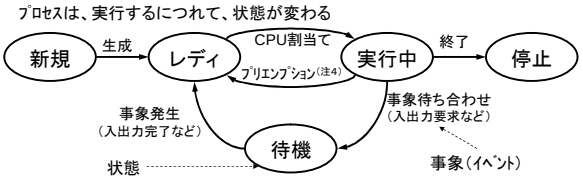
第2回　並行プロセス（1）

プロセス管理
スレッド
排他制御の必要性

（先週のｽﾗｲﾄﾞ）

プロセスの状態遷移

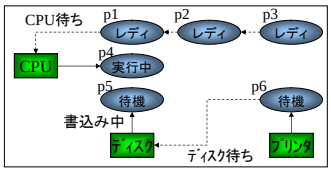
ﾌﾟﾛｾｽは、実行するにつれて、状態が変わる



各プロセスは事象の発生により状態が変化する。

事象（ｲﾍﾞﾝﾄ）の発生＝割り込み
・P4が処理を終了
　p1にCPUを割り当て→実行中
・P5が書き込み終了→実行待ち(注)
P6にはﾃﾞｨｽｸが割当てられる。読み込み処理を起動する場合には、p6も実行待ちとなる

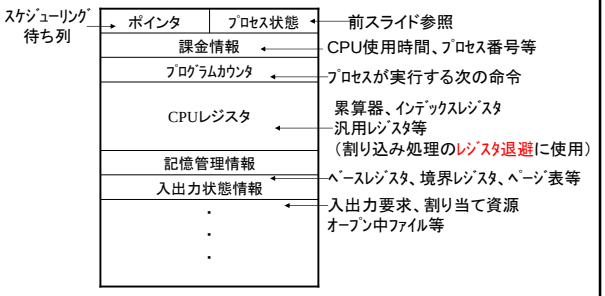
ｼｽﾃﾑ中には複数プロセスが存在



（先週のｽﾗｲﾄﾞ）

ﾌﾟﾛｾｽ制御ﾌﾞﾛｯｸ（PCB：Process Control Block）

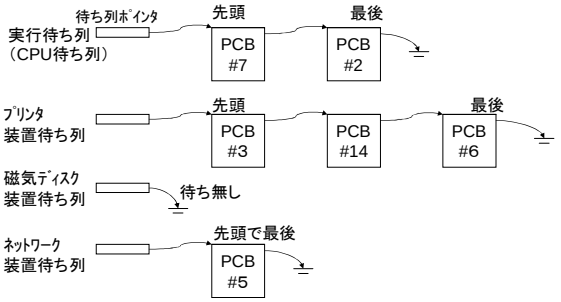
ﾌﾟﾛｾｽの生成時に（ﾌﾟﾛｾｽ毎に）作成され、終了時に消滅する。
活動中のﾌﾟﾛｾｽに関する情報が格納される。（ﾌﾟﾛｾｽはPCBで表される）



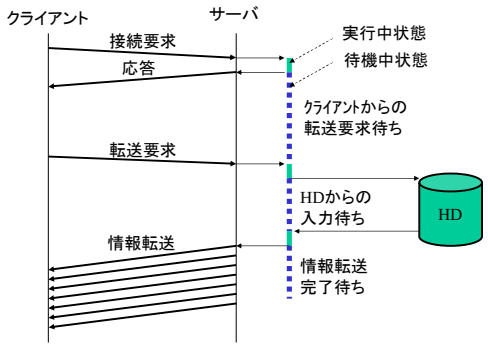
（先週のｽﾗｲﾄﾞ）

実行待ち列と様々な装置待ち列

実行待ち列：レディ状態のﾌﾟﾛｾｽが、CPUが割り当てられるまで待つ
装置待ち列：入出力要求をしたﾌﾟﾛｾｽ（待機中状態）が、装置が空くまで待つ
ﾌﾟﾛｾｽの待ち列は、PCBのﾘｽﾄにより表される。

