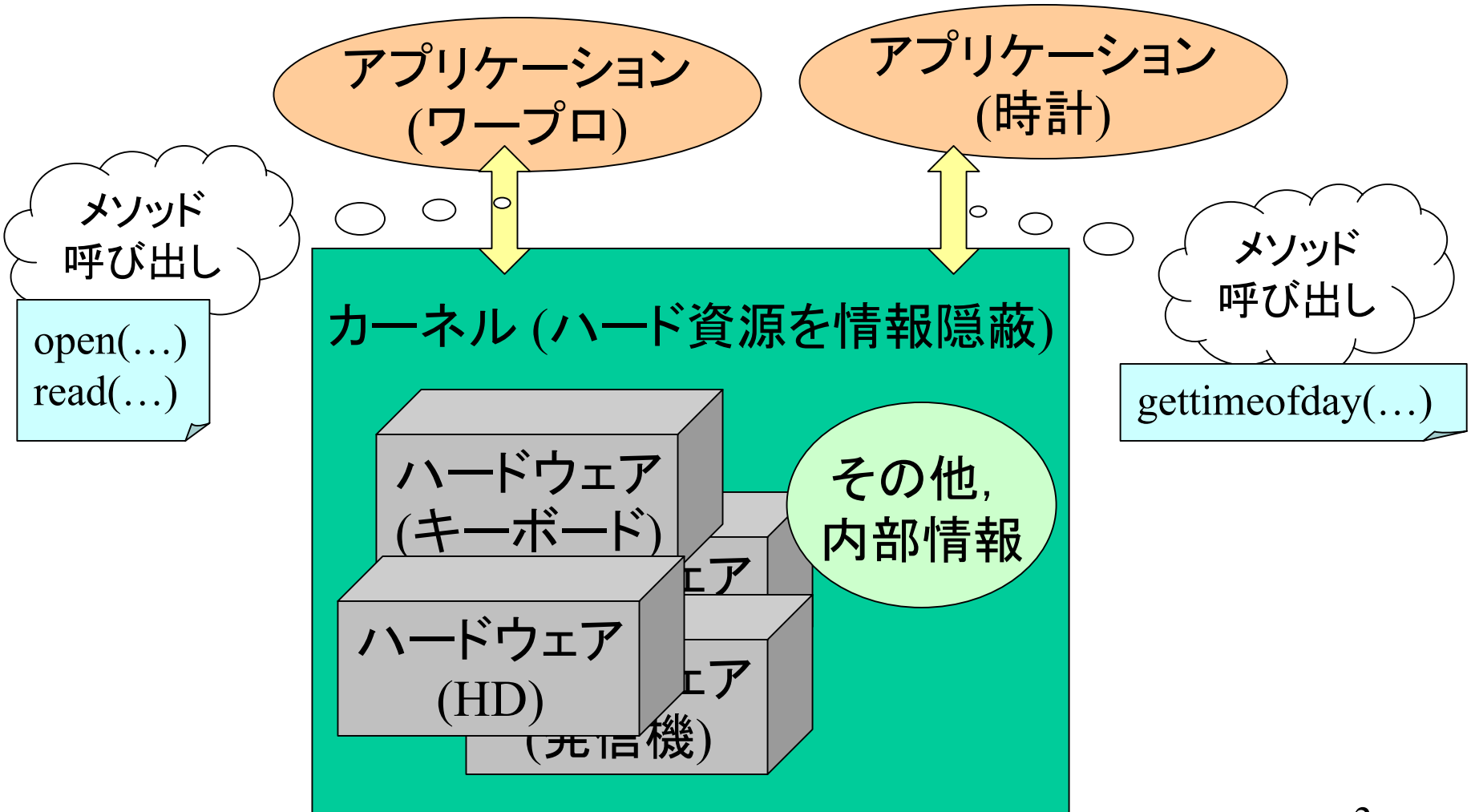
海谷 治彦

目次

- カーネルモード
- システムコール
- Linux2.4でのプロセスの実装
- fork()を使ったプログラム再び
  - 次回の演習にむけて
  - OSというより，むしろC言語のリハビリ

第1回より抜粋

オブジェクトとしてのカーネル



第1回より抜粋

カーネルの管理対象 1/2

- プロセス: プログラムのインスタンス. すなわち, 個々の実行中プログラムのこと.
  - 1つのプログラムが複数のプロセスにインスタンス化されるのが普通.
    - 例: kterm や less, bash など.
- システムリソース: プロセスが計算を進めるために必要な資源. RAM, CPU等

fork2.c の概要

```
 1| main(int argc, char* argv[]){
 2| pid_t ch; char buf[100];
 3|
 4|   while(fgets(buf, 100, stdin)!=NULL){
 5|     buf[strlen(buf)-1]='¥0';
 6|     if((ch=fork())==0){ // child
 7|       execl(buf, buf, NULL); // execveを呼ぶ
 8|     }else if(ch>0){ // parent
 9|       sleep(10);
10|       printf("done %d¥n", ch);
11|       wait(0);
12|     }
13|   }
14|
15| }
```

数々の疑問

- fork()はカーネルが管理するプロセスを生成するものだ.
- それにもかかわらず一般ユーザーのプログラム内から呼び出されている.

- ってことは，カーネル管理対象を一般ユーザーが直接いじれるってことになるのか？？
  ⇒ そりゃ，マズいだろう.

上記のような疑問を持たなかった人はむしろヤバい.

ユーザーモード・カーネルモード

- 必然性: 一般のアプリケーションが直接にハードウェアにアクセスして，OSの管理を混乱させるようなことはしたくない.
  - OSがアプリAにあるメモリ部分を割り当てたのに，アプリBが直接メモリにアクセスして，内容を上書きされたりしたら，計算が破綻しちゃう.
- よって，アプリ(一般ユーザーの処理)の処理権限を低めるのが，モード分けをする理由である.

