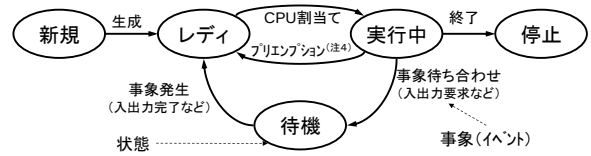
第2回　並行プロセス（1）

プロセス管理
スレッド
排他制御の必要性

（先週のスライド）

プロセスの状態遷移

プロセスは，実行するにつれて，状態が変わる

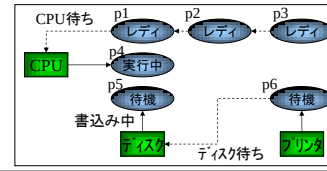


各プロセスは事象の発生により状態が変化する．

事象（イベント）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)

(注) P6にはデ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

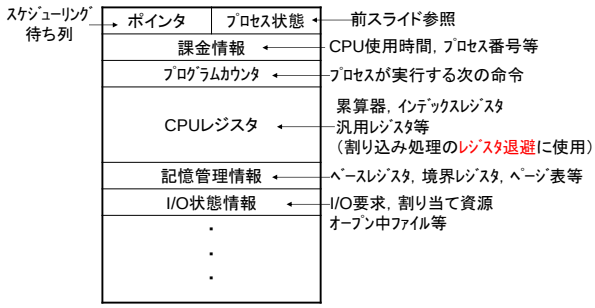
システム中には複数プロセスが存在



（先週のスライド）

プロセス制御ブロック（PCB：Process Control Block）

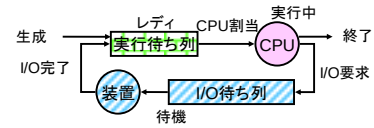
プロセスの生成時に（プロセス毎に）作成され，終了時に消滅する．
活動中のプロセスに関する情報が格納される．（プロセスはPCBで表される）

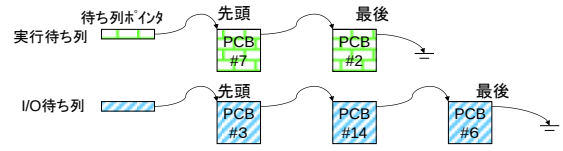


（先週のスライド）

実行待ち列とI/O待ち列

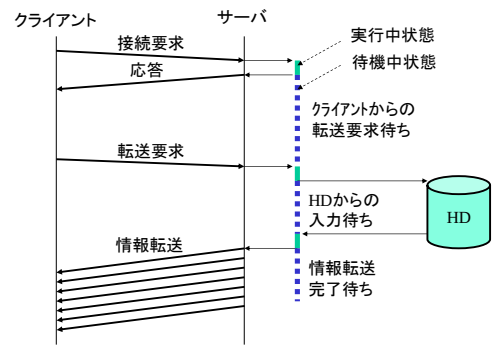
実行待ち列：レディ状態のプロセスが，CPUが割り当てられるまでの待ち列
I/O待ち列：I/O要求をした待機状態のプロセスが装置が空くまでの待ち列(注)
待ち列は，PCBのリストにより表される．





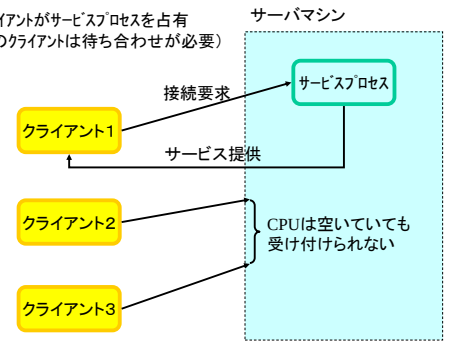
注：実際にはI/O待ち列は，I/O装置毎に作られる．以下，簡略化のため1装置の場合で説明