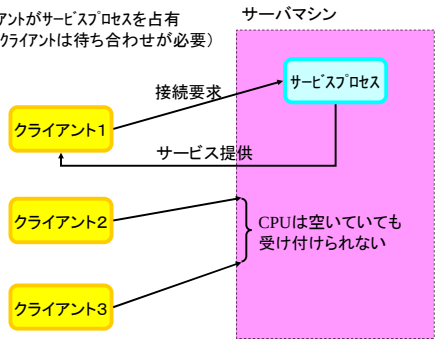


ネットワークサービスの例

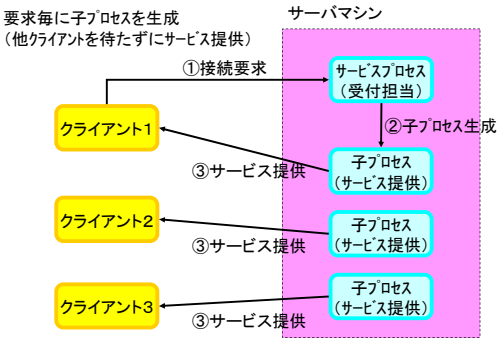


1プロセスでの実現

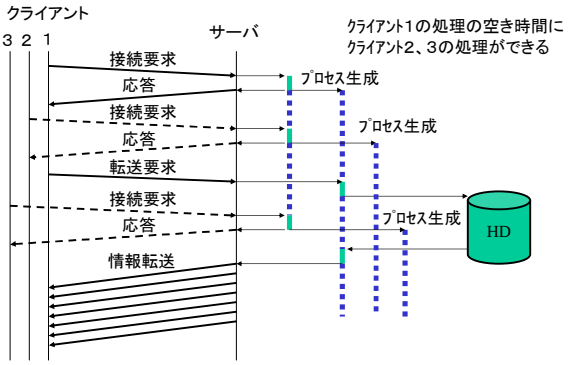
1ｸﾗｲｱﾝﾄがｻｰﾋﾞｽﾌﾟﾛｾｽを占有
（他のｸﾗｲｱﾝﾄは待ち合わせが必要）



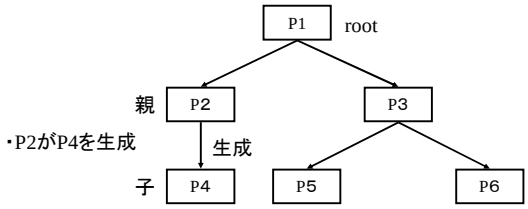
子プロセスを生成



ネットワークサービスの例



プロセスグラフ



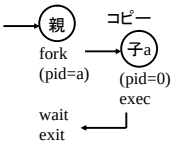
親プロセスは、子プロセスを生成し、以下の形態の実行が可能

実行の形態
形態1：親と子が並行して実行
形態2：親は子が終了するまで待つ

変数の共用
形態1：すべての変数を共用
形態2：一部の変数のみを共用

（参考）Unixのプロセス生成の例

fork( )システムコール
親プロセスのプログラムをコピーした子プロセスを生成し，
それぞれのプロセスにreturnする．



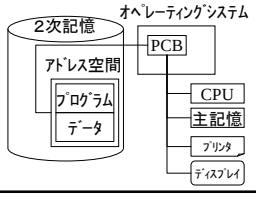

```
#include <stdio.h>
main(int argc, char *argv[ ] )
{
int pid;
pid = fork( ) ; /* 生成 */

if (pid = = 0) {        /* 子プロセスにreturnした場合*/
  execlp ("/bin/ls", "ls ", NULL) ; /*ﾌﾟﾛｸﾞﾗﾑを子プロセスのものに置換え*/
}
else {                  /* 親プロセスにreturnした場合*/
  wait (NULL) ; /* 子プロセスが終了するまで待機状態になる */
  printf ("Child Complete") ;
  exit (0) ; /*親の親に終了条件を通知*/
}
}
```

（参考）Webサーバプログラムの例

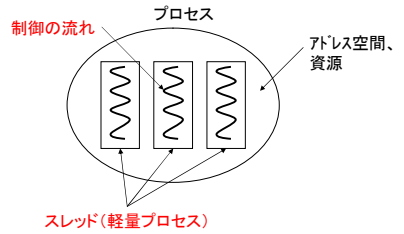
```
#include <stdio.h>
#include <string.h>     /* memset() */
#include <unistd.h>     /* fork() */
        /* メインループ */
        while(tcpserver_wait(&server, &client)) {   /* クライアント接続待ち */
          /* クライアントから接続された */
        pid_t child_pid = fork();               /* 子プロセス生成 */
        if(child_pid==0) {
          int ret = 0;                          /* 子プロセス */
           /* 親プロセスのソケットを子プロセスに触らせたくないのでクローズ */
          tcpserver_close(&server);
             /* 子プロセスにやらせたい処理→HTTPD処理（1リクエスト分）*/
             ret = do_httpd(&client);
             /* 後片付け：クライアントとの接続終了処理 */
             tcpclient_close(&client);    /* FIN,ACK送信 */
        exit(ret); /* 子プロセスはここで自ら終了 */
          }
          else {
```