文献5 p.10,

カーネルモード

- スーパーバイザモードとも呼ばれる.
- 計算機資源に直接アクセスすることを許す命令群を実行できる状態のこと.
- 要はなんでもできる.
- カーネルモードで実行されるプログラムは，通常，アプリに直接記述されているのではなく，
- カーネルの関数(システムコール)として記述されている.
- システムコールを介して資源にアクセスする限り，個々のプロセスは安全に並行動作すできることが保障されている. (されていなければOS じゃない！)

ユーザーモード

- 計算機資源等に関与しない普通の計算を実行されている状態.
- アプリは通常，ユーザーモードで動作する.

- 普通の計算
  - たとえばsin, cosなどの数値計算とか，
    - 正直，これは普通かどうか微妙. (FPUの利用がありうるため)
  - 文字列の長さを測る strlen()とか.

システムコール

- CPUやメモリ，ファイル，そしてプロセス等，注意深く処理しないと破綻をきたすような資源を操作する関数群.
- 具体的には read, write, そして先ほどの forkなど.
- システムコールの処理はカーネルモードで実行される.
  - そうじゃないと資源にアクセスできないし.

システムコールが必要な説明 問題提起編

fork()の場合

- ユーザーが自由に カーネル内のプロセスの内容を改訂できるとしたら,
  - 死んでないプロセスを消しちゃうかもしれない.
  - プロセスの親子関係を壊しちゃうかもしれない.

  等, 問題が出ると他のプログラムにも悪影響を与えるようなエラーが生じてしまう可能性がある.

システムコールが必要な説明 解決案編

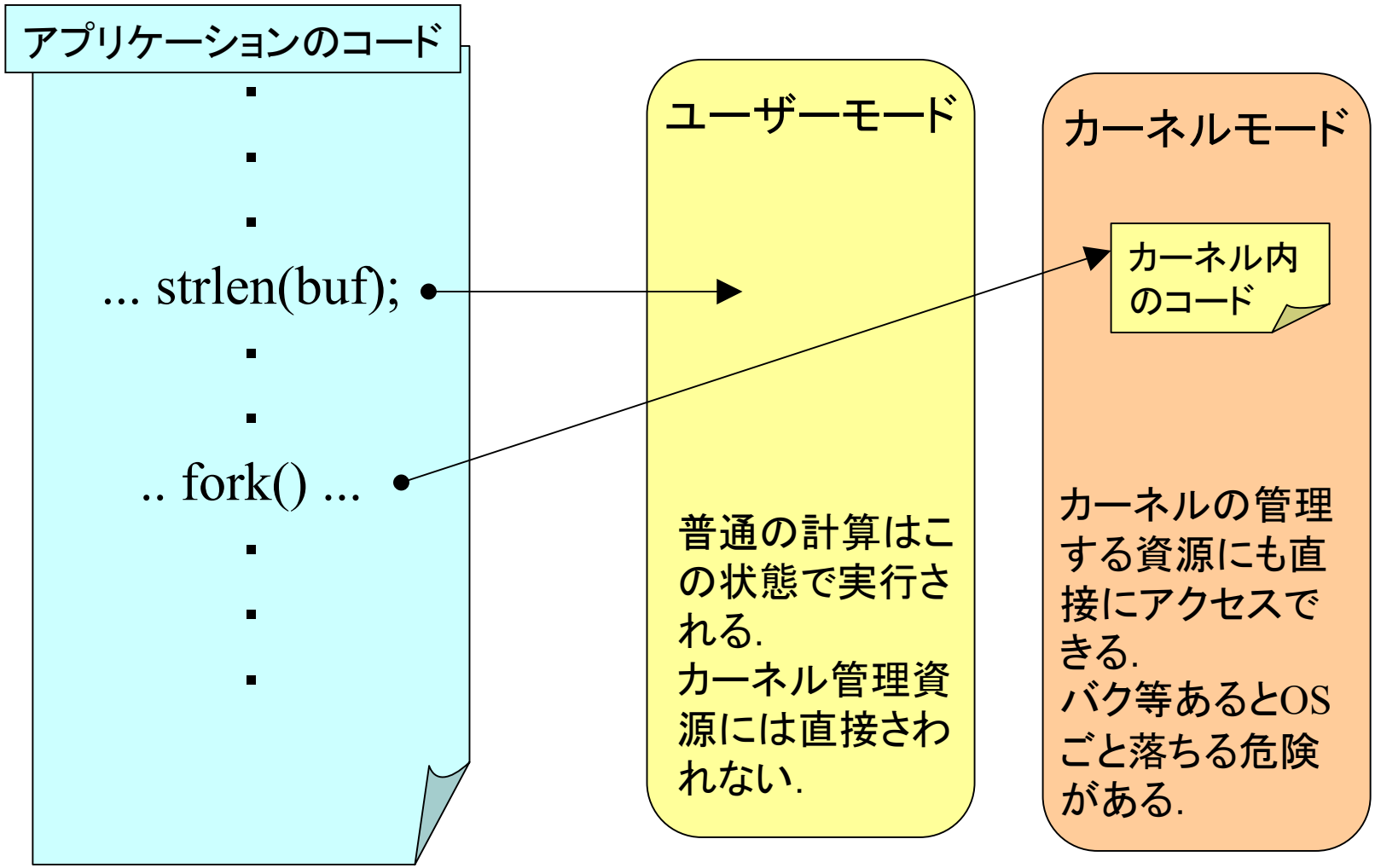
- よって，アプリがプロセスの情報に直接アクセスするのを禁止し，
- その代わり，プロセス複製のための機能を1つの関数にパッケージ化し，アプリに使ってもらうことになった.