ネットワークサービスの例

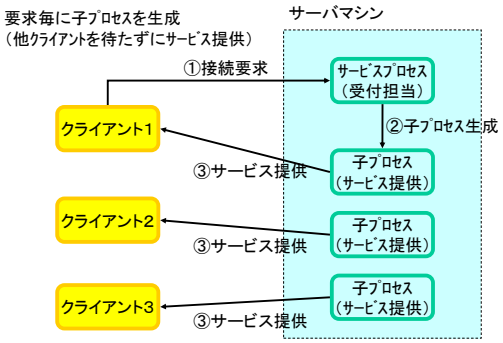


1プロセスでの実現

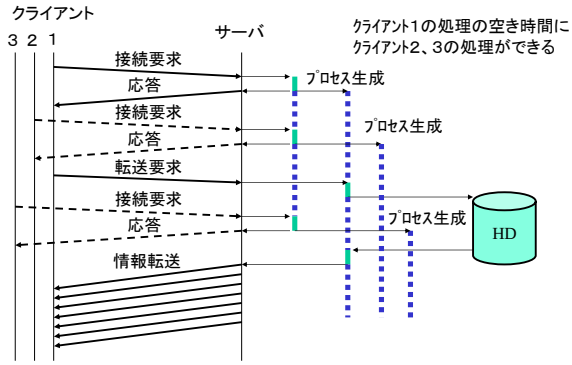
1クライアントがサービスプロセスを占有
（他のクライアントは待ち合わせが必要）



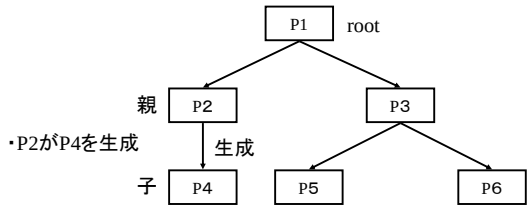
子プロセスを生成



ネットワークサービスの例



プロセスグラフ

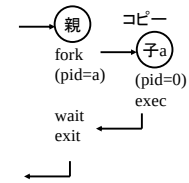


親プロセスは、子プロセスを生成し、以下の形態の実行が可能

実行の形態	変数の共用
形態1：親と子が並行して実行	形態1：すべての変数を共用
形態2：親は子が終了するまで待つ	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．



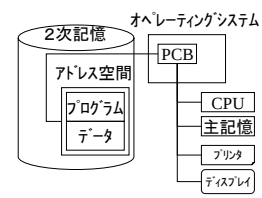

```
#include <stdio.h>
main(int argc, char *argv[ ] )
{
int pid;
pid = fork( ) ; /* 生成 */

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

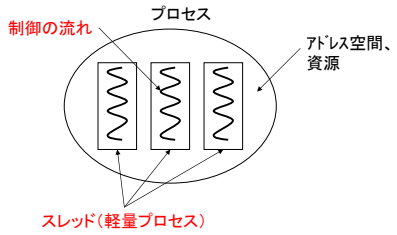
```
#include <stdio.h>
#include <string.h>    /* memset() */
#include <unistd.h>    /* fork() */
        /* メインループ */
        while(tcpserver_wait(&server, &client)) {   /* クライアント接続待ち */
          /* クライアントから接続された */
        pid_t child_pid = fork();              /* 子プロセス生成 */
         if(child_pid==0) {
           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



