目次

★1 プロセスの管理とスケジューリング（つづき）

★1.1 スケジューリング

カーネル内でプロセスのスケジューリングを担当する部分を**スケジューラ** (☆1) という．スケジューラは，プロセスの状態を管理し，実行すべきプロセスを選択し，選択したプロセスに CPU を割り当てる（制御を渡す）．

スケジューラは，レディ状態のプロセス（正確にはそのプロセス記述子へのポインタ）を**キュー** (☆2) に格納して処理する．CPU がアイドル状態になったら，キューからひとつプロセス（キューの先頭のもの）を取り出してそれにプロセッサを割り当てる．この処理の繰り返しによって，キューに並んだプロセスが順次プロセッサを割り当てられて実行されていく．後述するように，スケジューリングの方策によっては，一定時間が経過したなどの理由で実行中のプロセスを強制的にレディ状態に遷移させ（このような操作を**プリエンプション** (☆3) という），再びキューに入れることもある（そのプロセスはキューの一番後ろに並ぶ）．

スケジューリングの方策には，次のようなものがある．

到着順 プロセスをレディ状態になった順にキューに入れて実行していく．簡単に実現できるけれど，長時間かかるプロセスの実行開始直後に短時間で終わるプロセスが到着すると，長い時間待たされることになる．

優先度順 プロセス毎に何らかの基準で優先度を決めておき，その順に実行していく (☆4)．重要なプロセスの優先度を高く設定してシステム全体の効率を良くすることができるが，優先度の低いプロセスがいつまでも実行されないことがある（そのような状態を「飢餓状態 (starvation)」という）．

ラウンドロビン プロセスをレディ状態になった順にキューに入れて実行していくが，一定時間が経過したらプリエンプションしてキューの末尾に置く．どのプロセスも公平に実行される（レディなプロセスの数が平均 n ならそれらの経過時間はいずれも n 倍になる）が，コンテクストスイッチングが頻繁に起こるので，オーバヘッド (☆5) が大きい．

実際の OS では，これらの方策を組み合わせたり発展させた，より複雑な方策が用いられている．

☆1) スケジューラについては第 10 回資料も参照．

☆2) キュー: queue．**待ち行列**ともいう．スタックや木などと同じく基本的なデータ構造の一つ．先入れ先出し（First In First Out; FIFO）．

☆3) プリエンプション: preemption．日本語では横取り．現在では PC 向け OS も preemptive（横取り可能）であるが，昔の PC 向け OS の中にはそうではないものもあった．その場合，一つのアプリケーションが暴走すると，カーネルに制御が戻ってこなくてどうしようもなくなる（リセットするしかない）こともあった．

☆4) 優先度順でかつ preemptive な場合，あるプロセスの実行中により優先度の高いプロセスが到着したら，そちらがプロセッサを横取りする．

ラウンドロビン: Round Robin.

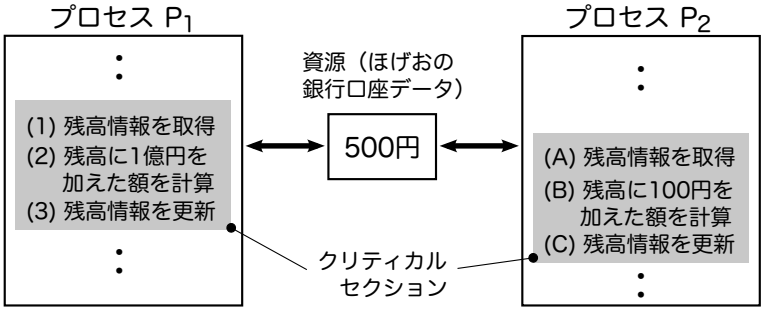
☆5) オーバヘッド: overhead．あることの実現のために余分に必要となるコストのこと．ここでは，プロセス切り替えの処理に要する時間が相当する．

Q1. 3 つのプロセス P_1, P_2, P_3 を到着順およびラウンドロビンでスケジューリングして実行した場合，各プロセスが生成されてから実行終了するまでの経過時間の平均はそれぞれ何秒になるか．ただし，次のことを仮定する