⇒ そのような関数群がシステムコール

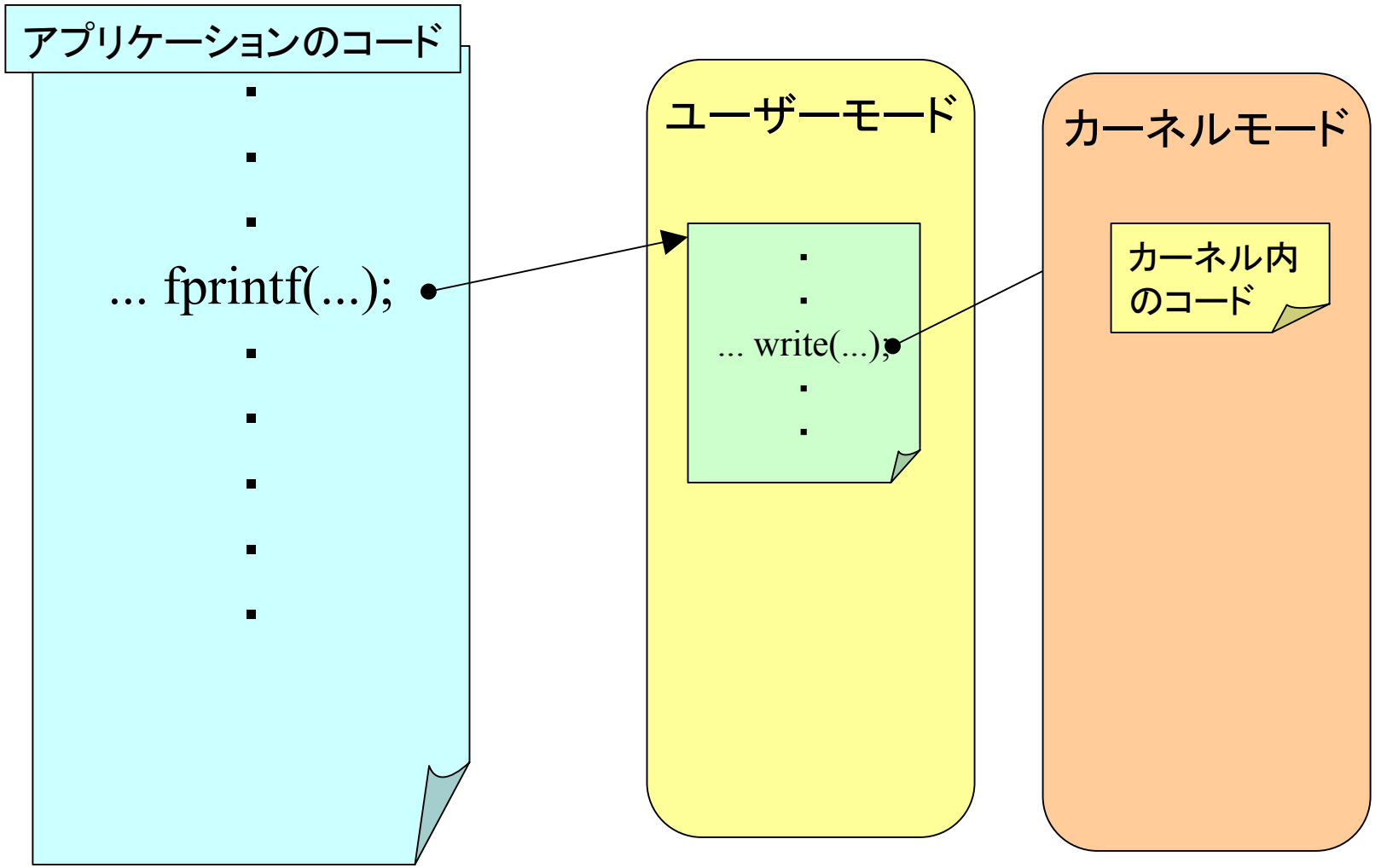
forkの他にも多数のシステムコールがある

POSIXの一部はシステムコールの標準を与える

あるプログラムの実行例



あるプログラムの実行例



資源とその利用上限

文献5 p.92

- プロセスが使う資源というのがピンと来ない人もいるかもしれないが,
- 具体的な資源は，その制限値の定義から伺い知ることができる.
- include/asm/resource.h を参照，代表的なものは,
  - CPU CPU利用時間
  - FSIZE 作成できるファイルサイズ
  - DATA 変数をおくメモリ，いわゆるヒープ
  - STACK 計算経過をおくメモリ，いわゆるスタック
  - NOFILE 利用できるファイルの数

  等 (i386アーキテクチャの場合)

プロセスの実現

- Linux2.4の場合. (2.2と結構変わった(涙))
- 単なるデータ構造とそのインスタンスではある.
  - 構造体 task_struct
  - include/linux/sched.h

構造体 task_struct

- include/linux/sched.h の中にある.
- 結構長い，130行くらい.
- 一つのプロセスに関係する情報が全て列挙されいる.