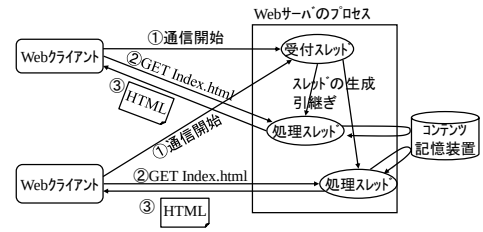
スレッド（軽量プロセス）



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

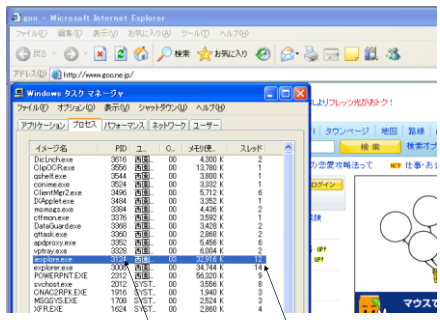
参考：スレッドとプロセス

マルチスレッド型Webサーバの処理

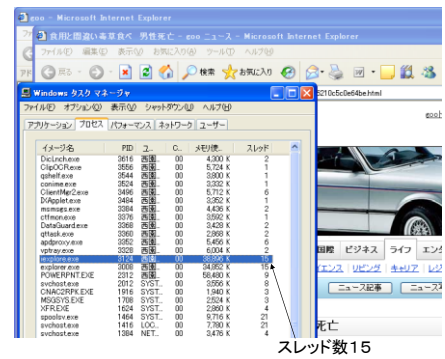


Webｸﾗｲｱﾝﾄの処理（Windowsのｲﾝﾀｰﾈｯﾄｴｸｽﾌﾟﾛｰﾗの場合）
・ｽﾀｰﾄﾒﾆｭｰからﾌﾟﾛｸﾞﾗﾑを起動→プロセスを生成（必要なｽﾚｯﾄﾞも生成）
・新しいウィンドウを開く→ｽﾚｯﾄﾞを作成

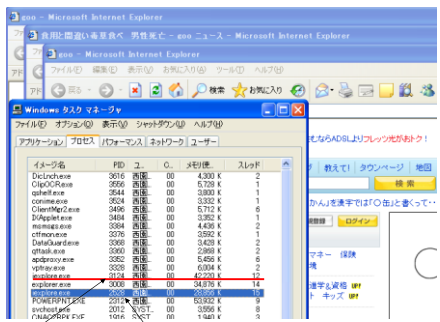
IE（ver 6）起動



新しいウィンドウを開く

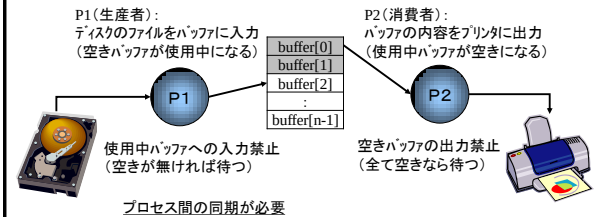


スタートメニューから再度IE起動



ﾌﾟﾛｾｽ間の協調と同期（制約ﾊﾞｯﾌｧ問題）

ﾃﾞｨｽｸからﾌｧｲﾙを読み込み，ﾌﾟﾘﾝﾀに出力．これを効率的に処理したい．
2つのﾌﾟﾛｾｽ（P1, P2）が複数のﾊﾞｯﾌｧを共有（共有資源）．これを用いた並行処理で実現．
ﾊﾞｯﾌｧの数は有限（そのため，制約ﾊﾞｯﾌｧ問題という）



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

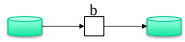
同期に必要な機能
・排他制御：共有資源を一度に1つのプロセスのみが使用するようにする
・プロセス間通信：プロセス間で情報の受け渡しをする

参考：Linuxの入出力システムコール

open(*pathname, flags): ファイルまたはﾃﾞﾊﾞｲｽの使用を開始し，ﾌｧｲﾙ識別子(fd)を返す．
*pathname：ﾌｧｲﾙのpathname
flags：読み出し専用（O_RDONLY），書き込み専用（O_WRONLY）などの指定
ﾌﾟﾘﾝﾀやﾈｯﾄﾜｰｸなどのﾃﾞﾊﾞｲｽもﾌｧｲﾙとして扱う．
close(fd): fdで指定されたﾌｧｲﾙの使用を終了し，割り当てられたfd値を解放する．

