目次

★1　オペレーティングシステムの概要（つづき）

★1.1　OS のためのハードウェア機能

プロセッサ (CPU) には，OS の動作を助けるための機能が備わっている．それらの機能について述べる．

★1.1.1　実行モード

OS は，資源を管理しユーザのプログラムの実行を制御する．このような作業は OS だけが行えるようにしておかねばならない．他のプログラムが間違ってまたは故意にそのような作業を行うと，コンピュータシステムに深刻な障害を引き起こしかねないからである．

そういう事態が生じないようシステムを保護するために，プロセッサには複数の**実行モード（CPU モード）**が用意されている．3 つ以上のモードを切り替えられるプロセッサもあるが，基本的には次の 2 つのモードがある．

特権モード　カーネルはこのモードで実行される．通常，そのプロセッサに用意された全ての命令を実行することができる．プロセスの制御や入出力の制御等のための命令の一部は，このモードでしか実行できない．

特権モード: **カーネルモード**, **スーパバイザモード**とも言う．

非特権モード　一般の応用プログラムはこのモードで実行される．

非特権モード: **ユーザモード**とも言う．

応用プログラムは非特権モードで動くので,「特権的な」命令を実行することはハードウェア的にできないようになっている．

★1.1.2　割り込み

プログラムの実行途中には，入出力動作が完了した，演算エラーが発生した，等の理由で，実行中のプログラムを中断して OS や他のプログラムの実行を割り込ませなければならないことがある．これを**割り込み**という．プロセッサには，割り込みの発生を検知して適切な処理を行う仕組みが備わっている．

割り込み: interruption

割り込みは，その発生要因が実行中のプログラム内部にあるのか外部にあるのかによって，**内部割り込み** (☆ 1) と**外部割り込み**に分類される．

☆ 1) 内部割り込みは，**例外** (exception または trap) と言うこともある．

内部割り込み ゼロ除算やオーバーフロー等の演算例外が発生した，仮想記憶でページフォールトが発生した，カーネル呼び出しを行った（システムコール），等の理由による割り込み．

外部割り込み 入出力装置の動作が完了した，タイマにセットされていた時間が経過した，等の理由による割り込み．プログラムを実行中にコントロールキーを押しながら C を押すと実行を中止できる場合があるが，これもその一種である．

割り込みが発生した場合，その処理の流れは通常は次の通りである:(1) 割り込みが終わった後で元のプログラムの実行に復帰できるように，プロセッサがレジスタ値等をスタック等に退避させる．(2) カーネルの割り込み制御部（割り込みハンドラ）が割り込みの要因に対応した処理を行う．(3) 元のプログラムの実行に復帰する．

このように，割り込みに対応することもカーネルの仕事の一つである (☆ 7)．マルチプログラミングにおけるプログラムの切り替えも，割り込みを利用して行っている．

☆ 2) 割り込みの種類によっては，プログラム側で制御して，割り込み要因となる事象が発生しても割り込みの動作を抑止することができる（割り込みをマスクすると言う）．カーネルが割り込み処理を行っている最中にさらに割り込みが発生することをさけるため，多くの場合カーネルは割り込み禁止状態で実行される．

★2 プロセスの管理とスケジューリング

★2.1 プロセスとは

プロセス (☆ 3) とは，実行中のプログラムのことである．前回説明したように，プロセスは生成されてから消滅するまでずっと CPU を占有して動作するわけではなく，様々な理由で実行が中断される．中断したプロセスの実行を再開させたり，複数のプロセスの状態を把握して適切にスケジューリングしたりするために，OS は次のような情報を必要とする．

☆ 3) プロセス: process．**タスク** (task) と呼ぶこともある．UNIX 系 OS では，ps コマンドや top コマンドでプロセスを一覧することができる．

プロセス識別子（プロセス ID） OS がプロセスの管理に用いる番号．プロセス生成時に，存在する他のプロセスの ID と重複しないように割り当てられる．

プロセッサの状態 各種レジスタ (☆ 4) の値など．プロセスの実行を中断する際には，あとで再開できるようにこれらの値をどこかに退避させておく必要がある（後述の「コンテキストスイッチング」参照）．

☆ 4) プログラムレジスタ，フラグレジスタ，汎用レジスタ，etc.

スケジューリングに関する情報 現在のプロセスの状態（後述）や優先度など．

資源利用に関する情報 そのプロセスがどのような資源を使っているかなどの情報．これには，そのプロセス自体がメモリのどこに割り付けられているか，どのファイルを開いているか，といった情報も含まれる．

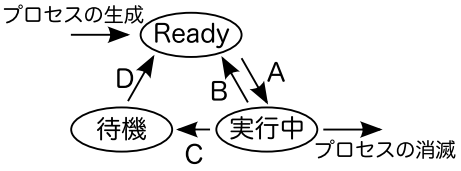
これらの情報は，プロセス毎にまとめて**プロセス記述子** (☆ 5) と呼ばれるデータ構造に格納される．OS は，このプロセス記述子を操作することでプロセスを管理する．

☆ 5) プロセス記述子: process descriptor，**プロセスコントロールブロック** (process control block; PCB) や**タスクコントロールブロック** (TCB) と呼ばれることもある．