プロセスの生成と管理

- プロセスの実行に必要な資源
  - CPU、主記憶装置・2次記憶装置、入出力装置
- プロセスの生成時に作られるもの
  - アドレス空間：実行形式のﾌﾟﾛｸﾞﾗﾑとﾃﾞｰﾀを配置する空間
  - PCB：プロセスの状態や割り当て資源を管理する
- プロセスの生成にはかなりの負荷が伴う
  - アドレス空間の生成
  - システム資源の消費



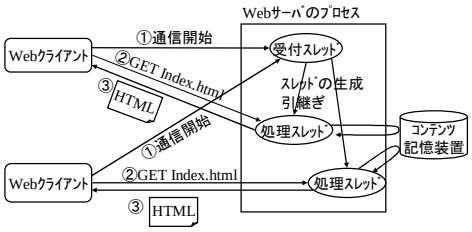
スレッド（軽量プロセス）



1つのプロセスの中に複数の制御の流れを持てるようにする
この制御の流れをスレッドと言う
スレッドはプロセスが持つアドレス空間や資源を共有する
スレッド毎に、プロセスと同様の独立した処理ができる
プロセスを生成するよりも少ない負荷でスレッドが生成できる
但し、他のｽﾚｯﾄﾞの影響を十分に考慮したﾌﾟﾛｸﾞﾗﾐﾝｸﾞが必要

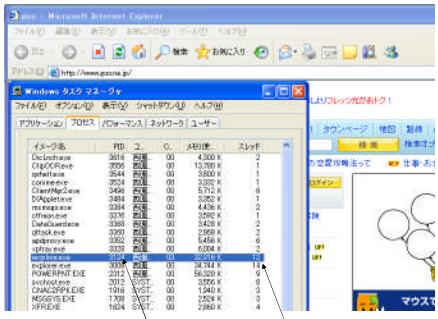
（参考）スレッドとプロセス

マルチスレッド型Webサーバの処理

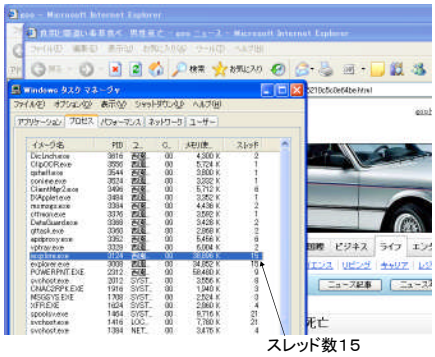


Webｸﾗｲｱﾝﾄの処理（Windowsのｲﾝﾀｰﾈｯﾄｴｸｽﾌﾟﾛｰﾗの場合）
・ｽﾀｰﾄﾒﾆｭｰからﾌﾟﾛｸﾞﾗﾑを起動→プロセスを生成（必要なｽﾚｯﾄﾞも生成）
・新しいｳｨﾝﾄﾞｳを開く→ｽﾚｯﾄﾞを作成

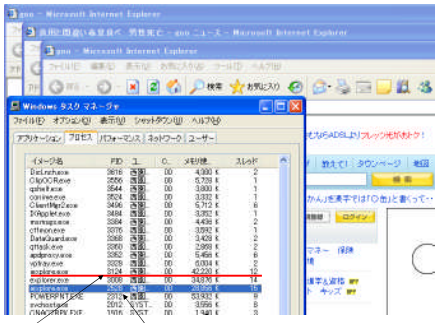
IE（ver 6）起動



新しいウィンドウを開く



スタートメニューから再度IE起動



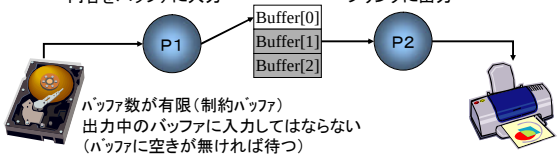
ﾌﾟﾛｾｽ間の協調と同期（制約ﾊﾞｯﾌｧ問題）

バッファを共有する2つのプロセス（P1、P2）

資源

P1：ハードディスクの
内容をバッファに入力

P2：バッファのデータを
プリンタに出力



ﾊﾞｯﾌｧ数が有限（制約ﾊﾞｯﾌｧ）
出力中のバッファに入力してはならない
（ﾊﾞｯﾌｧに空きが無ければ待つ）
入力中のバッファを出力してはならない
（ﾊﾞｯﾌｧが全て空なら待つ）

