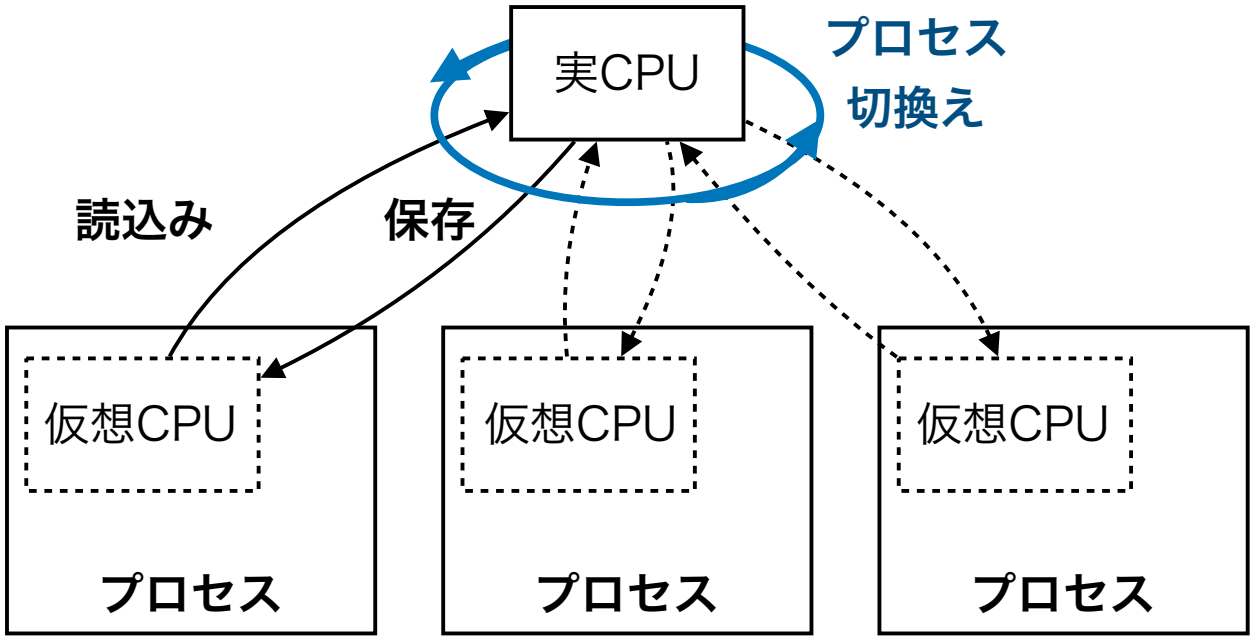
オペレーティングシステム
第３章 CPU の仮想化

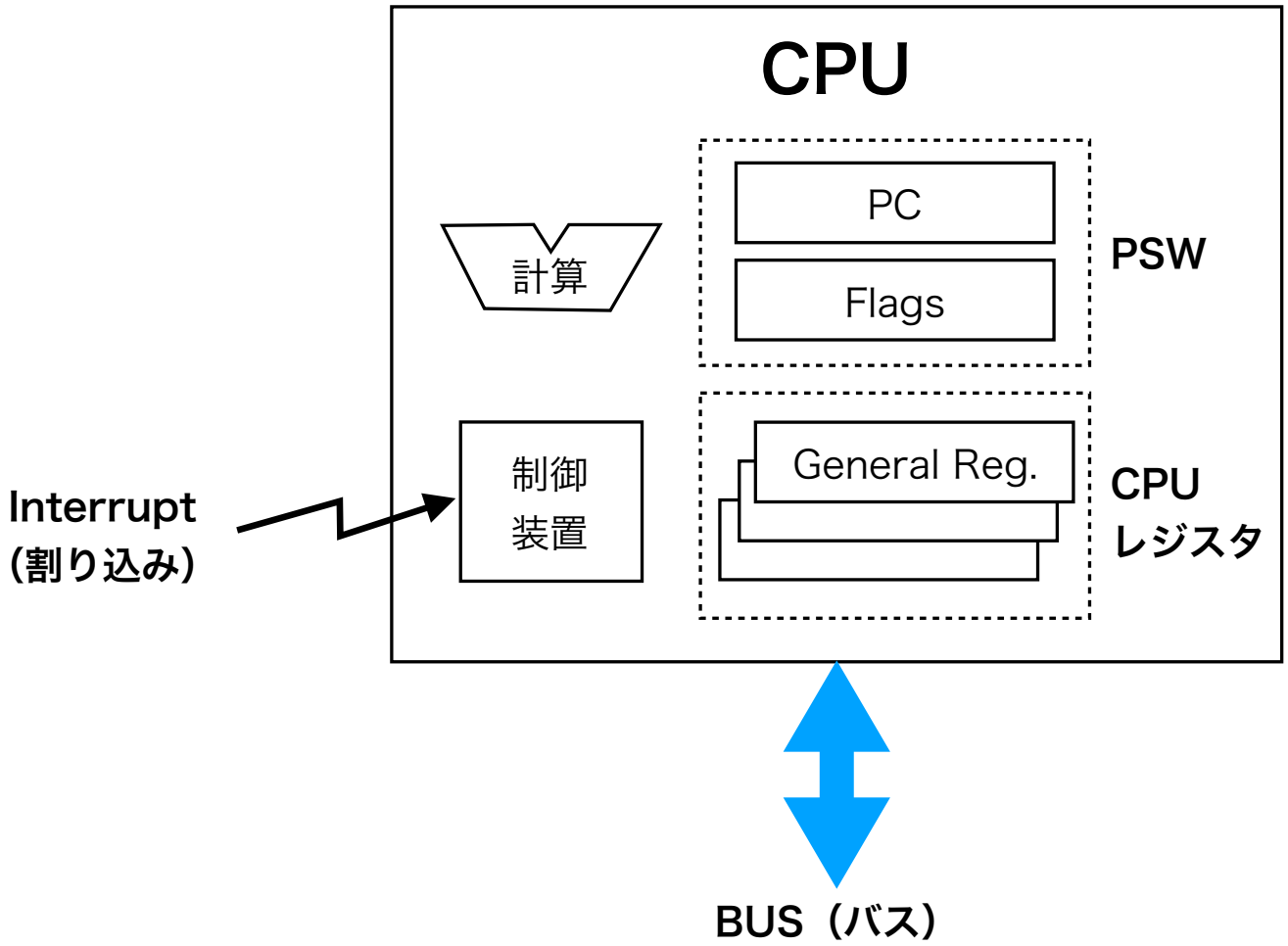
https://github.com/tctsigemura/OSTextBook