- 主たるものは次のページ

task_structの主たるメンバー

```
struct task_struct { // 無論，抜粋です include/linux/sched.h
  volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
  unsigned long flags;  /* per process flags, defined below */

  volatile long need_resched;
  long counter;
  long nice;

  struct task_struct *next_task, *prev_task;
  struct list_head run_list;

  pid_t pid; // プロセスのID
  struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;

  struct tty_struct *tty; // 対応する端末装置
  struct fs_struct *fs;    // カレントディレクトリ
  struct files_struct *files; // FDへのポインタ
  struct mm_struct *mm; // メモリーリージョンディスクリプタへのポインタ
  struct signal_struct *sig; // 受信シグナル
};
```

```
// include/linux/list.h
struct list_head {
        struct list_head *next, *prev;
};
```

プロセスは状態をもってました

```
[kaiya@linux2001 ~]% ps x
  PID TTY      STAT   TIME COMMAND
13909 ?        S      0:07 Xvnc :1 -desktop X -auth /home/s
13921 ?        S      0:00 twm
31408 pts/0    S      0:00 -csh
31796 ttyp0    S      0:00 -bin/tcsh
31943 ttyp0    T      0:00 vi a.c
31949 ttyp0    R      0:00 ps x
[kaiya@linux2001 ~]%
```

ここの部分が現在の状態を示す.

struct task_struct { // 無論，抜粋です include/linux/sched.h
volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
}

プロセスの状態

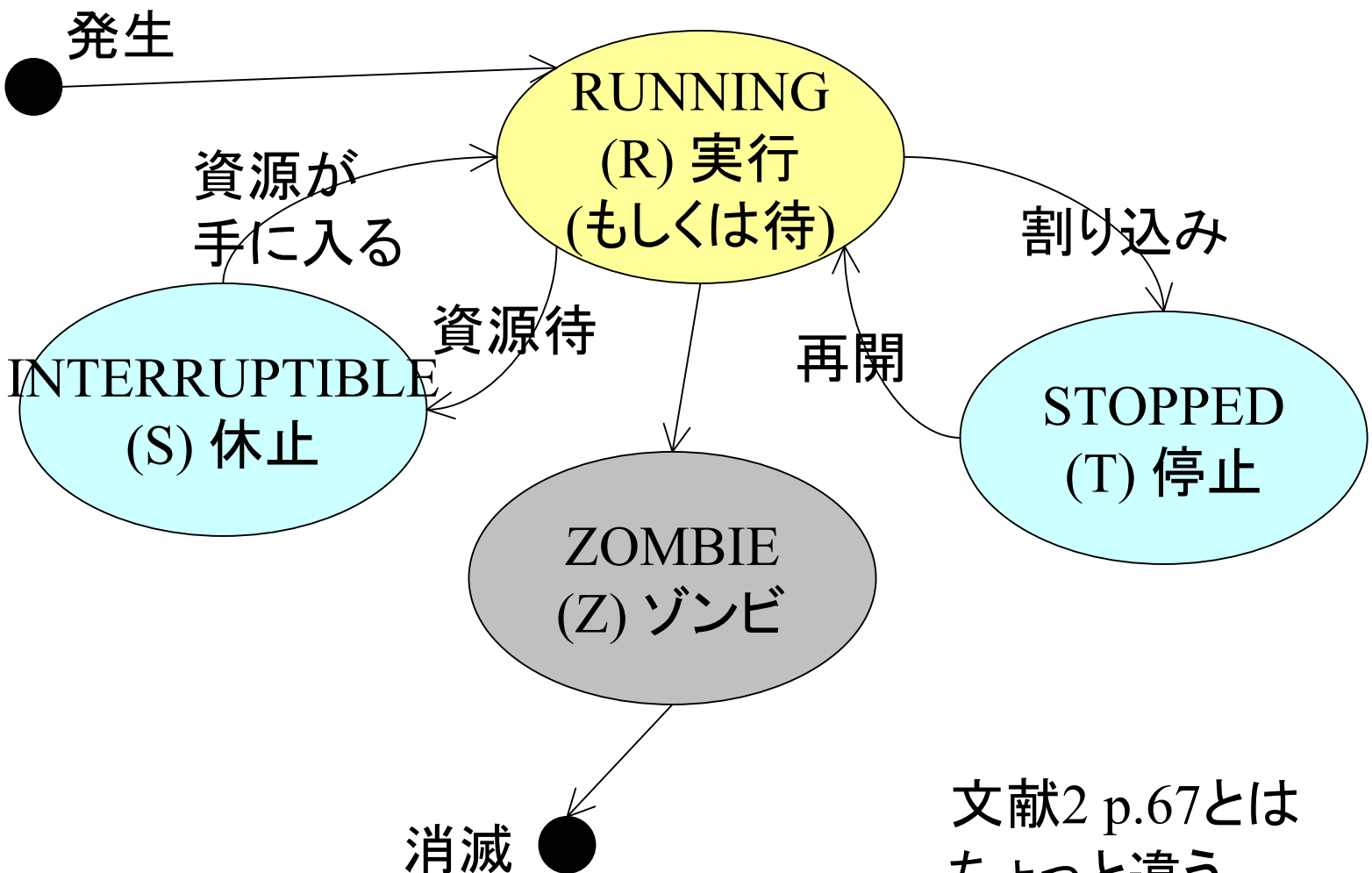
- そもそもCPUは1個程度なので，ある瞬間に実行されているプロセスも1個程度.
- 資源確保の関係等で，プロセスは常に実行状態とは限らない.
  - 例えば，diskの書き込み待ちとか.
- よって，各プロセスは状態変数 state を持ち，個々のプロセスの状態をkernelが知ることができる.
- とりうる状態は，6個くらいらしいが，主なものを次項に示す.

状態の値

- (R) TASK_RUNNING 実行中もしくは実行待ち
- (S) TASK_INTERRUPTIBLE ある条件が成り立つのを待っている状態，例えば必要な資源が空くのを待っているとか.
- (Z) TASK_ZOMBIE プロセスは終了しているが，完全に削除されていない状態.
- (T) TASK_STOPPED 外部からの割り込み等でプロセスが停止している状態

実際の値は include/linux/sched.h の85行目あたり.