read(fd, *buf, count): fdで指定されたﾌｧｲﾙから最大countﾊﾞｲﾄを変数bufに読み込む．
write(fd, *buf, count):変数bufからfdで指定されたﾌｧｲﾙに最大countﾊﾞｲﾄを書き出す．
（どちらのｼｽﾃﾑｺｰﾙも返り値は，実際に読み書きされたﾊﾞｲﾄ数）

```
int fdi, fdo;
char b[4096];
fdi = open("in_file", O_RDONLY);
fdo = open("out_file", O_WRONLY);
while (0 < read(fdi, b, 4096)) { /*ﾌｧｲﾙを4096ﾊﾞｲﾄずつread. 最後は返り値が0になる*/
      write(fdo, b, 4096);       /*このﾌﾟﾛｸﾞﾗﾑは簡単だが効率が悪い*/
}
   close(fdi);
   close(fdo);
```

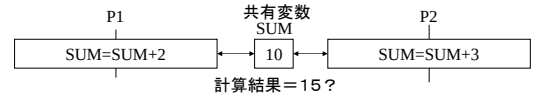


制約ﾊﾞｯﾌｧ問題のﾌﾟﾛｸﾞﾗﾑは，複数ﾊﾞｯﾌｧを使用し，並行ﾌﾟﾛｾｽにより，読み込みと書き込みを同時に行うので効率が良い．そのかわり，排他制御が必要になる．

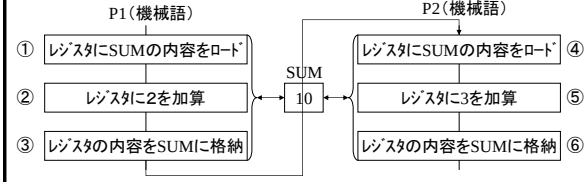
p.104

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？



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ｽｹｼﾞｭｰﾘﾝｸﾞ（ﾗｳﾝﾄﾞﾛﾋﾞﾝ）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

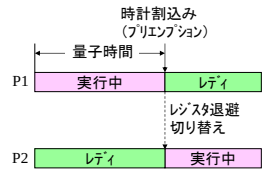
例2：仮想記憶で次の命令が主記憶に無い（ﾍﾟｰｼﾞﾌｫｰﾙﾄ）場合
P2が実行中状態、P1がレディ状態
（1）P2が実行中にページフォールト（ｲﾘｰｶﾞﾙ番地ｱｸｾｽの割込みが発生）
（2）P2の実行が中断（OSの割り込み処理にジャンプ）
（3）OSは，ページアウト，ページイン処理を開始
（4）P2を待機状態にする（ﾚｼﾞｽﾀ退避、入出力待ち列に並ばせる）
（5）P1のﾚｼﾞｽﾀを復元、実行中状態にし、P1を再開させる

実行中のプロセスは，突然，処理を中断させられる

p.93

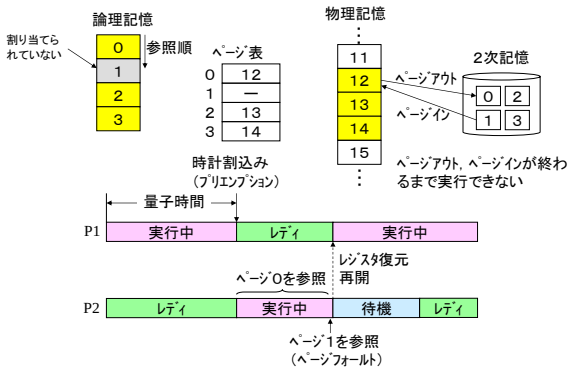
CPUｽｹｼﾞｭｰﾘﾝｸﾞ（ﾗｳﾝﾄﾞﾛﾋﾞﾝ）

- 各ﾌﾟﾛｾｽに量子時間(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、実行中のﾌﾟﾛｾｽを中断（CPUを取り上げ）、次のﾌﾟﾛｾｽにCPUを割り当てて実行させる．