時分割多重による CPU の仮想化

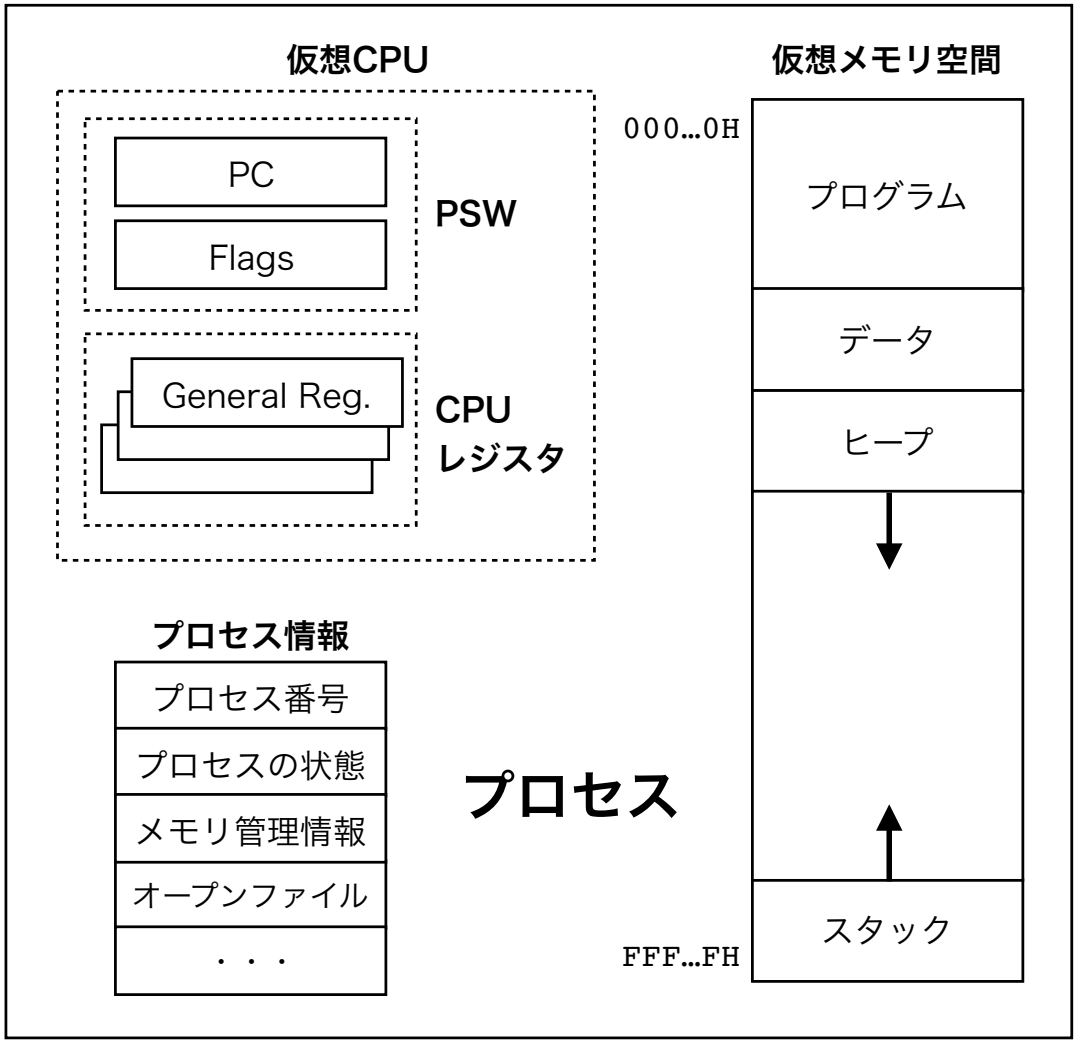


- 時分割多重：CPU が実行するプロセスを次々切換える。
- ディスパッチ：プロセスに CPU を割り付ける。(実行開始)
- ディスパッチャ：ディスパッチするプログラムのこと。

