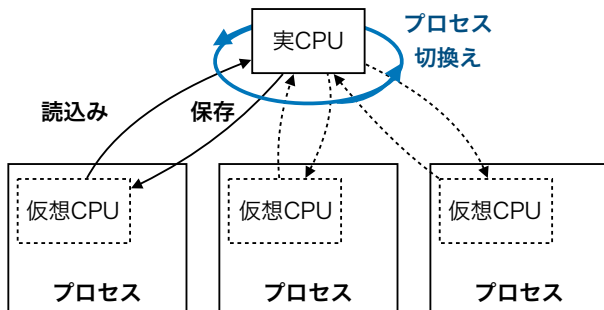


オペレーティングシステム

第3章 CPU の仮想化

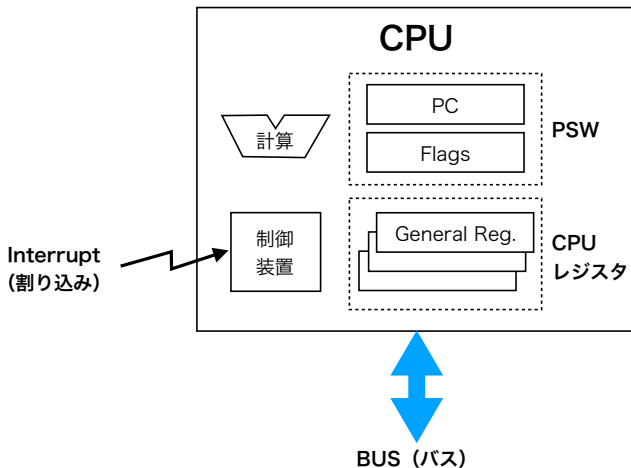
<https://github.com/tctsigemura/OSTextBook>

時分割多重による CPU の仮想化



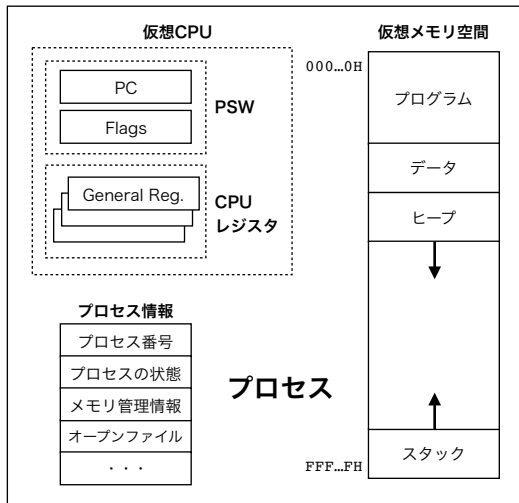
- 時分割多重：CPU が実行するプロセスを次々切換える。
- コンテキストスイッチ：CPU が実行するプロセスを切換えること。
- ディスパッチ：プロセスに CPU を割り付ける。(実行開始)
- ディスパッチャ：ディスパッチするプログラムのこと。

CPU の構造 (参考)



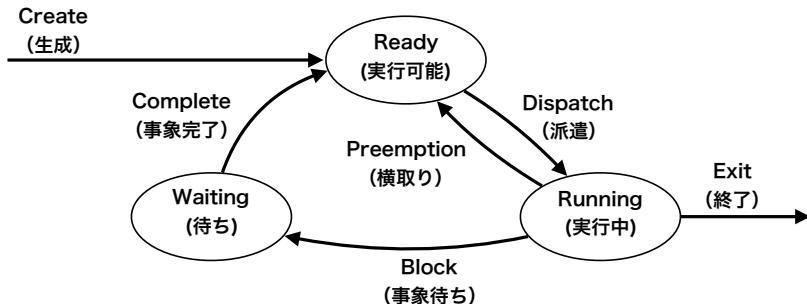
- コンテキスト = PSW + CPU レジスタ
- コンテキストを保存・ロードして次のプロセスに
- コンテキストスイッチ