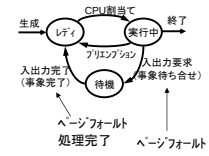
p.140

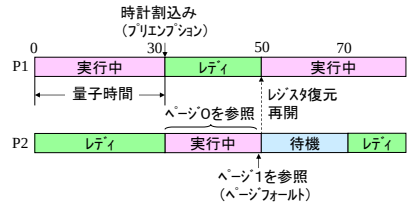
仮想記憶におけるページフォールト



ﾍﾟｰｼﾞ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	ﾌﾟﾘｴﾝﾌﾟｼｮﾝ，CPU割当て	レディ	実行中
50	ﾍﾟｰｼﾞﾌｫｰﾙﾄ，CPU割当て	実行中	待機
70	ﾍﾟｰｼﾞﾌｫｰﾙﾄ処理完了	〃	レディ
	：		



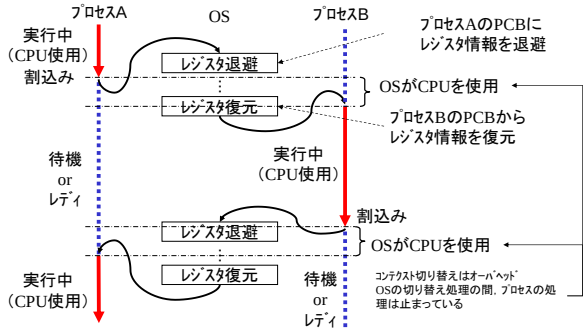


（基礎OSのｽﾗｲﾄﾞ）

コンテクスト切替え（コンテクストスイッチとも言う）

実行中状態のﾌﾟﾛｾｽから他のﾌﾟﾛｾｽにCPU状態（レジスタなど）を切り替える

context：文脈，状況→処理の流れはCPU状態（レジスタなど）で表される．これをコンテクストという．

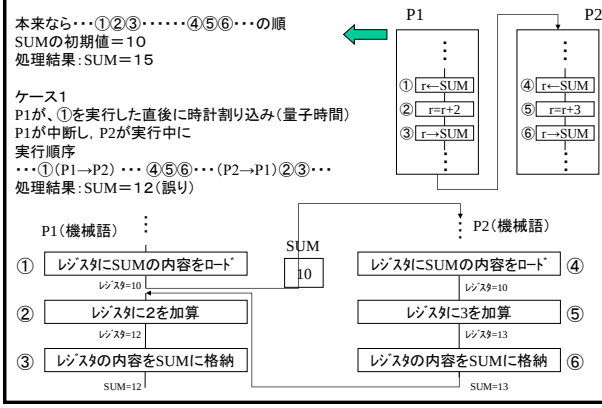


p.104

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）

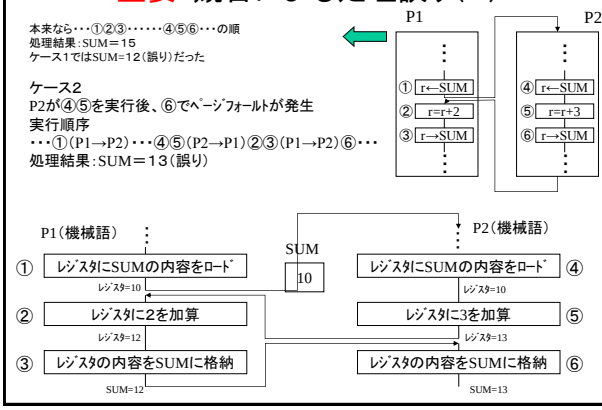


p.104

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

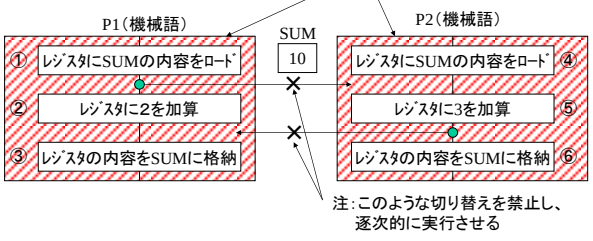
	ﾚｼﾞｽﾀ	SUM
① r←SUM	10	10
割り込み		
ﾚｼﾞｽﾀ退避（10）、P2に切り替え		
･･･		