1つのプロセスの状態遷移



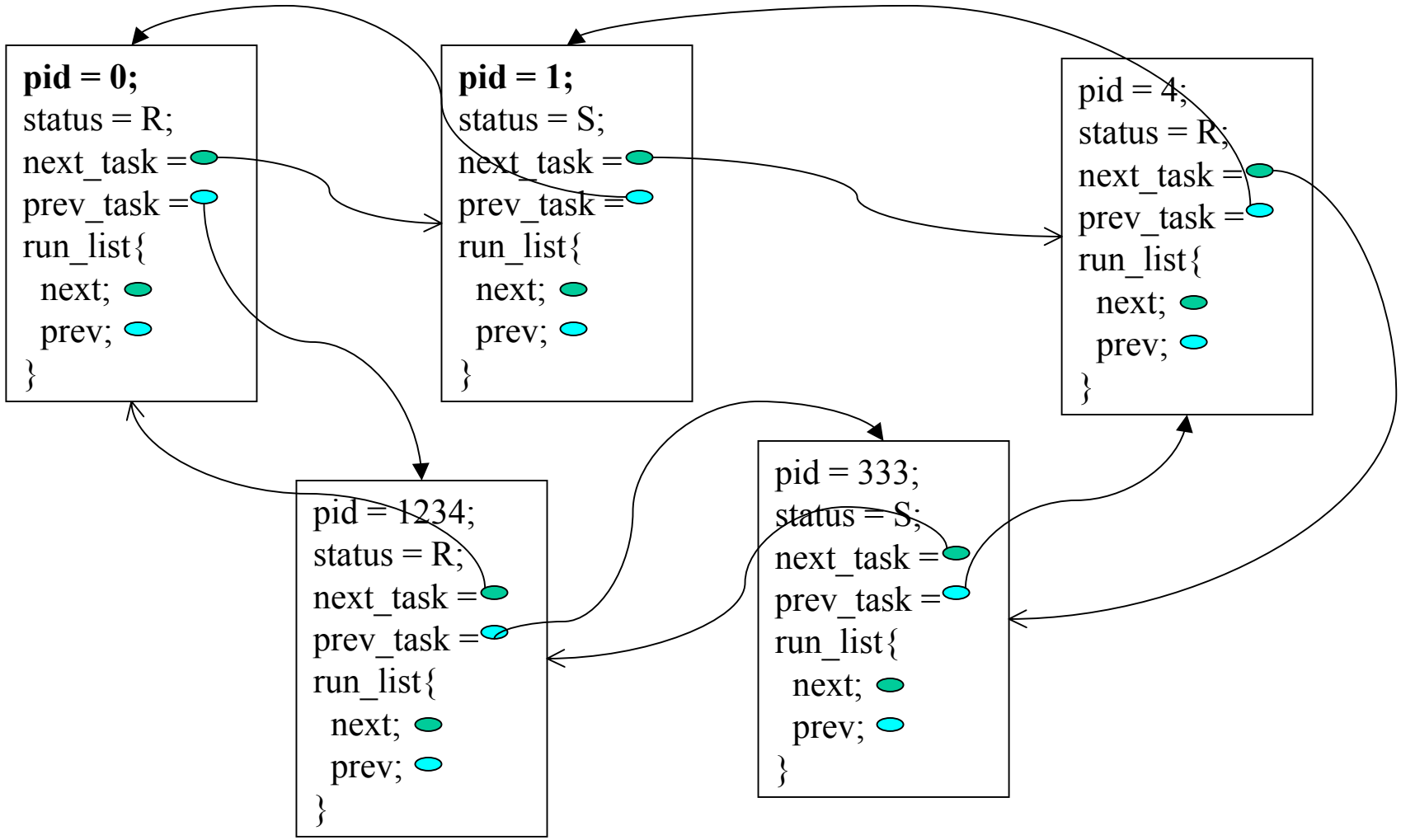
文献2 p.67とは
ちょっと違う

プロセスリスト

- 上記の名前は文献5での名前.
- 存在するプロセスの情報を保持するtask_struct構造体のインスタンスを双方向リストで結んでいる.
- リストの先頭(といっても双方向なので先頭はないが)はSwapperの情報を保持

```
struct task_struct { // 無論，抜粋です include/linux/sched.h
  volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
  struct task_struct *next_task, *prev_task;
};
```