プロセスの構造（参考）



- 仮想 CPU にコンテキストを保存

プロセスの状態遷移



- 基本的な三つの状態
- 六つの状態遷移

プロセスの切換えの原因

- イベント

プロセス自ら「システムコールを発行する」Block する

他のプロセスから「**干渉**を受け」Block する

他のプロセスから「**干渉**を受け」Preemption する

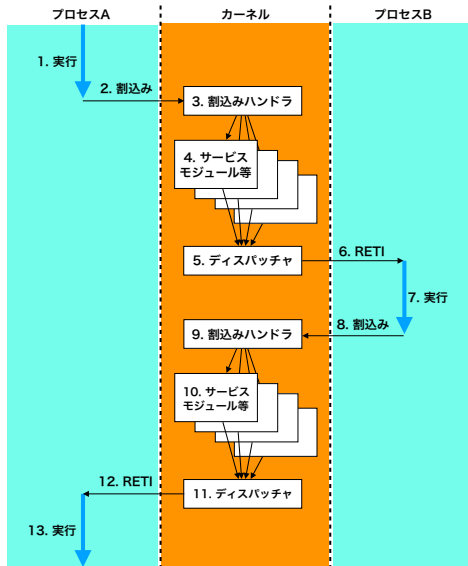
他のプロセスから「**干渉**を受け」Complete する

i/O 完了やタイマ完了のイベントにより Complete する

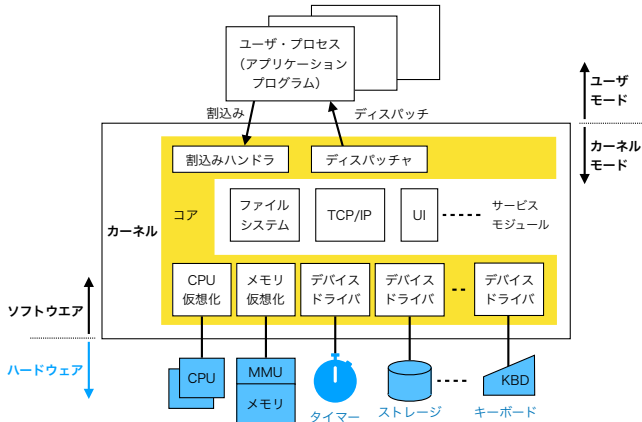
- タイムスライシング

クオンタムタイムを使い切ったプロセスは Preemption する

プロセスの切換え手順

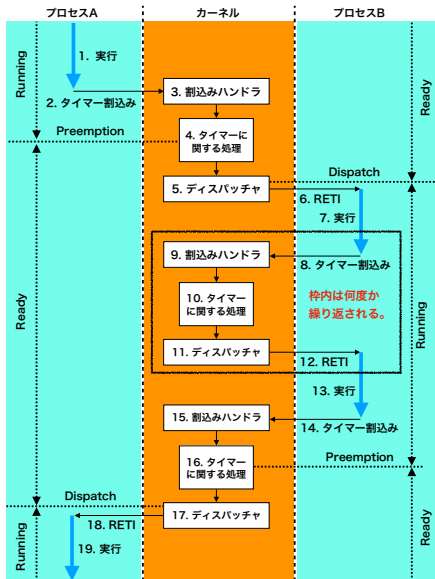


オペレーティングシステムの構造（参考）



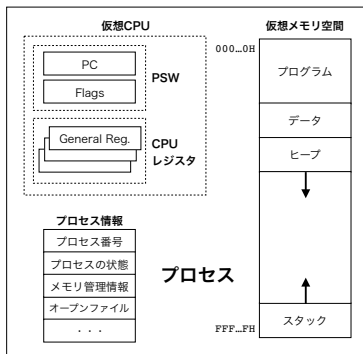
- 割り込みハンドラ
- サービスモジュール
- ディスパッチャ

プロセスの切換えの例



PCB (Process Control Block)

