

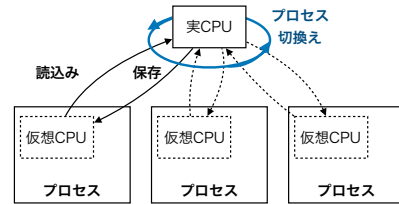
## オペレーティングシステム 第3章 CPUの仮想化

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

CPUの仮想化

1 / 36

### 時分割多重によるCPUの仮想化

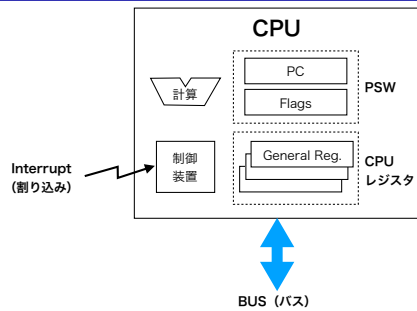


- 時分割多重：CPUが実行するプロセスを次々切替える。
- コンテキストスイッチ：CPUが実行するプロセスを切替えること。
- ディスパッチ：プロセスにCPUを割り付ける。(実行開始)
- ディスパッチャ：ディスパッチするプログラムのこと。

CPUの仮想化

2 / 36

### CPUの構造 (参考)

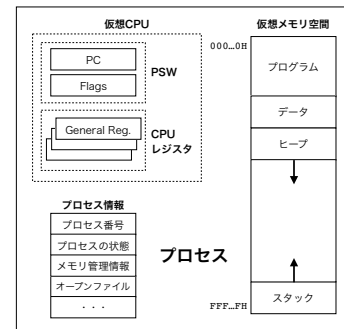


- コンテキスト = PSW + CPU レジスタ
- コンテキストを保存・ロードして次のプロセスに
- コンテキストスイッチ

CPUの仮想化

3 / 36

### プロセスの構造 (参考)

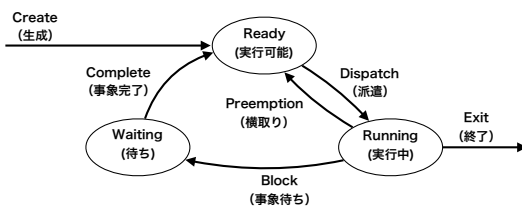


- 仮想CPUにコンテキストを保存

CPUの仮想化

4 / 36

### 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

CPUの仮想化

5 / 36

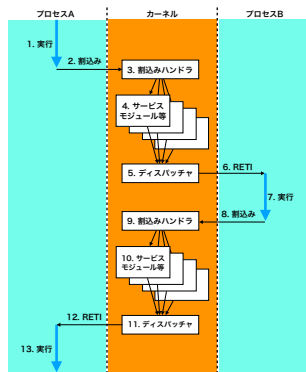
### プロセス切換えの原因

- 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
  - クォンタムタイムを使い切ったプロセスはPreemptionする