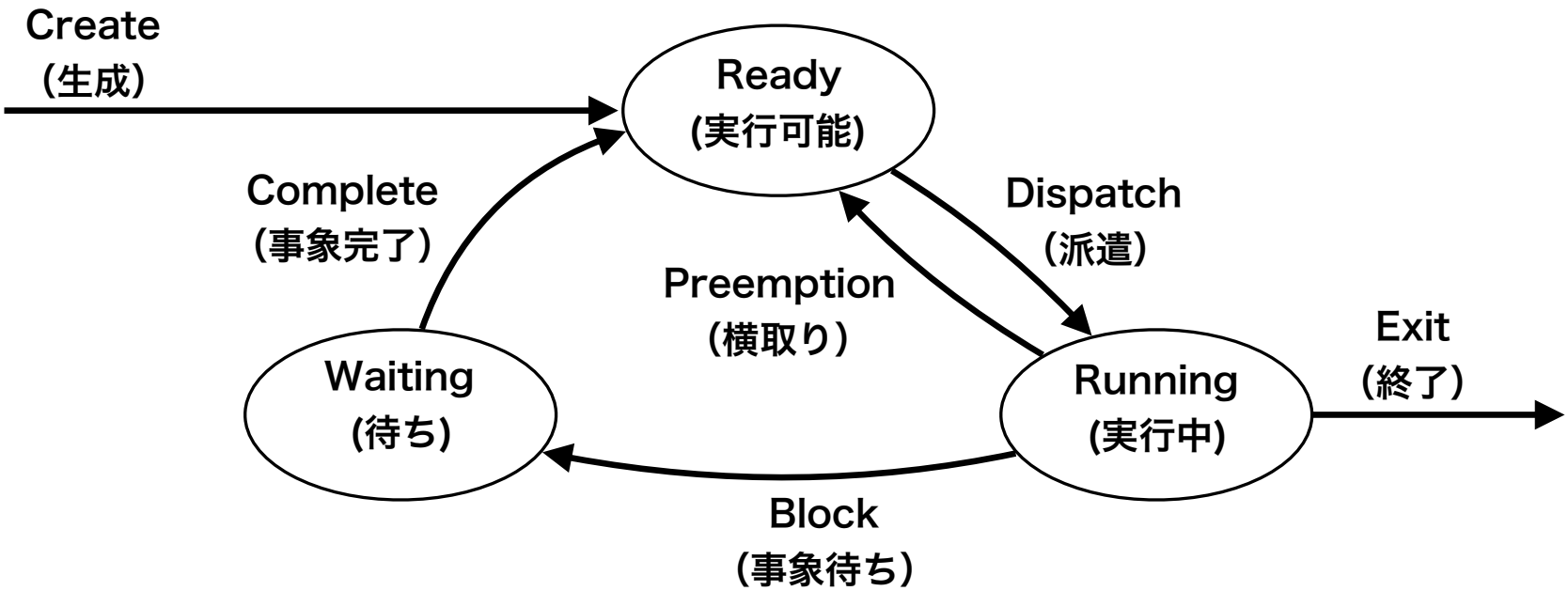
CPU の構造（参考）


CPU
PC
Flags
PSW
計算
制御
装置
General Reg.
CPU
レジスタ
Interrupt
(割り込み)
BUS（バス）

プロセスの構造（参考）

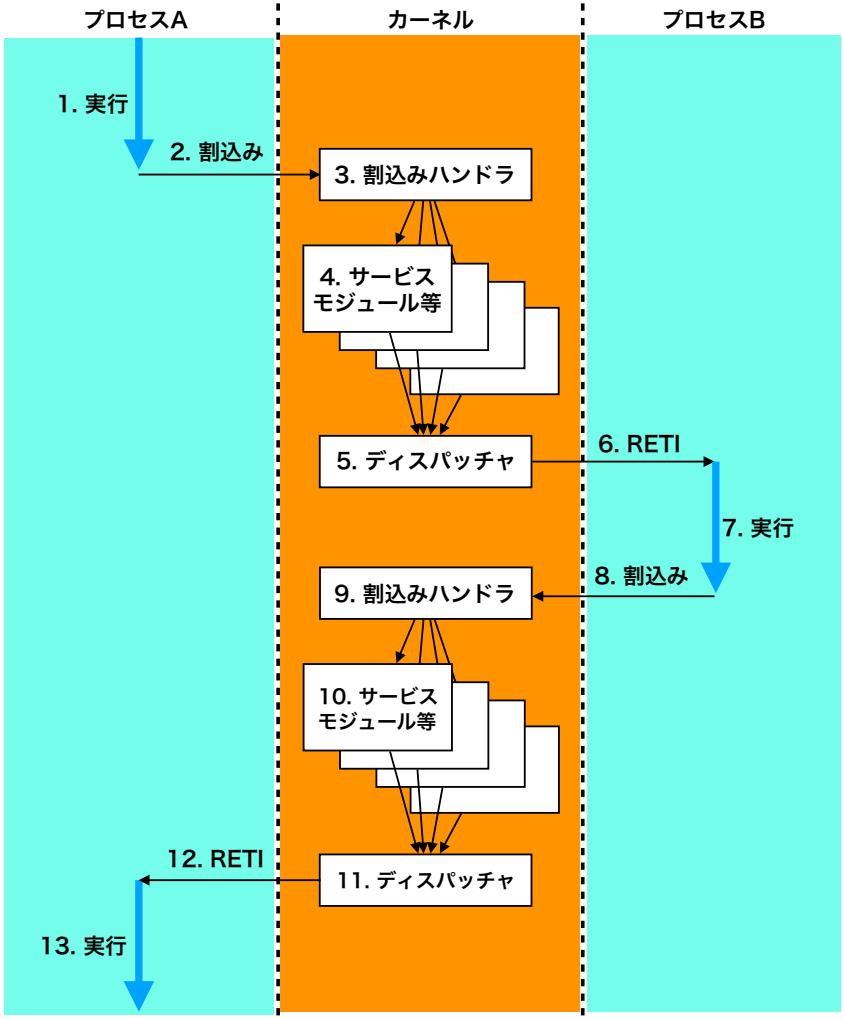

仮想CPU
PC
Flags
PSW
General Reg.
CPU
レジスタ
プロセス情報
プロセス番号
プロセスの状態
メモリ管理情報
オープンファイル
・・・
プロセス
仮想メモリ空間
000…0H
プログラム
データ
ヒープ
スタック
FFF…FH