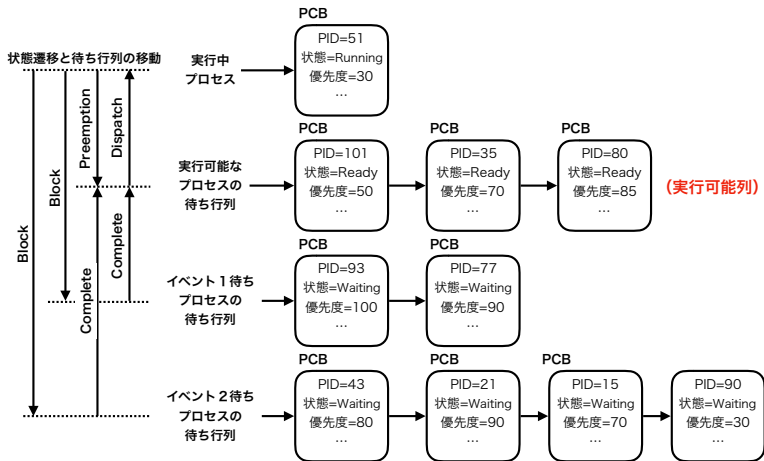
- プロセスを表現するカーネル内データ構造
- プロセス毎に存在する
- カーネル内のプロセステーブルに格納される



PCB の内容

- 仮想 CPU
- プロセス番号
- 状態 (Running, Waiting, Ready 等)
- 優先度
- 統計情報 (CPU 利用時間等)
- 次回のアラーム時刻
- 親プロセス
- 子プロセス一覧
- シグナルハンドリング
- 使用中のメモリ
- オープン中のファイル
- カレントディレクトリ
- プロセス所有者のユーザ番号
- PCB のリストを作るためのポインタ

PCB のリスト



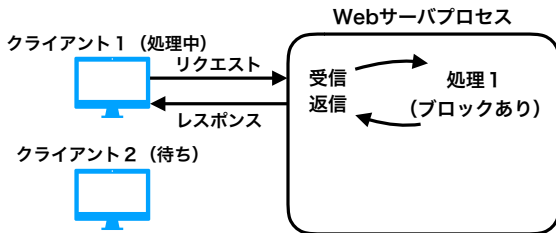
- Ready 状態 PCB のリスト = 実行可能列 (優先順位順にソート)
- イベント毎の Waiting 状態 PCB のリスト = イベント待ち行列

スレッド (Thread)

- CPU の仮想化によりマルチプログラミングが可能になった.
- プロセスが並行 (Concurrent) に実行できる.
- プロセスは**一つ**の仮想 CPU を持っている.
- プロセスはコンピュータを仮想化したもの.
 - CPU が一つしかないコンピュータを仮想化している.
 - CPU を複数持つ SMP を仮想化するには不十分.
- 一つのプロセスが複数の仮想 CPU をもつモデルを導入する.
- プロセスが処理の流れ**スレッド**を複数持つことができる.

スレッドの役割（１）

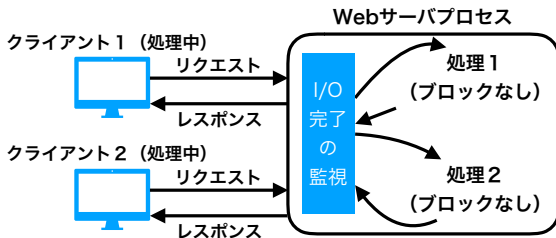
マルチプログラミングを用いない Web サーバ



- 処理は順番に処理される。
- 前の処理が終わるまでクライアントは待たされる。

スレッドの役割（2）

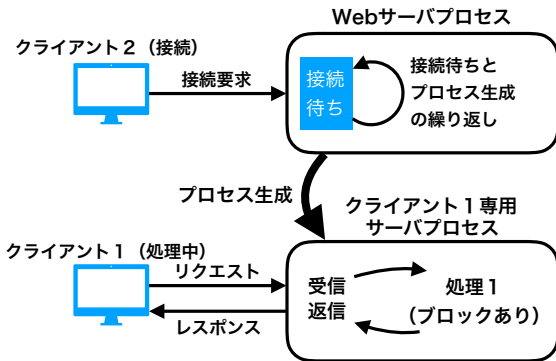
マルチプログラミングを用いない Web サーバ



- 工夫すると並列して処理することも可能
- しかし、プログラミングが難しい。

スレッドの役割 (3)