CPUの仮想化

6 / 36

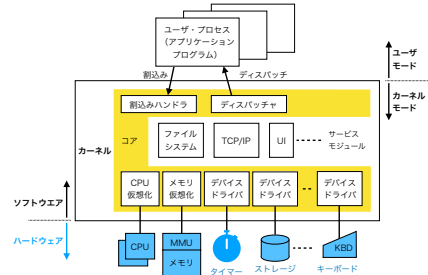
## プロセスの切換え手順



CPU の仮想化

7 / 36

## オペレーティングシステムの構造 (参考)

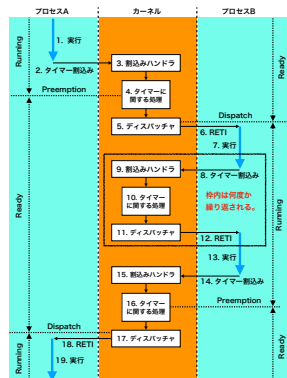


- 割込みハンドラ
- サービスモジュール
- ディスパッチャ

CPU の仮想化

8 / 36

## プロセスの切換えの例

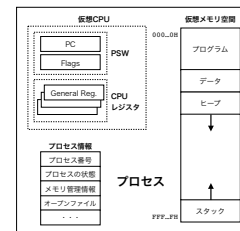


CPU の仮想化

9 / 36

## 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



CPU の仮想化

10 / 36

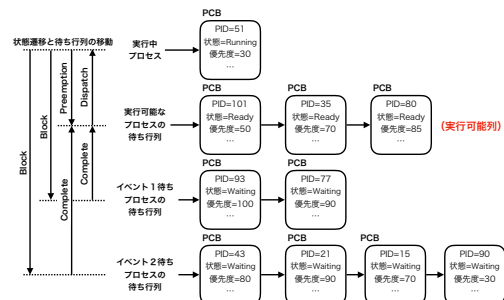
## PCB の内容

- 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

CPU の仮想化

11 / 36

## 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

CPU の仮想化

12 / 36

## スレッド (Thread)

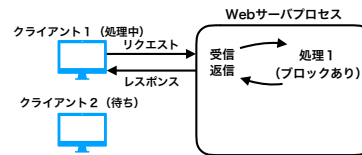
- CPU の仮想化によりマルチプログラミングが可能になった。
- プロセスが並行 (Concurrent) に実行できる。
- プロセスは一つの仮想 CPU を持っている。
- プロセスはコンピュータを仮想化したもの。
  - CPU が一つしかないコンピュータを仮想化している。
  - CPU を複数持つ SMP を仮想化するには不十分。
- 一つのプロセスが複数の仮想 CPU をもつモデルを導入する。
- プロセスが処理の流れスレッドを複数持つことができる。

CPU の仮想化

13 / 36

## 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



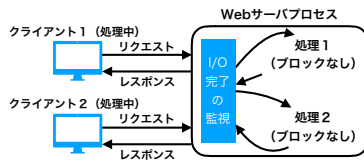
- 処理は順番に処理される。
- 前の処理が終わるまでクライアントは待たされる。

CPU の仮想化

14 / 36

## 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



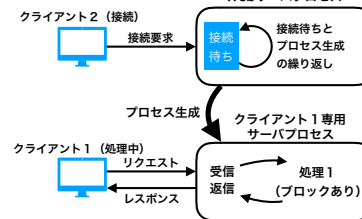
- 工夫すると並列して処理することも可能
- しかし、プログラミングが難しい。
- しかし、SMP が活かせない。

CPU の仮想化

15 / 36

## 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)



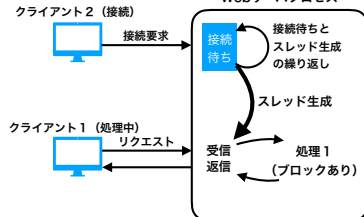
- クライアント毎にプロセスを起動 (fork()) する。
- プログラミングは易しい。
- しかし、処理が重いし、プロセス間の情報共有の効率が悪い。