★ 2.2 プロセスの状態とその管理

プロセスは一般に次の 3 つの状態をもっている．

- レディ(Ready): CPU を割り当ててもらえるのを待っており，いつでも実行可能な状態
- 実行中: CPU を割り当てられて実行中の状態
- 待機: 何らかのイベント（例えば入出力の完了）を待っており，実行できない状態



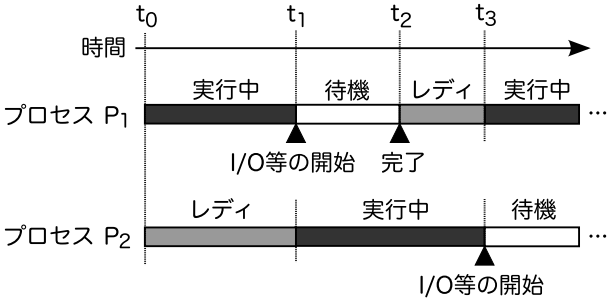
プロセスは，**スケジューラ** (☆ 6) の制御に従って状態を遷移させる．CPU が 1 つのコンピュータでは，複数のプロセスが同時に実行中になることはできない．図のそれぞれの状態遷移は次のような時に起こる．

A スケジューラがそのプロセスを選択した．実行開始または再開．
B スケジューラが他のプロセスを選択した．
C 実行中のプロセスが入出力等のためにカーネルを呼び出した．その動作が完了するまで実行できないので待機状態に遷移する．
D 待機状態を解除可能になった（入出力動作が完了した，など）(☆ 7)．

☆ 6) スケジューラ: 次節参照．

☆ 7) 例えば入出力動作の完了は割り込みによって通知される．前節参照．

下図にプロセスの実行の様子の例を示す．



時刻 t_0: プロセス P_1 が実行中．プロセス P_2 がレディ（実行可能な）状態で待っている．
時刻 t_1: P_1 が入出力等のために待機状態へ遷移．プロセスの切り替えが発生し，P_2 が実行開始される．
時刻 t_2: P_1 の入出力等の処理が完了したのでレディ状態へ遷移．P_2 が実行中のため，P_1 はそのまま待つ (☆ 8)．
時刻 t_3: P_2 が入出力等のために待機状態へ遷移．プロセスの切り替えが発生し，P_1 の実行が再開される．

☆ 8) 後述のように，スケジューリング方策によってはこの時点で P_1 が CPU を横取りする場合もある．

プロセスを切り替える（それまで実行していたプロセスをレディ状態に遷移させて別のプロセスを実行する）際には，中断したプロセスの実行を後で再開できるように，そのプロセスの状態をプロセス記述子に保存しておく必要がある．このような操作のことを，**コンテクストスイッチング** (☆ 9) という．これにはかなりの計算コストを要するため，頻繁にスイッチングを起こしてシステムの性能を低下させないように OS はうまくプロセスをスケジューリングする必要がある．

☆ 9) コンテクストスイッチング: context switching. context とは「文脈」のこと．いまどきのコンピュータの場合，CPU に専用の機構を備えてハードウェアで処理することで，できるだけすばやくスイッチングできるようにしている．

★2.3 おまけ

今日の講義と関連するプログラム例等をいくつか示す．いずれも UNIX の機能を利用するものなので，他の OS ではそのままでは動かないだろう．

● 自分のプロセス ID を表示するプログラム 以下のプログラムを実行中に ps コマンドや top コマンドを実行してみるとよい．

自分のプロセス ID を表示するプログラム

```
#include <stdio.h>
// ↓は getpid() システムコールのために．
// "unistd"は"UNIX Standard"の略．
#include <unistd.h>

int main(int argc, char **argv)
{
  int pid;  // PID (Process ID) 用変数

  pid = getpid();  // このプロセスの PID を取得
  printf("ぼくはプロセス %s . PID は %d だよ．\n", argv[0], pid);
  sleep(30);  // 30 秒お休みなさい

  return 0;
}
```

● フォークして子プロセスを生成するプログラム

フォークして子プロセスを生成するプログラム

```
#include <stdio.h>
// ↓は fork() システムコールのために．
#include <unistd.h>

int main(int argc, char **argv)
{
  int pid;

  // fork() を呼ぶと，プロセスが分岐．
  // 元のプロセスを親とする子プロセスが生成される．
  // 子プロセスは，親のコピー．
  // 親子それぞれ同じプログラムを実行し続ける
  pid = fork();
  if(pid < 0) printf("fork failed\n");

  if(pid == 0){  // fork() の戻り値が 0 なら子
    printf("ぼくは子プロセス．PID は %d だよ．\n", getpid());
    while(1){
      sleep(2);
      printf("子 「とうちゃん腹へった〜」\n");
    }
  }else{ // 親
    printf("わしは親プロセス．PID は %d じゃ．\n", getpid());
    while(1){
      sleep(5);
      printf("親 「息子よ，元気にしているか．」\n");
    }
  }

  return 0;
}
```

● パイプによるプロセス間通信 UNIX のシェル上では，パイプ '|' を使って 2 つのプログラムの入出力をつないでしまうことができる．これはプロセス間通信 (☆ 10) の一種である．

☆ 10) プロセス間通信については，次回解説予定．

```
$ ls -l                  ← ファイル一覧
$ ls -l  |  sort -k 5    ← その出力を 5 列目をキーとしてソート
$ ls -l  |  sort -k 5  |  awk '{ print $5 }'
```