プロセスの状態遷移

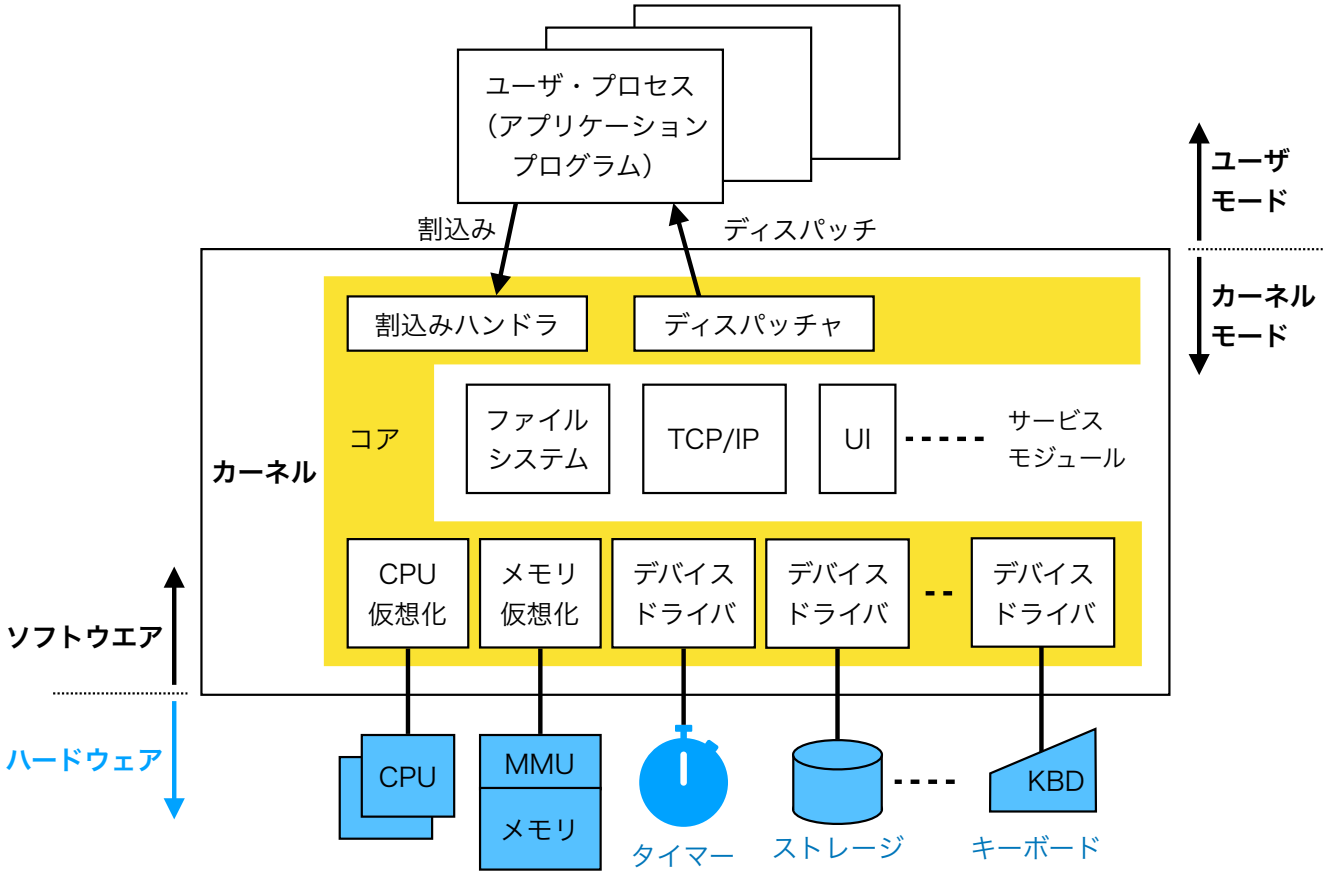


- 基本的な三つの状態
- 六つの状態遷移

プロセスの切換え

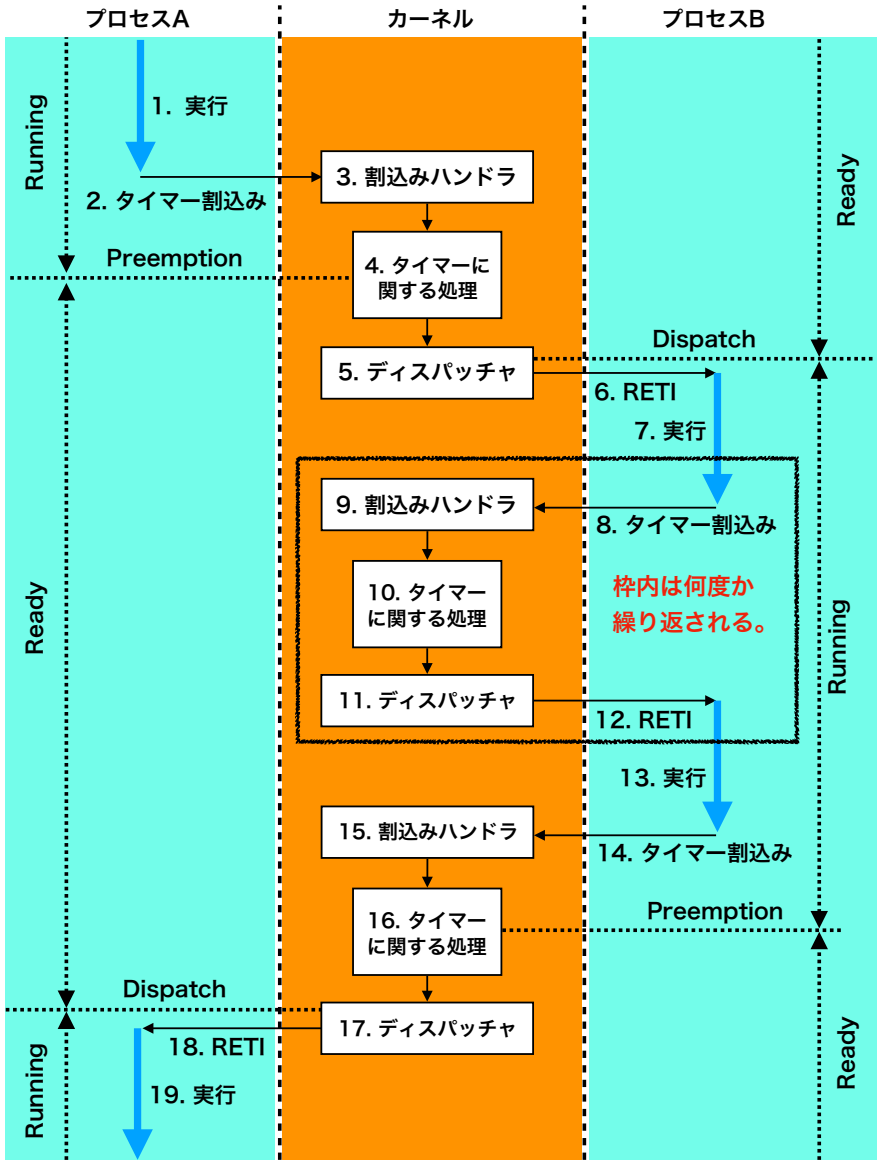

プロセスA
カーネル
プロセスB
1. 実行
2. 割込み
3. 割込みハンドラ
4. サービス
モジュール等
5. ディスパッチャ
6. RETI
7. 実行
8. 割込み
9. 割込みハンドラ
10. サービス
モジュール等
11. ディスパッチャ
12. RETI
13. 実行

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

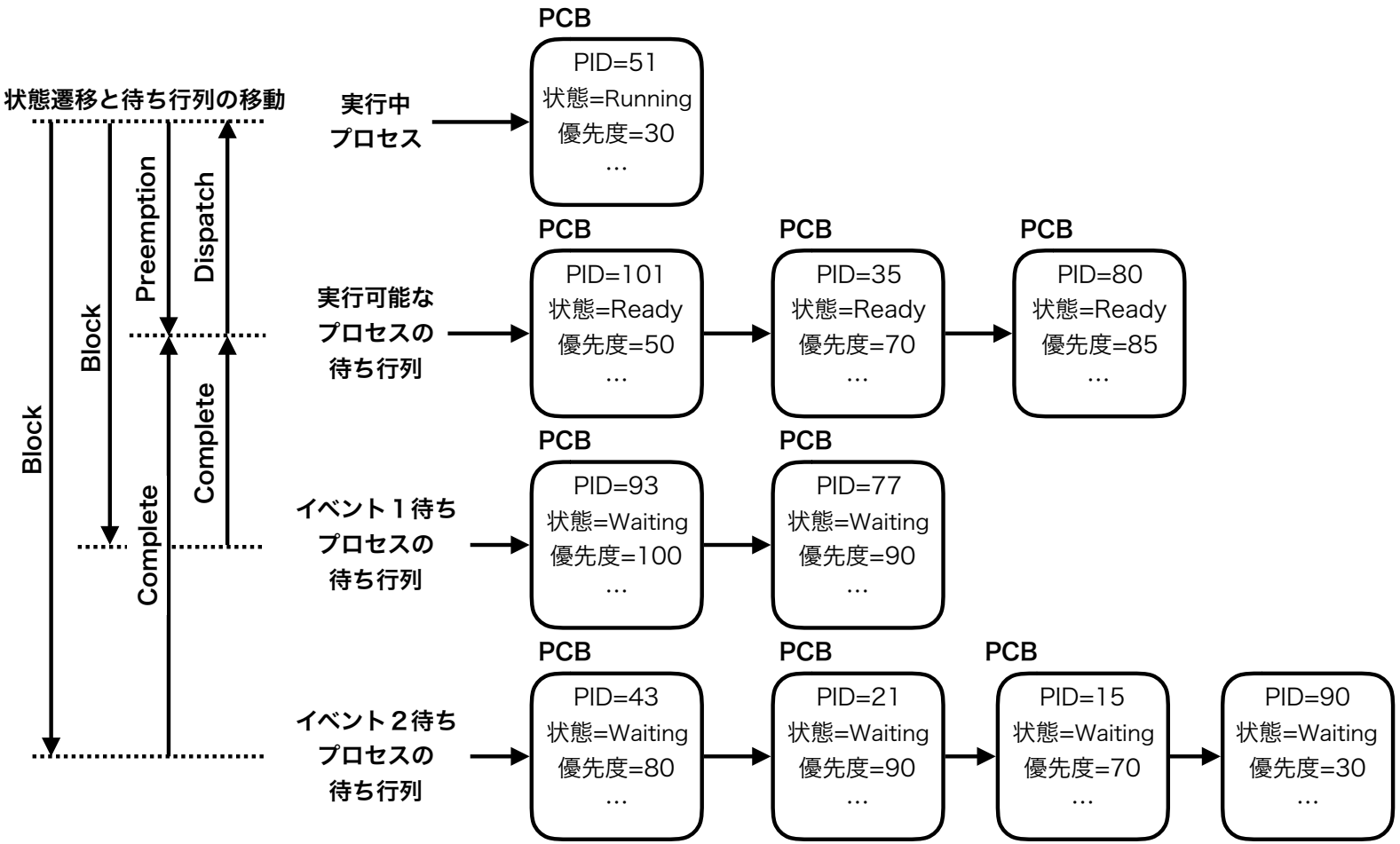
プロセスの切換えの例


プロセスA
カーネル
プロセスB
Running
1. 実行
2. タイマー割込み
Preemption
Ready
3. 割込みハンドラ
4. タイマーに関する処理
5. ディスパッチャ
Dispatch
Ready
6. RETI
7. 実行
9. 割込みハンドラ
8. タイマー割込み
10. タイマーに関する処理
枠内は何度か繰り返される。
11. ディスパッチャ
12. RETI
Running
13. 実行
15. 割込みハンドラ
14. タイマー割込み
16. タイマーに関する処理
Preemption
Ready
Dispatch
17. ディスパッチャ
18. RETI
Running
19. 実行

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


状態遷移と待ち行列の移動
Block
Block
Preemption
Dispatch
Complete
Complete
実行中
プロセス
PCB
PID=51
状態=Running
優先度=30
…
実行可能な
プロセスの
待ち行列
PCB
PID=101
状態=Ready
優先度=50
…
PCB
PID=35
状態=Ready
優先度=70
…
PCB
PID=80
状態=Ready
優先度=85
…
イベント１待ち
プロセスの
待ち行列
PCB
PID=93
状態=Waiting
優先度=100
…
PCB
PID=77
状態=Waiting
優先度=90
…
イベント２待ち
プロセスの
待ち行列
PCB
PID=43
状態=Waiting
優先度=80
…
PCB
PID=21
状態=Waiting
優先度=90
…
PCB
PID=15
状態=Waiting
優先度=70
…
PCB
PID=90
状態=Waiting
優先度=30
…