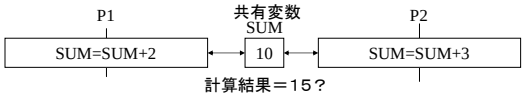
プロセス間の同期が必要　同期：時間的な処理の流れを制御すること

同期に必要な機能
・排他制御：共有資源を一度に1つのプロセスのみが使用するようにする
・プロセス間通信：プロセス間で情報の受け渡しをする

p.104

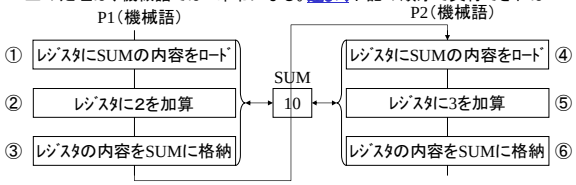
排他制御の必要性

P1とP2が変数SUMを共有し、それぞれ下記の命令を実行。SUMの値は？



計算結果＝15？

上の処理は、機械語では3命令になる。運よく下記の順序で実行できれば



①②③の結果、SUM=12。④⑤⑥の結果SUM=15

プロセス切り替えの例

命令の切れ目で割込みが発生し、プロセスの処理が切り替わることがある

例1：CPUスケジューリング（ラウンドロビン）における量子時間が経過
P1が実行中状態、P2がレディ状態
（1）P1が実行中に量子時間経過（時計割込みが発生）
（2）P1の実行が中断（OSの割り込み処理にジャンプ）
（3）OSは，P1をレディ状態にする（レジスタ退避、実行待ち列に並ばせる）
（4）P2を実行中状態にする

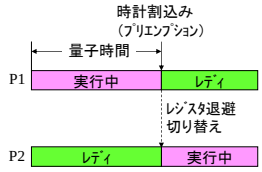
例2：仮想記憶で次の命令が主記憶に無い（ページフォールト）場合
P2が実行中状態、P1がレディ状態
（1）P2が実行中にページフォールト（イリーガル番地アクセスの割込みが発生）
（2）P2の実行が中断（OSの割り込み処理にジャンプ）
（3）OSは，ページアウト，ページイン処理を開始
（4）P2を待機状態にする（レジスタ退避、入出力待ち列に並ばせる）
（5）P1のレジスタを復元、実行中状態にし、P1を再開させる

実行中のプロセスは，突然，処理を中断させられる

p.93

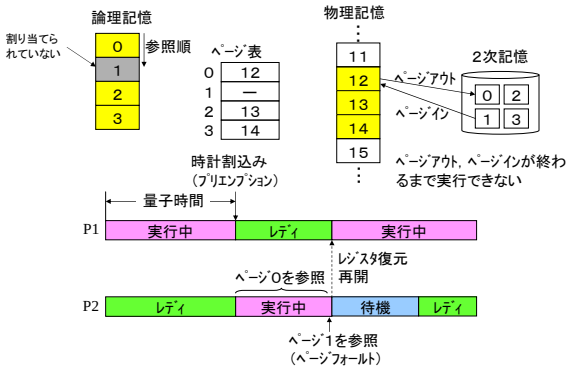
CPUスケジューリング（ラウンドロビン）

- 各プロセスに量子時間(time quantum)(注)単位でCPUを割当てる
  - 各プロセスを量子時間ずつ順繰りに実行することを繰り返す
    - 量子時間が経過すると、実行中のプロセスを中断（CPUを取り上げ）、次のプロセスにCPUを割り当てて実行させる.