双方向リストの例



文献5 p.88改

現在実行中のプロセスの識別

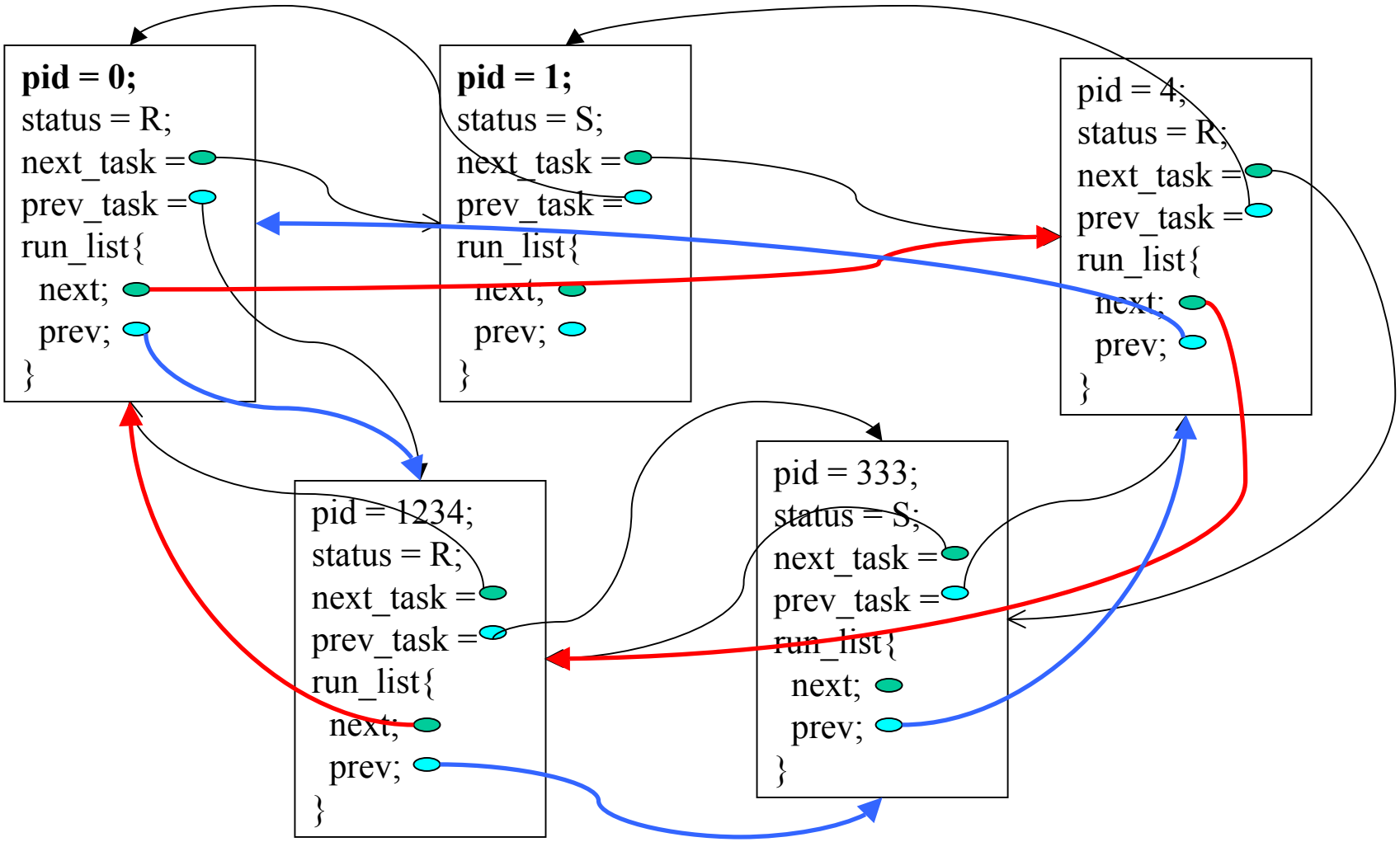
- run_listを使った双方向リストで識別する.

```
struct task_struct { // 無論，抜粋です include/linux/sched.h
  volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */

  struct list_head run_list;
};
```

```
// include/linux/list.h
struct list_head {
        struct list_head *next, *prev;
};
```

実行中のプロセスの例



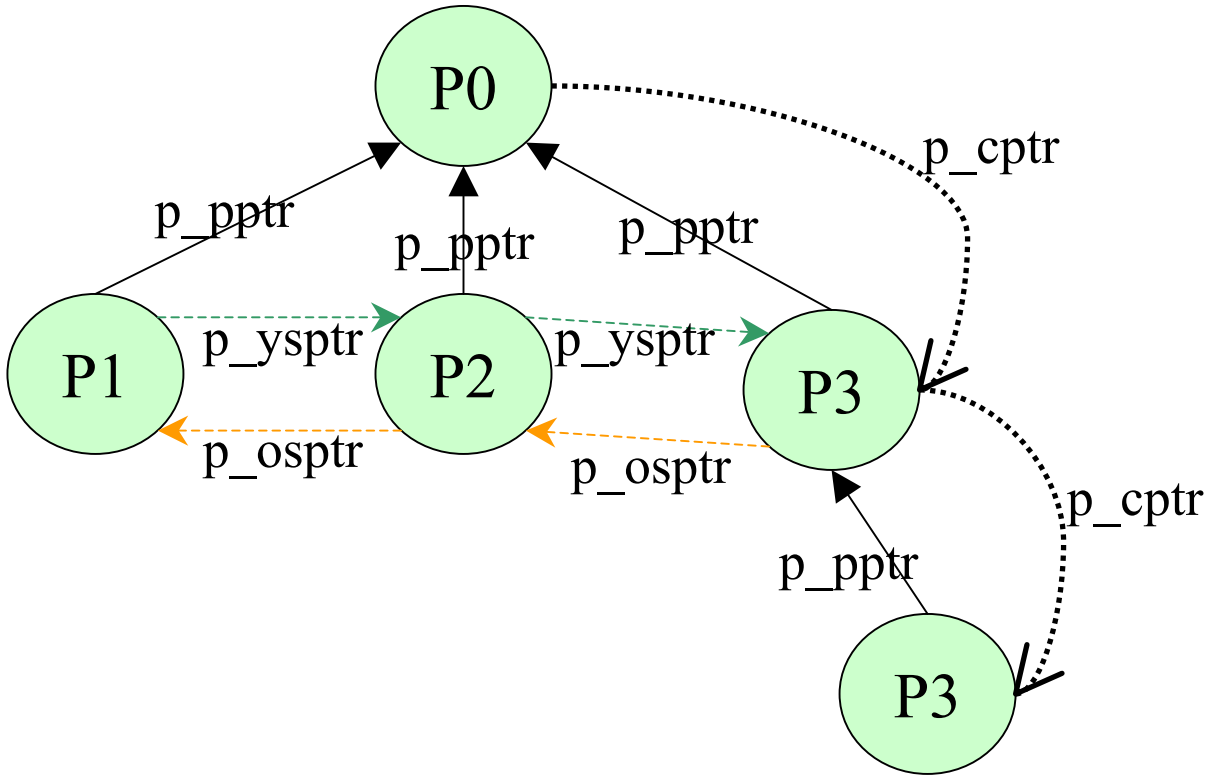
文献5 p.88改

プロセスの親子関係他

- プロセスは親の複製として生成されることを既に述べた.
- 「親」といっても実際は二種類の親がある.
  - p_opptr 生みの親. 生成もとのプロセス. もし親が先に消滅した場合, プロセス1を新しい親とする.
  - p_pptr 親. 子プロセス終了の通知を受け取るプロセス. p_opptrと同じ場合がほとんどだが, 異なる場合もある.

```
struct task_struct { // 無論, 抜粋です include/linux/sched.h
  pid_t pid; // プロセスのID
  struct task_struct *p_opptr, *p_pptr,
                     *p_cptr,  // 子. このプロセスが最後に生成したプロセス
                     *p_ysptr, // 若い兄弟. Pの直後に親に生成されたプロセス
                     *p_osptr; // 年長の兄弟. Pの直前に親に生成されたプロセス
};
```