TacOS の PCB

- 仮想 CPU（SP）
- プロセス番号（pid）
- 状態 (stat)
- 優先度（nice, enaice）
- 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）

TaC の PCB（前半）

```
#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

// 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
struct PCB {                // PCB を表す構造体
  int sp;                   // コンテキスト (他の CPU レジスタと PSW は
                            //                プロセスのカーネルスタックに置く)
  int pid;                  // プロセス番号
  int stat;                 // プロセスの状態
  int nice;                 // プロセスの本来優先度
  int enice;                // プロセスの実質優先度 (将来用)
  int idx;                  // この PCB のプロセステーブル上のインデクス

  // プロセスのイベント用セマフォ
  int evtCnt;               // カウンタ (>0:sleep 中, ==-1:wait 中, ==0:未使用)
  int evtSem;               // イベント用セマフォの番号
```

TaC の PCB（後半）

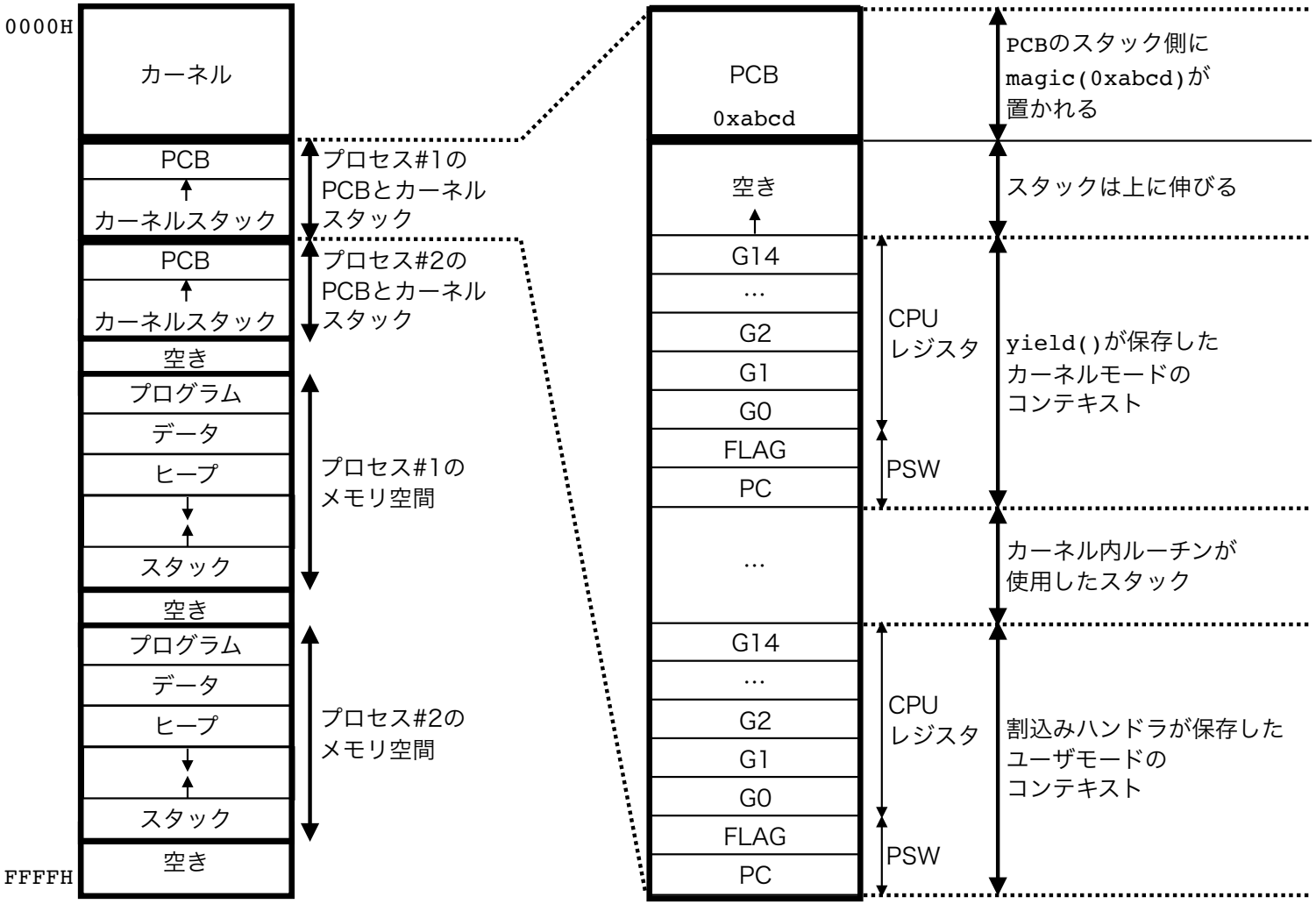
```
  // プロセスのアドレス空間 (text, data, bss, ...)
  char[] memBase;          // プロセスのメモリ領域のアドレス
  int memLen;              // プロセスのメモリ領域の長さ

  // プロセスの親子関係の情報
  PCB parent;              // 親プロセスへのポインタ
  int exitStat;            // プロセスの終了ステータス

  // オープン中のファイル一覧
  int[] fds;               // オープン中のファイル一覧

  // プロセスは重連結環状リストで管理
  PCB prev;                // PCB リスト (前へのポインタ)
  PCB next;                // PCB リスト (次へのポインタ)
  int magic;               // スタックオーバーフローを検知
};
```