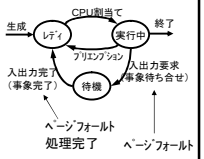
p.140

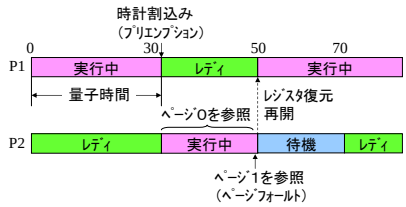
仮想記憶におけるページフォールト



ページフォールトと状態遷移の例

時刻	事象の発生	P1	P2
0	P1にCPU割当て	実行中	レディ
30	プリエンプション，CPU割当て	レディ	実行中
50	ページフォールト，CPU割当て	実行中	待機
70	ページフォールト処理完了	〃	レディ
	：		



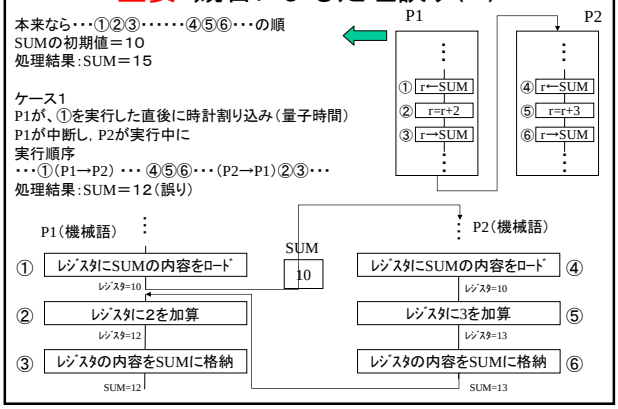


p.104

重要：競合による処理誤り（1）

本来なら・・・①②③・・・・・・④⑤⑥・・・の順
SUMの初期値＝10
処理結果：SUM＝15

ケース1
P1が、①を実行した直後に時計割り込み（量子時間）
P1が中断し，P2が実行中に
実行順序
・・・①（P1→P2）・・・④⑤⑥・・・（P2→P1）②③・・・
処理結果：SUM＝12（誤り）

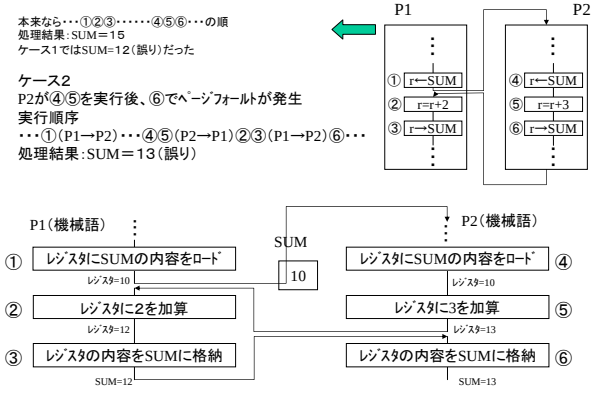


p.104

重要：競合による処理誤り（2）

本来なら・・・①②③・・・・・④⑤⑥・・・の順
処理結果：SUM＝15
ケース1ではSUM=12（誤り）だった

ケース2
P2が④⑤を実行後、⑥でﾍﾟｰｼﾞﾌｫｰﾙﾄが発生
実行順序
・・・①（P1→P2）・・・④⑤（P2→P1）②③（P1→P2）⑥・・・
処理結果：SUM＝13（誤り）



実行結果

ケース1

	レｼﾞｽﾀ	SUM
① r←SUM	10	10
割り込み		
レｼﾞｽﾀ退避（10）、P2に切り替え		
・・・		
④ r←SUM	10	10
⑤ r＝r+3	13	10
⑥ r→SUM	13	13
・・・ P2が終了		
P1が再度実行中になる		
（レｼﾞｽﾀを復元）	10	13
② r＝r+2	12	13
③ r→SUM	12	12
・・・ P1が終了		

レジスタの値はPCBに退避される

切り替えの順番や初期値などを変えて出題
SUMの値が何になるか、理解しておくこと

ケース2

	レｼﾞｽﾀ	SUM
① r←SUM	10	10
割り込み		
レｼﾞｽﾀ退避（10）、P2に切り替え		
・・・		
④ r←SUM	10	10
⑤ r＝r+3	13	10
割り込み		
レｼﾞｽﾀ退避（13）、P1に切り替え		
P1が再度実行中になる		
（レｼﾞｽﾀを復元）	10	10
② r＝r+2	12	10
③ r→SUM	12	12
・・・ P1が終了		
P2が再度実行中になる		
（レｼﾞｽﾀを復元）	13	12
⑥ r→SUM	13	13
・・・ P2が終了		