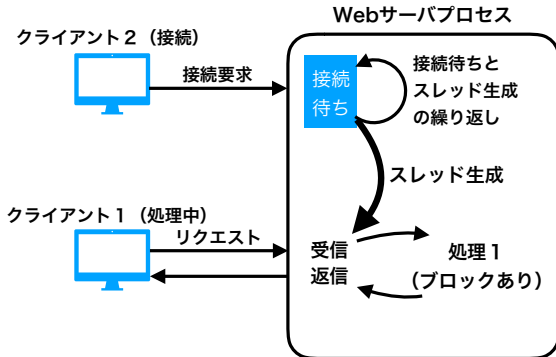
マルチプログラミングを用いる Web サーバ (プロセス版)



- クライアント毎にプロセスを起動 (fork()) する.
- プログラミングは易しい.
- しかし、処理が重い.

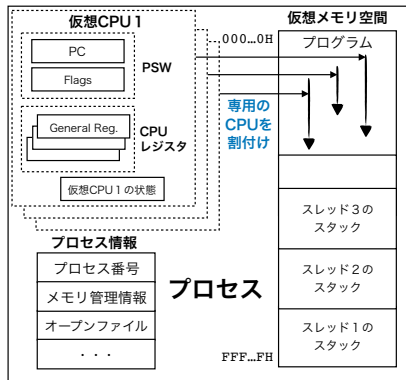
スレッドの役割（４）

マルチプログラミングを用いる Web サーバ（スレッド版）



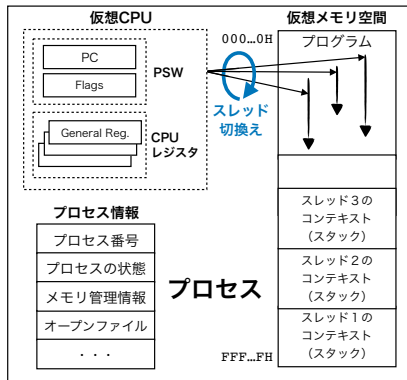
- クライアント毎にスレッドを起動する.
- プロセスの起動より 10~100 倍速い.
- スレッド間は情報を共有しやすい.
- プログラミングは少し難しい.

スレッドの形式（１）－カーネルスレッド－



- プロセスが複数の仮想 CPU を持つ。

スレッドの形式（２）－ユーザスレッド－



- ユーザプログラム（ライブラリ）の工夫でスレッドを実現する。
- 並行（Parallel）実行はできない。
- どれかのスレッドがブロックすると全スレッドが停止する。

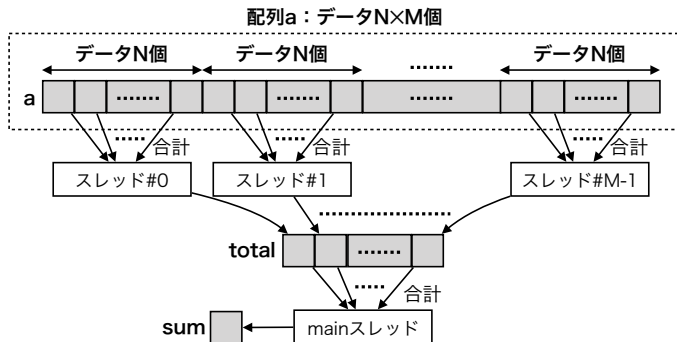
スレッドの形式（３）－スレッドモデル－

上記の２方式を組み合わせた３種類のスレッドモデルがある．

- *One-to-One Model*
一つのユーザスレッドを一つのカーネルスレッドで実行する．
- *Many-to-One Model*
複数のユーザスレッドを一つのカーネルスレッドで実行する．
- *Many-to-Many Model*
複数のユーザスレッドを複数のカーネルスレッドで実行する．

スレッドの使用例（1）

M 個のスレッドで手分けをして合計を計算する様子
(複数のカーネルスレッド (CPU) で手分けすることで短時間で処理が終わるはず)



スレッドの使用例 (2)