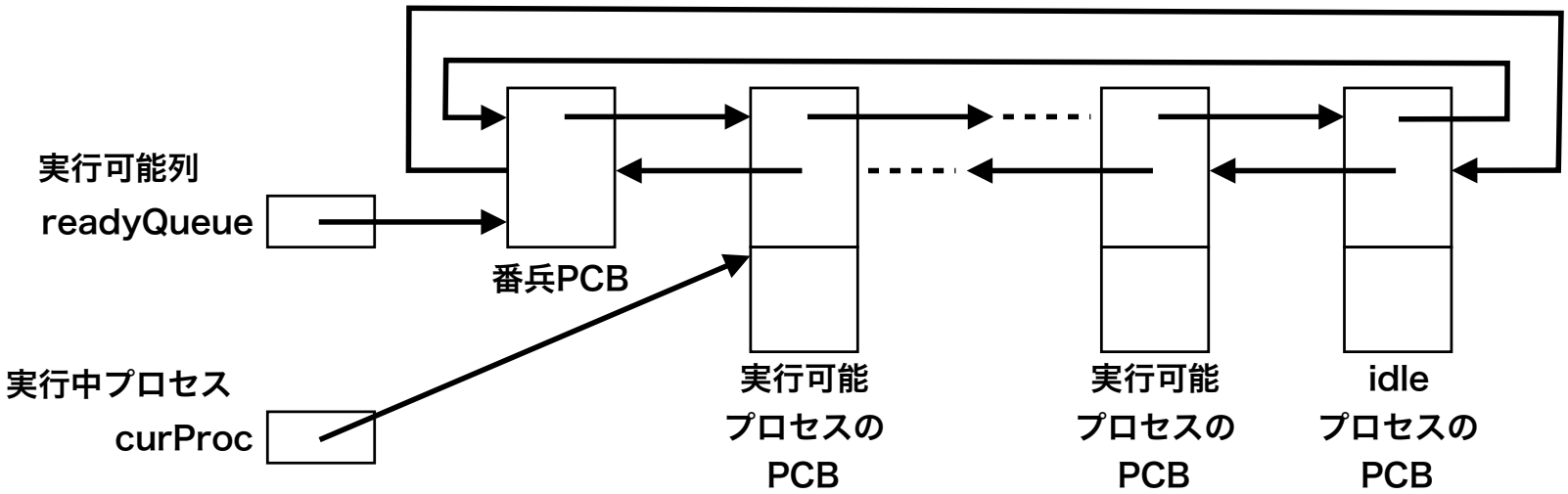
TacOS のメモリ配置



(a) 物理メモリ空間　　(b) PCBとカーネルスタック

TacOS の実行可能列

- `yield`
- `dispatch`
- 実行可能列



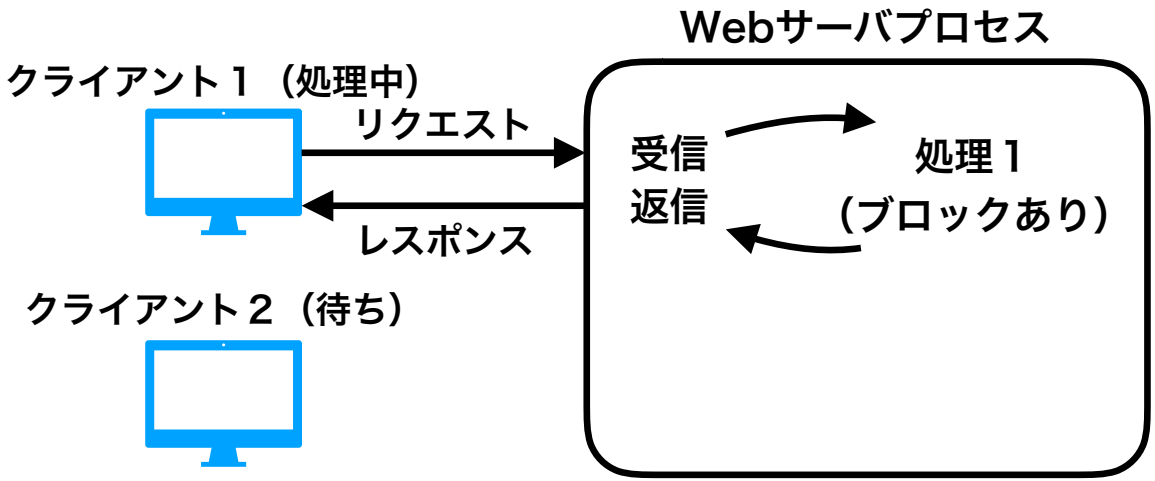
プロセス切換えプログラム（yield()）

```
; 現在のプロセス (curProc) が CPU を解放する. その後, 新プロセスへディスパッチする.
_yield                              ; 高級言語からは yield() 関数として呼出す.
        ;--- G13(SP) 以外の CPU レジスタと FLAG をカーネルスタックに退避 ---
        push    g0                  ; FLAG の保存場所を準備する
        push    g0                  ; G0 を保存
        ld      g0,flag             ; FLAG を上で準備した位置に保存
        st      g0,2,sp             ;
        push    g1                  ; G1 を保存
        push    g2                  ; G2 を保存
        push    g3                  ; G3 を保存
        ...
        ...                         ; G4 から G10 も同様に保存する
        ...
        push    g11                 ; G11 を保存
        push    fp                  ; フレームポインタ (G12) を保存
        push    usp                 ; ユーザモードスタックポインタ (G14) を保存
        ;------- G13(SP) を PCB に保存 --------------------------
        ld      g1,_curProc         ; G1 <- curProc
        st      sp,0,g1             ; [G1+0] は PCB の sp フィールド
        ;------- [curProc の magic フィールド] をチェック ---------
        ld      g0,30,g1            ; [G1+30] は PCB の magic フィールド
        cmp     g0,#0xabcd          ; P_MAGIC と比較、一致しなければ
        jnz     .stkOverFlow        ; カーネルスタックがオーバーフローしている
```

プロセスの切換えプログラム（dispatch()）

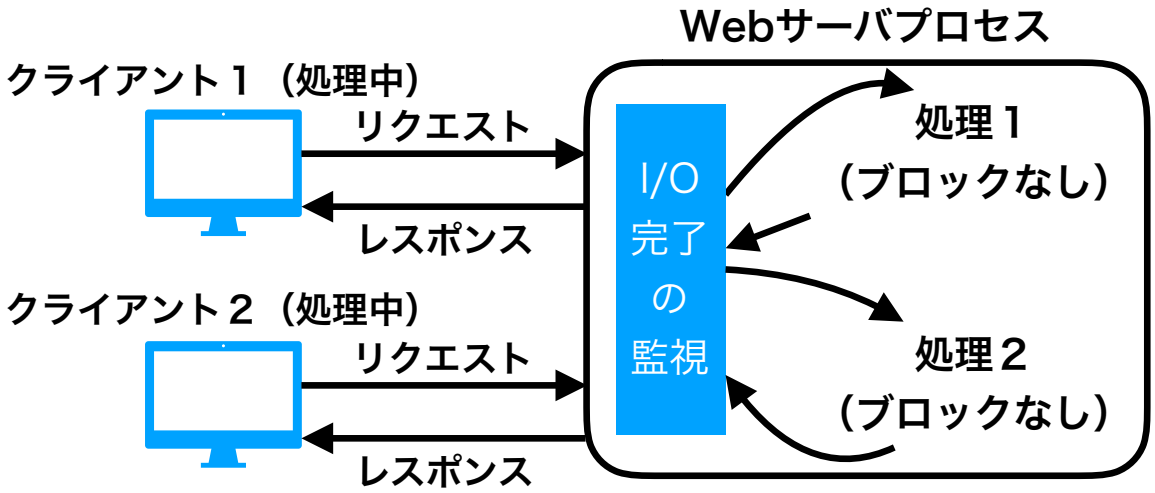
```
;
; 最優先のプロセス(readyQueue の先頭プロセス)へディスパッチする.
_dispatch                       ; 高級言語からは dispatch() 関数として呼出す.
        ;-------- 次に実行するプロセスの G13(SP) を復元 ----------
        ld      g0,_readyQueue  ; 実行可能列の番兵のアドレス
        ld      g0,28,g0        ; [G0+28] は PCB の next フィールド(先頭の PCB)
        st      g0,_curProc     ; 現在のプロセス(curProc)に設定する
        ld      sp,0,g0         ; PCB から SP を取り出す
        ;-------- G13(SP) 以外の CPU レジスタを復元 -----------
        pop     usp             ; ユーザモードスタックポインタ(G14)を復元
        pop     fp              ; フレームポインタ(G12)を復元
        pop     g11             ; G11 を復元
        ...
        ...                     ; G10 から G4 も同様に復元する
        ...
        pop     g3              ; G3 を復元
        pop     g2              ; G2 を復元
        pop     g1              ; G1 を復元
        pop     g0              ; G0 を復元
        ;------------- PSW(FLAG と PC) を復元 -----------------
        reti                    ; RETI 命令で一度に POP して復元する
```