親子関係の図示例


P0
p_cptr
p_pptr
p_pptr
p_pptr
P1
p_ysptr
P2
p_ysptr
P3
p_osptr
p_osptr
p_cptr
p_pptr
P3

fork()を使うプログラム再び

- 明日の演習1も視野にいれて，shellのプログラム周辺のプログラミング技術を学ぶ.
- shell
  - コマンドインタプリターとも呼ばれる.
  - 実行可能プログラム(コマンド)のファイルを指定すると，そのプログラムを実行させるプログラム.
  - csh (/bin/tcsh)やbash(/bin/bash)が代表的な例.

また第2回より抜粋

fork2.c の概要 shellの枠組プログラム

```
 1| main(int argc, char* argv[]){
 2| pid_t ch; char buf[100];
 3|
 4|   while(fgets(buf, 100, stdin)!=NULL){
 5|     buf[strlen(buf)-1]='¥0';
 6|     if((ch=fork())==0){ // child
 7|       execl(buf, buf, NULL); // execveを呼ぶ
 8|     }else if(ch>0){ // parent
 9|       sleep(10);
10|       printf("done %d¥n", ch);
11|       wait(0);
12|     }
13|   }
14|
15| }
```

前述プログラムの問題点

- コマンドを原則完全パス名(フルパス)で指定しないといけない.
- 引数を渡すことができない.
- 子の終了状態を得られない.

環境変数

- アプリケーション固有もしくは共有のデータ.
  例
  - PATH 実行ファイルのサーチディレクトリのリスト
  - TZ タイムゾーン，日本はGMT+9
    - 日本では Japan グリニッジではGMT等を使う.
    - /usr/share/zoneinfo/ 内に使える情報がある.
  - LANG アプリで利用される言語.
    - 日本語では ja_JP.eucJP 英語ではC等が一般的.
    - /usr/share/locale/locale.alias ファイルに主たる値の例がある.
- 名前と値の対からなる.
- 個々のプロセスが保持することができる.
- どの変数を何に使うかは基本的にはアプリケーション依存だが，ある程度使い道が決まっているものもある.

mainの第三引数 envp

- 処理系によってはmain関数の第三引数として，そのプログラムのインスタンス(プロセス)に設定される環境変数とその値のリストを得られる.

サンプルプログラムと結果

```
#include <stdio.h>

main(int argc, char* argv[], char* envp[]){
char** ptr;
	for(ptr=envp; *ptr!=NULL; ptr++){
		printf("<%s>¥n", *ptr);
	}
}
```

```
<USER=kaiya>
<LOGNAME=kaiya>
<HOME=/home/kaiya>
<PATH=/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin>
<SHELL=/bin/tcsh>
<HOSTTYPE=i386-linux>
<VENDOR=intel>
<OSTYPE=linux>
<MACHTYPE=i386>
<TZ=Japan>
<LANG=ja_JP.eucJP>
```

execファミリの関数

- execl, execlp, execv, execvp等がある.
- 基本的にプロセスの中身を書き換えるシステムコール execveのフロントエンドである.
  - フロントエンド ～ 引数等を使いやすくしたもの.

<table>
<tr><th>関数名</th><th>パスの検索</th><th>コマンドライン引数</th><th>環境配列引数</th></tr>
<tr><td>execl</td><td>なし</td><td rowspan="3">列挙</td><td>なし</td></tr>
<tr><td>execlp</td><td>あり</td><td>なし</td></tr>
<tr><td>execle</td><td>なし</td><td>あり</td></tr>
<tr><td>execv</td><td>なし</td><td rowspan="3">配列</td><td>なし</td></tr>
<tr><td>execvp</td><td>あり</td><td>なし</td></tr>
<tr><td>exece</td><td>なし</td><td>あり</td></tr>
</table>

execvの仕様

- 関数名 execve
- 返り値 int 成功する場合，返らない．失敗すると–1が返る.
- 引数 3つ
  - const char *filename プログラムの完全パス名
  - char *const argv[] コマンド名も含めた引数のリスト.
  - char *const envp[] 環境変数名と値の対のリスト，main の引数と同じ.
  - 尚，argv, envpはNULLで終わっている必要がある.
- 機能: 呼び出したプロセスを引数で指定したプログラムのプロセスに書き換える.

使用例

```
#include <stdio.h>
#include <unistd.h>

main(){
  char* filename="/bin/ls";
  char* argv[4]={
    "ls", "-l", "/", NULL
  };
  char* envp[3]={
     "PATH=/sbin:/usr/local/bin", "LANG=ja_JP", NULL
  };

  execve(filename, argv, envp);
}
```

execlp

- 関数名 execlp 返り値 execveに同じ.
- 外部変数 char **environ から環境変数を取得.
- 第1引数 const char* file コマンドのファイル名. / ではじまらない場合, 前述のenvironをもとにファイルを検索する.
- 第2引数 const char* arg1 実行時のコマンド名, 通常 上記のfileと同じものを指定.
- 第3引数以降 引数を文字列で指定する.
- 最終引数 NULL でなければならない.
- 機能: 指定されたコマンド名に現プロセスを書き換える.