CPU の仮想化

16 / 36

## 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)

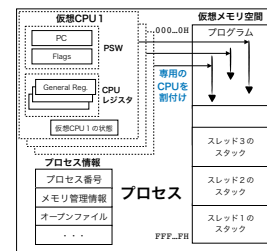


- クライアント毎にスレッドを起動する。
- プロセスの起動より 10~100 倍速い。
- スレッド間は情報を共有しやすい。
- プログラミングは少し難しい。

CPU の仮想化

17 / 36

## スレッドの形式 (1) -カーネルスレッド-

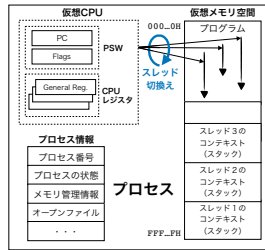


- プロセスが複数の仮想 CPU を持つ。

CPU の仮想化

18 / 36

## 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。

CPU の仮想化

19 / 36

## 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。

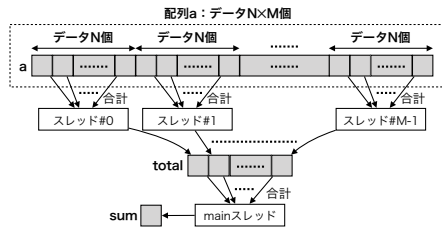
- *One-to-One Model*  
一つのスレッドを一つのカーネルスレッドで実行する。
- *Many-to-One Model*  
複数のスレッドを一つのカーネルスレッドで実行する。
- *Many-to-Many Model*  
複数のスレッドを複数のカーネルスレッドで実行する。

CPU の仮想化

20 / 36

## 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)



CPU の仮想化

21 / 36

## 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[N*M]; // 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計
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 }

```

CPU の仮想化

22 / 36

## 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++) {
28         Args *p = malloc(sizeof(Args)); // 各スレッドについて
29         p->no = m; // 引数領域を確保
30         p->min = N*m; // m 番目のスレッド
31         p->max = N*(m+1); // 担当範囲下限
32         pthread_attr_init(&attr[m]); // 担当範囲上限
33         pthread_create(&tid[m], &attr[m], thread, p); // アトリビュート初期化
34     } // スレッドを生成しスタート
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+...+M*d=%d\n", N*M, sum);
42     return 0;
43 }

```

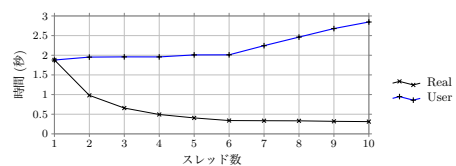
CPU の仮想化

23 / 36

## スレッドの使用例 (4)

実行時間の計測結果

M	スレッド数 (M)・データ件数 (N*M)									
	1	2	3	4	5	6	7	8	9	10
N	10,000	5,000	3,333	2,500	2,000	1,666	1,428	1,250	1,111	1,000
N*M	10,000	10,000	9,999	10,000	10,000	9,996	9,996	10,000	9,999	10,000
移動時間 (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

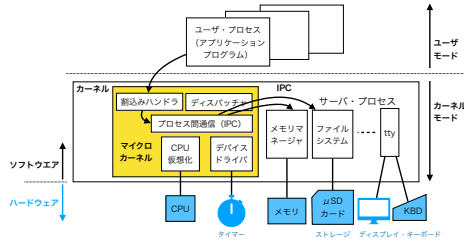


6 コアの Mac Pro で計測  
(Hyper-Threading のお陰で6 コアと12 コアの間的な振舞)

CPU の仮想化

24 / 36

## 実装例

第19章  
TacOSのCPU仮想化

CPUの仮想化

25 / 36

## TacOSのPCB

- 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ックス (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)