マルチプログラミングを用いない Web サーバ



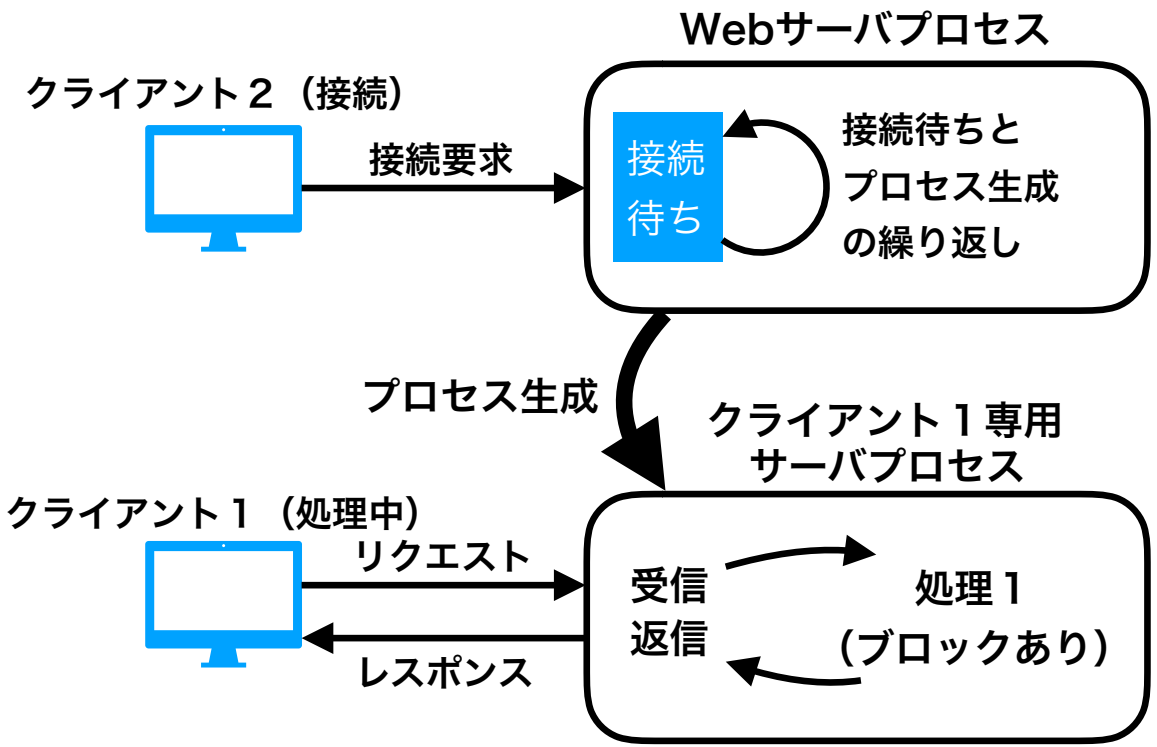
(a) 最も基本的なWebサーバのモデル

マルチプログラミングを用いない Web サーバ



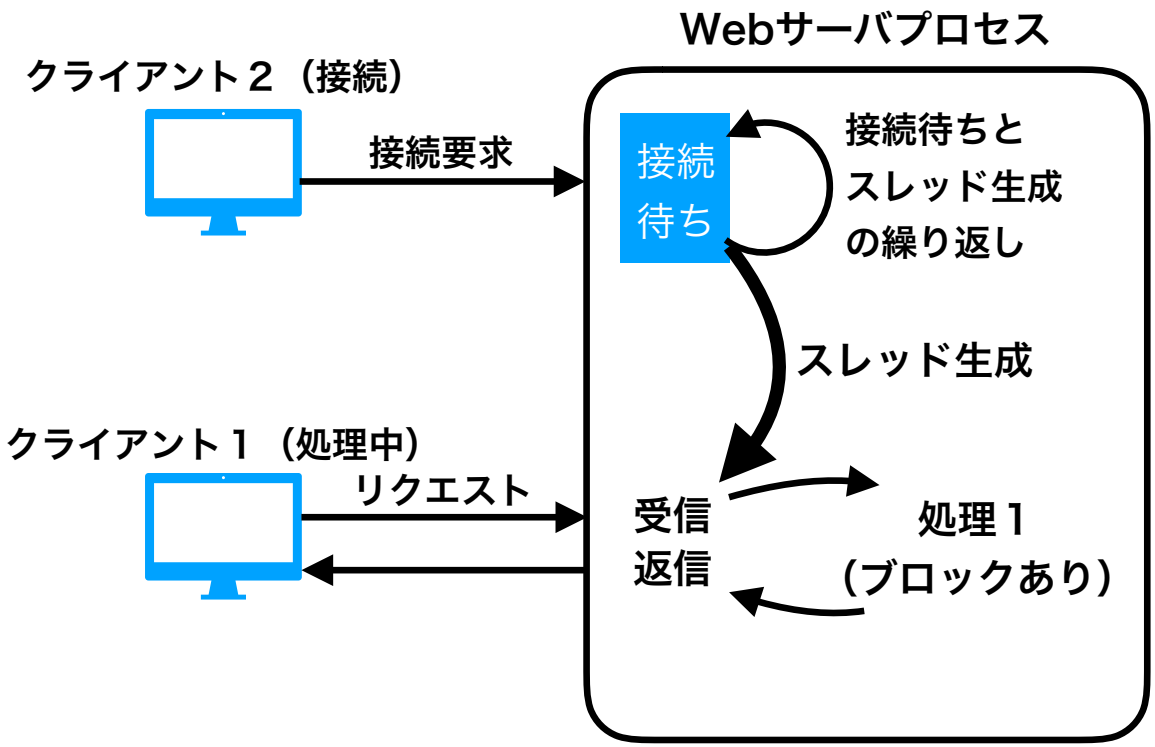
(b) 改良したWebサーバのモデル

マルチプログラミングを用いる Web サーバ