例

```
#include <stdio.h>
#include <unistd.h>

extern char** environ;

main(){
  char* envp[3]={
    // "PATH=/sbin:/usr/local/bin", "LANG=german", NULL
    "PATH=/bin:/usr/local/bin", "LANG=german", NULL
  };
  int ret;

  environ = envp;

  ret=execlp("ls", "ls", "-l", NULL);
  printf("fail %d¥n", ret);
}
```

wait, exitと終了状態

- pid_t wait(int *status)
- 親プロセスが子プロセスの終了を待つのに使う関数.
- 同時に終了した子を完全に消去する処理も行う.
  - ゾンビを消去する.
- 親が子から終了通知を受けると，値が返る.
  - 返り値は終了したプロセスのID
  - 引数には子の終了ステータスが整数値がセットされる.
- 終了ステータスの値
  - 慣習上，実行が成功した場合はゼロ，
  - 失敗した場合はゼロ以外が帰るようにアプリケーションは作成される.

サンプルコード

```
main(int argc, char* argv[]){
pid_t ch;
char buf[100];

  while(fgets(buf, 100, stdin)!=NULL){
    buf[strlen(buf)-1]='¥0';
    if((ch=fork())==0){ // child
      execl(buf, buf, NULL);
    }else if(ch>0){ // parent
    int stat;
      ch=wait(&stat);
      printf("done %d, status = %d¥n", ch, stat);
    }else{ // fail
      fprintf(stderr, "fork fail");
      exit(1);
    }
  }
}
```

その他，文字列処理を思い出してね

- strchr() 文字列中の文字を探す．昔はindex()が良く使われた.
- malloc(), calloc() 引数を構成するのに必要かもしれません.
- free() 動的に確保した値はGC(Garbage Collection)しないといけません.

以下は雑多な用語の解説

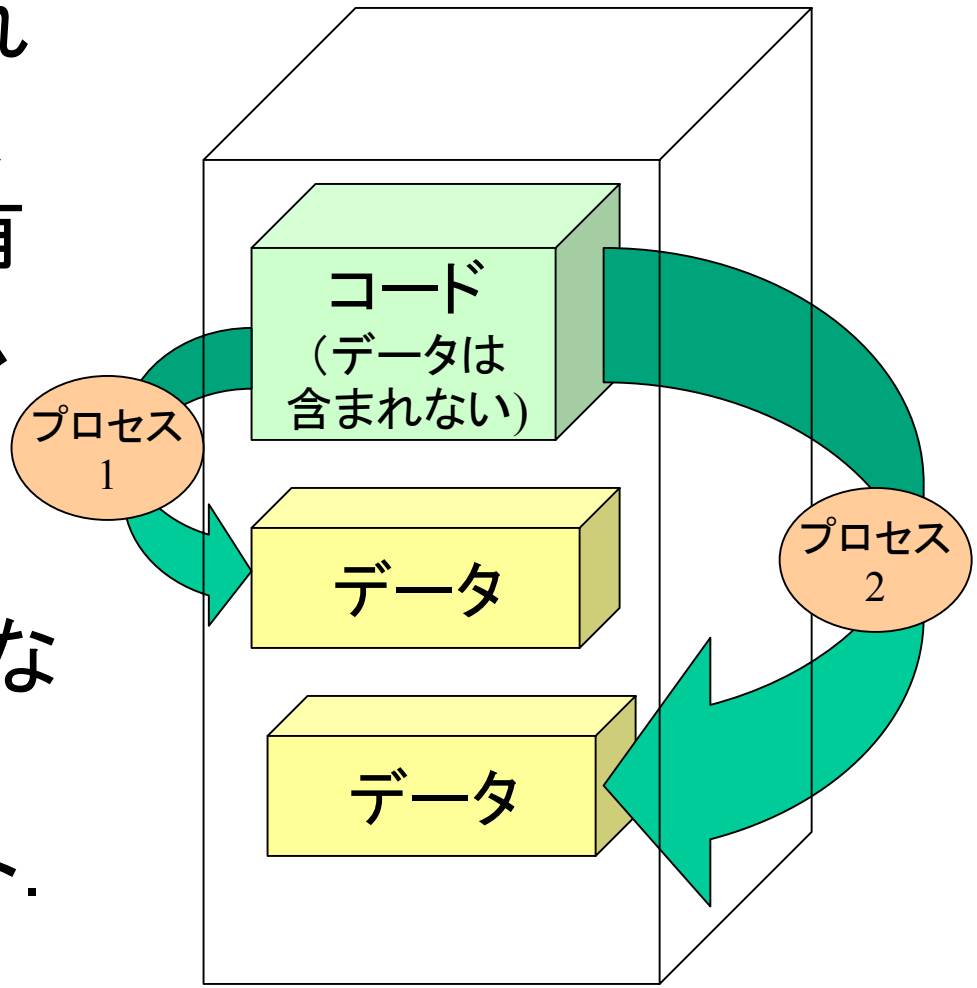
アンケートで目についた用語

- ゾンビ，defunct
- システムコール
- shell, bash, tcsh
- プロセスとスレッドの違い
- Swapper
- clone
- exec系の関数
- httpd
- リソース
- リエントラント
- フロントエンド
- 環境変数
- wait
- ライブラリ関数，システムコール
- init

文献2 p.82, 文献5 p.26

リエントラント (再入可能)

- メモリにロードされた時点でも，複数のプロセスが共有可能なプログラムの性質.
- コード側にデータ(static変数のようなもの)がなければ，普通リエントラント.



80386

- インテル社のCPUで，現在広く使われているペンティアム等の直系の祖先となる.
- 現在のインテル系CPUの基礎的技術が確立されたCPU.
- i386とかx86とか80x86とかIA32とかいう略称は，すべて80386とその子孫(ペンティアム等)を指す.

今日はおしまい