CPUの仮想化

26 / 36

## 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 {
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ックス
```

CPUの仮想化

27 / 36

## TacOSのPCB (後半)

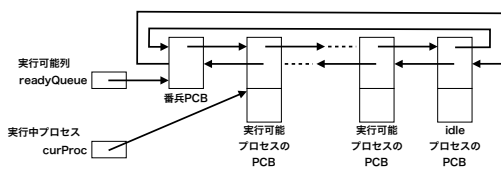
```
10 // プロセスのイベント用セマフォ
11 int evtCnt; // カウンタ (>0:sleep 中, ==-1:wait 中, ==0:未使用)
12 int evtSem; // イベント用セマフォの番号
13
14 // プロセスのアドレス空間 (text, data, bss, ...)
15 char[] memBase; // プロセスのメモリ領域のアドレス
16 int memLen; // プロセスのメモリ領域の長さ
17
18 // プロセスの親子関係の情報
19 PCB parent; // 親プロセスへのポインタ
20 int exitStat; // 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 };
```

CPUの仮想化

28 / 36

## TacOSの実行可能列

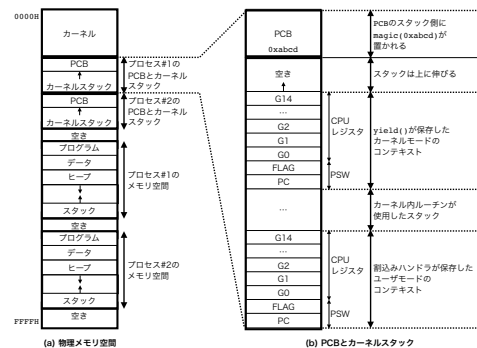
- PCBの双方向環状リスト
- 優先度順にソート (curProcは実行中のプロセス)
- 末尾に idle プロセスが常駐



CPUの仮想化

29 / 36

## TacOSのメモリ配置



CPUの仮想化

30 / 36

## 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) {
11            int cnt = p.evtCnt - TICK;      // タイマー稼働中なら
12            if (cnt<=0) {                  // 残り時間を計算
13                cnt = 0;                  // 時間が来たら
14                disp = iSemV(p.evtSem) || disp; // タイマーを停止し
15                // プロセスを起こす
16                p.evtCnt = cnt;
17            }
18        }
19        if (disp) yield();                // 必要ならディスパッチ
20    }
21 }

```

CPU の仮想化

31 / 36

## 参考 (普通の関数)

## C--言語の void 型関数

```

1 void func();
2
3 void sFunc() {
4     for (int i=0; i<10; i=i+1) {
5         func(); // 関数呼び出し
6     }
7 }

```

```

1 .sFunc  PUSH  FP
2        LD   FP,SP
3        PUSH G3
4        LD   G3,#0
5
6        CMP  G3,#10
7        JGE  .L2
8        CALL .func
9        LD   G0,G3
10       ADD  G0,#1
11       LD   G3,G0
12       JMP  .L1
13
14 .L2    POP   G3
15       POP   FP
16       RET

```

CPU の仮想化

32 / 36

## 参考 (interrupt 関数)

## C--言語の interrupt 型関数

```

1 void func();
2
3 interrupt iFunc() {
4     for (int i=0; i<10; i=i+1) {
5         func(); // 関数呼び出し
6     }
7 }

```

```

1 .iFunc  PUSH  G0
2        PUSH  G1
3        PUSH  G2
4        PUSH  FP
5        LD   FP,SP
6        PUSH  G3
7        LD   G3,#0
8
9        .L1  CMP  G3,#10
10       JGE  .L2
11       CALL .func
12       LD   G0,G3
13       ADD  G0,#1
14       LD   G3,G0
15       JMP  .L1
16
17       .L2  POP   G3
18       POP   FP
19       POP   G2
20       POP   G1
21       POP   G0
22       RETI

```

CPU の仮想化

33 / 36

## 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
9
10 ...
11
12 push  g11         ; G11 を保存
13 push  fp          ; フレームポインタ (G12) を保存
14 push  usp         ; ユーザーモードスタックポインタ (G14) を保存
15
16 ;
17 ;----- G13(SP) を PCB に保存 -----
18 ld    g1,_curProc ; G1 ← curProc
19 st    sp,0,g1     ; [G1+0] は PCB の sp フィールド
20
21 ;
22 ;----- [curProc の magic フィールド] をチェック -----
23 ld    g0,30,g1    ; [G1+30] は PCB の magic フィールド
24 cmp   g0,0xabcd   ; P_MAGIC と比較、一致しなければ
25 jnz   .stkOverflow ; カーネルスタックがオーバーフローしている
26
27
28

```

CPU の仮想化

34 / 36

## TacOS のコンテスト復旧プログラム (dispatch())

\_dispatch は \_yield (28 行) の直後にある. (つながっている!)

```

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           ; ユーザー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 ...
16
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

```

CPU の仮想化

35 / 36

## 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。

CPU の仮想化

36 / 36