M 個のスレッドで合計を計算するプログラム

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #define N 1000                                // 1スレッドの担当データ数
5  #define M 10                                  // スレッド数
6  pthread_t tid[M];                             // M個のスレッドのスレッド ID
7  pthread_attr_t attr[M];                       // M個のスレッドの属性
8  int a[M*N];                                   // このデータの合計を求める
9  int total[M];                                 // 各スレッドの求めた部分合
10 typedef struct { int no, min, max; } Args;    // スレッドに渡す引数の型定義
11
12 void *thread(void *arg) {                      // 自スレッドの担当部分のデータの合計を求める
13     Args *args = arg;                         // m 番目のスレッド
14     int sum = 0;                               // 合計を求める変数
15     for (int i=args->min; i<args->max; i++) {   // a[N*m ... (N+1)*m] の
16         sum += a[i];                           // 合計を sum に求める。
17     }
18     total[args->no]=sum;                        // 担当部分の合計を記録
19     return NULL;                               // スレッドを正常終了する
20 }
```

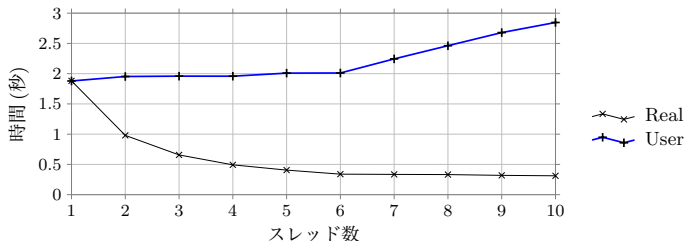
スレッドの使用例 (3)

```
22 int main() {                                     // main スレッドの実行はここから始まる
23     // 擬似的なデータを生成する
24     for (int i=0; i<M*N; i++)                     // 配列 a を初期化
25         a[i] = i+1;
26     // M 個のスレッドを起動する
27     for (int m=0; m<M; m++) {                     // 各スレッドについて
28         Args *p = malloc(sizeof(Args));           // 引数領域を確保
29         p->no = m;                                  // m 番目のスレッド
30         p->min = N*m;                               // 担当範囲下限
31         p->max = N*(m+1);                           // 担当範囲上限
32         pthread_attr_init(&attr[m]);               // アトリビュート初期化
33         pthread_create(&tid[m], &attr[m], thread, p); // スレッドを生成しスタート
34     }
35     // 各スレッドの終了を待ち, 求めた小計を合算する
36     int sum = 0;
37     for (int m=0; m<M; m++) {                     // 各スレッドについて
38         pthread_join(tid[m], NULL);                // 終了を待ち
39         sum += total[m];                           // 小計を合算する
40     }
41     printf("1+2+...+%d=%d\n", N*M, sum);
42     return 0;
43 }
```

スレッドの使用例 (4)

実行時間の計測結果

| M N M*N | スレッド数 (M) ・ データ件数 (M*N) | | | | | | | | | |
|-----------------|-------------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| | 10,000 | 5,000 | 3,333 | 2,500 | 2,000 | 1,666 | 1,428 | 1,250 | 1,111 | 1,000 |
| 経過時間 (s) | 1.881 | 0.980 | 0.657 | 0.493 | 0.406 | 0.339 | 0.335 | 0.332 | 0.319 | 0.312 |
| ユーザ CPU 時間 (s) | 1.879 | 1.953 | 1.959 | 1.958 | 2.009 | 2.011 | 2.244 | 2.462 | 2.679 | 2.846 |
| システム CPU 時間 (s) | 0.002 | 0.002 | 0.002 | 0.001 | 0.001 | 0.002 | 0.003 | 0.003 | 0.003 | 0.002 |



6 コアの Mac Pro で計測
(Hyper-Threading のお陰で 6 コアと 1 2 コアの間近な振舞)

第19章 TacOSのCPU仮想化

- 仮想 CPU (sp)
- プロセス番号 (pid)
- 状態 (stat)
- 優先度 (nice, enice)
- プロセステーブルのインデクス (idx)
- イベント用カウンタとセマフォ (evtCnt, evtSem)
- プロセスのアドレス空間 (memBase, memLen)
- プロセスの親子関係の情報 (parent, exitStat)
- オープン中のファイル一覧 (fds[])
- PCB リストの管理 (prev, next)
- スタックオーバーフローの検知 (magic)

TacOS の PCB (前半)

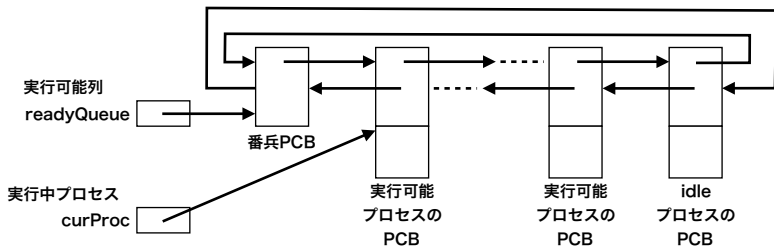
```
// ----- プロセス関連 -----  
#define PRC_MAX 10          // プロセスは最大 10 個  
#define P_KERN_STKSIZ 200   // プロセス毎のカーネルスタックのサイズ  
#define P_LOW_PRI 30000     // プロセスの最低優先度  
#define P_RUN 1             // プロセスは実行可能または実行中  
#define P_WAIT 2            // プロセスは待ち状態  
#define P_ZOMBIE 3          // プロセスは実行終了  
#define P_MAGIC 0xabcd      // スタックオーバーフロー検知に使用  
#define P_FILE_MAX 4        // プロセスがオープンできるファイルの最大数  
  