- これらのプロセスを単独で実行すると次の実行時間を要する:
  P_1 は 360 秒，P_2 は 54 秒, P_3 は 36 秒.
- これらのプロセスは P_1, P_2, P_3 の順に生成されたが，ほとんど同時であり，その時間差は上記の実行時間に比べて十分小さいので無視できる．
- プロセス切り替えまでの一定時間や切り替えに要するオーバヘッドは上記の実行時間に比べて十分小さいので無視できる．

★ 1.2 プロセス間の同期と通信

複数のプロセスが資源を共有しながら並行して動作する場合，それらがタイミングをあわせて（同期をとって）処理を行うようにしないと，矛盾が生ずることがある．例えば，右図のように銀行口座データのファイルを扱うプロセス P_1 と P_2 が並行動作していたとする．このとき，(1) → (A) → (2) → (3) → (B) → (C) という順に処理が行われると，大変なことになってしまう．



Q2. どう大変なことになるのか説明しなさい．

この問題を避けるためには，一方のプロセスが図のグレイの部分の処理を行う際には資源を占有して，その部分の処理が終わるまでは他方のプロセスで同じ資源にアクセスする処理を開始できないようにすればよい．このようにプロセスの実行を制御することを**排他制御**という．また，図のグレイの部分のように資源を占有してひと続きに実行する必要のある箇所を**クリティカルセクション**という．

クリティカルセクション: critical section

排他制御によく用いられるのは，**ロック**という方式である．これは，“lock”（鍵をかける）という名が表す通り，資源に鍵がかかっている／いないを表す 2 値変数を用いる方式である (☆ 6)．クリティカルセクションを実行する時は，次のようにする；(1) 対象資源のロック変数の値を確認し，ロックされていれば待機．さもなくばロックしてクリティカルセクションの実行に入る．(2) クリティカルセクションを抜けたらロックを解除．

☆ 6) プロセスの同期と排他制御の機構としては，資源のロック状態を 2 値ではなく整数で表す**セマフォ**もよく用いられるが，この授業では説明を省略する．

ロックのような排他制御の機構では，複数の資源をロックしたい場合などに，下手をすると 2 つのプロセスが互いに相手のロック解除を待って処理が先に進まなくなる**デッドロック** (☆ 7) 状態に陥ることがある．排他制御の仕組みは，デッドロックが起こらないよう注意深く設計しなければならない．

☆7) デッドロック: deadlock. 3 つ以上のプロセスの間でも起こり得る．

また，複数のプロセスが互いにメッセージをやりとりできるように，OS は一般に**プロセス間通信**の機構を備えている．例えば，UNIX 系 OS では，パイプ，ソケット，共有メモリなど様々なプロセス間通信の手段が提供されている．

プロセス間通信: InterProcess Communication, IPC

★ 1.3 ［発展］ スレッド

上述のように，OS は自分が管理する資源をプロセス毎に割り当てる．UNIX などの OS では，アドレス空間 (☆ 8) もプロセス毎に独立に生成する．そのため，

- プロセスの生成やコンテクストスイッチングのオーバヘッドが大きい
- アドレス空間を共有していないので，プロセス同士がメモリ経由でデータをやりとりするのが簡単ではない．プロセス間通信を行う必要がある

☆ 8) アドレス空間: メモリの空間．詳しくは次回以降．

といった問題を抱えている．最近の OS では，このような問題点を解決するため，1 つのプロセスの中に処理の流れを複数持たせることができるようにしている．この処理単位のことを**スレッド** (☆ 9) という．同じプロセス内のスレッドは資源を共有していて切り替えも「軽い」．複数の処理を切り替えながら実行するネットワークサーバや GUI を備えたプログラムなどの処理効率の改善に効果がある．SMP 等による並列処理にも向いている．

☆ 9) スレッド: thread.「軽い」ので，軽量プロセスと呼ぶこともある．

★1.4 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の一種である．

```
$ 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 }'
```