④ r←SUM	10	10
⑤ r＝r+3	13	10
割り込み		
ﾚｼﾞｽﾀ退避（13）、P1に切り替え		
P1が再度実行中になる		
（ﾚｼﾞｽﾀを復元）	10	10
② r＝r+2	12	10
③ r→SUM	12	12
･･･ P1が終了		
P2が再度実行中になる		
（ﾚｼﾞｽﾀを復元）	13	12
⑥ r→SUM	13	13
･･ P2が終了		

p.104

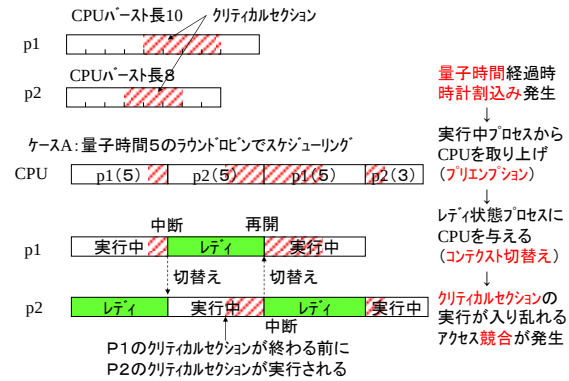
クリティカルセクション（危険区域）

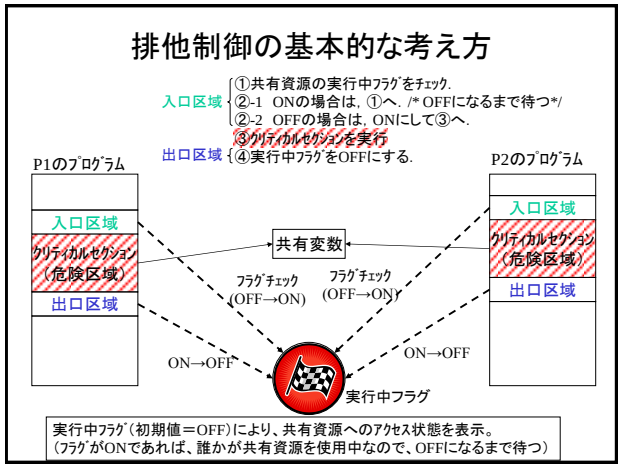
共有変数SUMへのアクセスが競合すると不正な結果となる
↑
（P1、P2の両方が使用する資源）

これを防止するために、①～③と④～⑥を排他的に実行する(注)必要がある
このように、排他的実行が必要な部分をクリティカルセクションという



CPUｽｹｼﾞｭｰﾘﾝｸﾞによる競合




排他制御の基本的な考え方
入口区域
①共有資源の実行中ﾌﾗｸﾞをﾁｪｯｸ.
②-1 ONの場合は，①へ．/* OFFになるまで待つ*/
②-2 OFFの場合は，ONにして③へ.
③ｸﾘﾃｨｶﾙｾｸｼｮﾝを実行
出口区域
④実行中ﾌﾗｸﾞをOFFにする.
P1のﾌﾟﾛｸﾞﾗﾑ
P2のﾌﾟﾛｸﾞﾗﾑ
入口区域
ｸﾘﾃｨｶﾙｾｸｼｮﾝ（危険区域）
出口区域
共有変数
ﾌﾗｸﾞﾁｪｯｸ (OFF→ON)
ﾌﾗｸﾞﾁｪｯｸ (OFF→ON)
ON→OFF
ON→OFF
実行中フラグ
実行中ﾌﾗｸﾞ（初期値＝OFF）により、共有資源へのｱｸｾｽ状態を表示。
（ﾌﾗｸﾞがONであれば、誰かが共有資源を使用中なので、OFFになるまで待つ）