// プロセスコントロールブロック (PCB)  
// 優先度は値が小さいほど優先度が高い  
1 struct PCB {               // PCB を表す構造体  
2     int sp;                // コンテキスト (他の CPU レジスタと PSW は  
3                             // プロセスのカーネルスタックに置く)  
4     int pid;               // プロセス番号  
5     int stat;              // プロセスの状態  
6     int nice;              // プロセスの本来優先度  
7     int enice;             // プロセスの実質優先度 (将来用)  
8     int idx;               // この PCB のプロセステーブル上のインデクス
```

TacOS の PCB (後半)

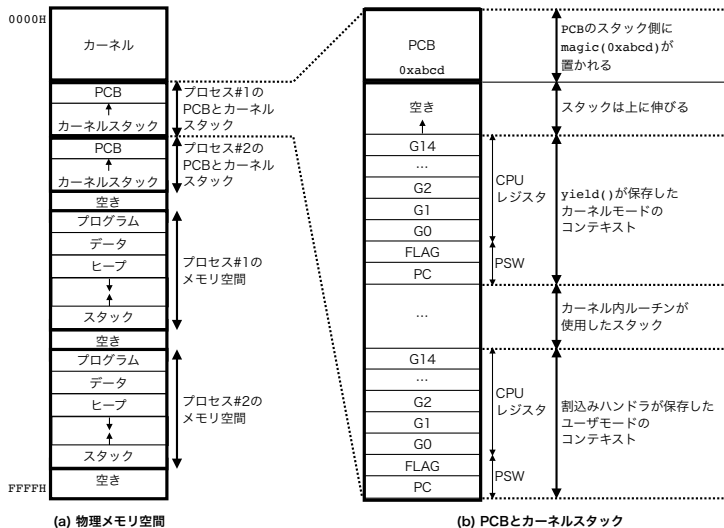
```
10 // プロセスのイベント用セマフォ
11 int evtCnt;           // カウンタ (>0:sleep 中, ==-1:wait 中, ==0:未使用)
12 int evtSem;           // イベント用セマフォの番号
13
14 // プロセスのアドレス空間 (text, data, bss, ...)
15 char[] memBase;       // プロセスのメモリ領域のアドレス
16 int memLen;           // プロセスのメモリ領域の長さ
17
18 // プロセスの親子関係の情報
19 PCB parent;           // 親プロセスへのポインタ
20 int exitStat;         // プロセスの終了ステータス
21
22 // オープン中のファイル一覧
23 int[] fds;            // オープン中のファイル一覧
24
25 // プロセスは重連結環状リストで管理
26 PCB prev;             // PCB リスト (前へのポインタ)
27 PCB next;             // PCB リスト (次へのポインタ)
28 int magic;            // スタックオーバーフローを検知
29 };
```

TacOS の実行可能列

- PCB の双方向環状リスト
- 優先度順にソート（curProc は実行中のプロセス）
- 末尾に idle プロセスが常駐



TacOS のメモリ配置



TacOS のタイマー管理プログラム

```
3 // タイマー割り込みハンドラ (10ms 毎に割り込みによって起動される)
4 interrupt tmrIntr() {
5     boolean disp = false;                                // ディスパッチの必要性
6
7     // 起きないといけないプロセスを起こしてまわる
8     for (int i=0; i<PRC_MAX; i=i+1) {
9         PCB p = procTbl[i];
10        if (p!=null && p.evtCnt>0) {                       // タイマー稼働中なら
11            int cnt = p.evtCnt - TICK;                     // 残り時間を計算
12            if (cnt<=0) {                                    // 時間が来たら
13                cnt = 0;                                     // タイマーを停止し
14                disp = iSemV(p.evtSem) || disp;             // プロセスを起こす
15            }
16            p.evtCnt = cnt;
17        }
18    }
19
20    if (disp) yield();                                     // 必要ならディスパッチ
21 }
```

TacOS のコンテキスト保存プログラム (yield())

```
1  _yield
2      ;--- G13(SP) 以外の CPU レジスタと FLAG をカーネルスタックに退避 ---
3      push    g0                ; FLAG の保存場所を準備する
4      push    g0                ; G0 を保存
5      ld      g0,flag           ; FLAG を上で準備した位置に保存
6      st      g0,2,sp           ;
7      push    g1                ; G1 を保存
8      push    g2                ; G2 を保存
   ...
17     push    g11               ; G11 を保存
18     push    fp                ; フレームポインタ (G12) を保存
19     push    usp               ; ユーザモードスタックポインタ (G14) を保存
20     ;
21     ;----- G13(SP) を PCB に保存 -----
22     ld      g1,_curProc       ; G1 ← curProc
23     st      sp,0,g1           ; [G1+0] は PCB の sp フィールド
24     ;
25     ;----- [curProc の magic フィールド] をチェック -----
26     ld      g0,30,g1          ; [G1+30] は PCB の magic フィールド
27     cmp     g0,#0xabcd        ; P_MAGIC と比較、一致しなければ
28     jnz     .stkOverflow      ; カーネルスタックがオーバーフローしている
```


TacOS のコンテスト復旧プログラム (dispatch())

```
1  _dispatch
2      ;----- 次に実行するプロセスの G13(SP) を復元 -----
3      ld      g0,_readyQueue  ; 実行可能列の番兵のアドレス
4      ld      g0,28,g0        ; [G0+28] は PCB の next フィールド (先頭の PCB)
5      st      g0,_curProc     ; 現在のプロセス (curProc) に設定する
6      ld      sp,0,g0         ; PCB から SP を取り出す
7      ;
8      ;----- G13(SP) 以外の CPU レジスタを復元 -----
9      pop     usp             ; ユーザモードスタックポインタ (G14) を復元
10     pop     fp              ; フレームポインタ (G12) を復元
11     pop     g11             ; G11 を復元
12     pop     g10             ; G10 を復元
13     pop     g9              ; G9 を復元
14     ...
15
21     pop     g1              ; G1 を復元
22     pop     g0              ; G0 を復元
23     ;
24     ;----- PSW(FLAG と PC) を復元 -----
25     reti                  ; RETI 命令で一度に POP して復元する
```

練習問題

- 次の言葉の意味を説明しなさい.
 - 時分割多重
 - コンテキストスイッチ
 - Dispatch (ディスパッチ)
 - Preemption (プリエンプション)
 - プロセスの状態
 - プロセスの状態遷移
 - RETI 命令
 - PCB
 - 待ち行列
 - 実行可能列
 - スレッド
 - カーネルスレッド
 - ユーザスレッド
 - One-to-One Model
 - Many-to-One Model
 - Many-to-Many Model
- POSIX スレッドについて調査しなさい.