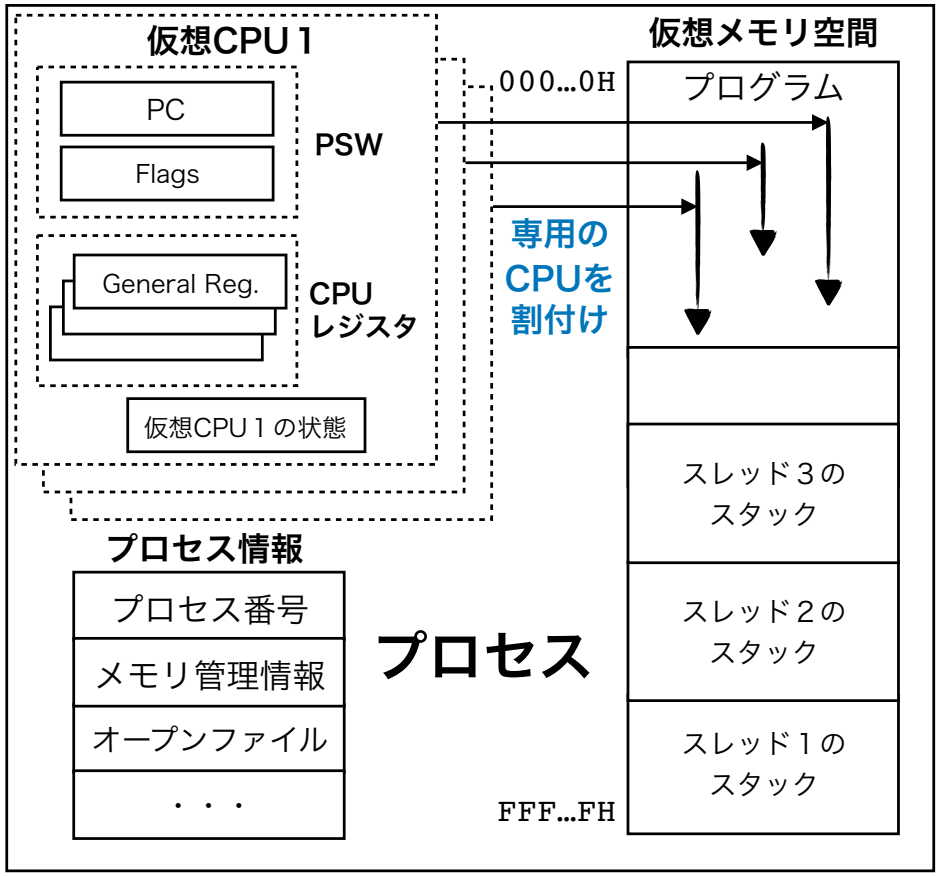
(a) マルチプロセスにしたWebサーバのモデル

マルチプログラミングを用いる Web サーバ



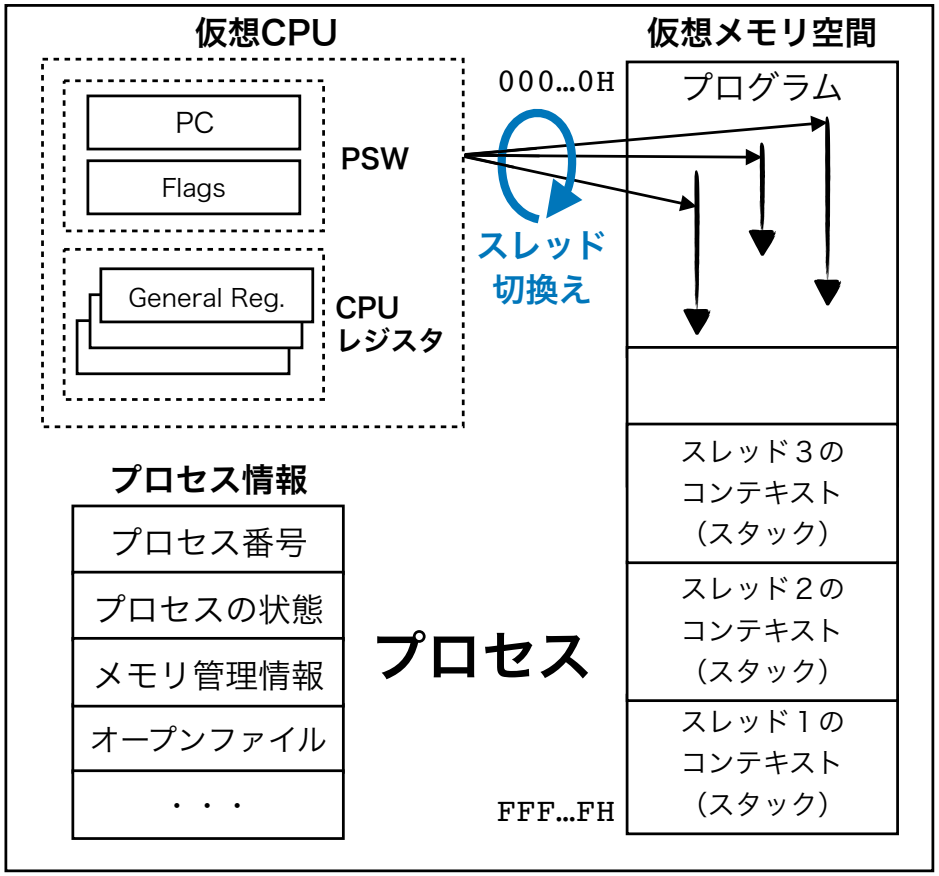
(b) マルチスレッドにしたWebサーバのモデル

ユーザスレッドとカーネルスレッド



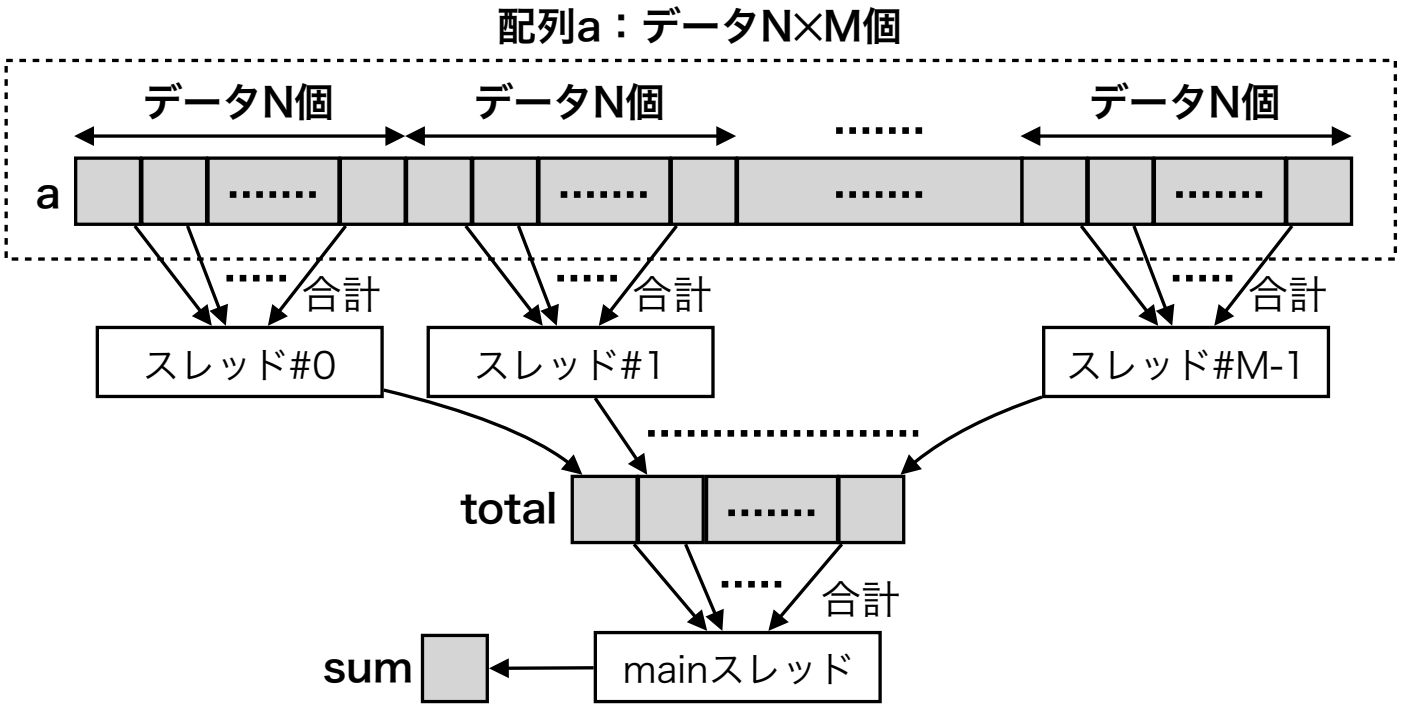
(a) カーネルスレッド

ユーザスレッドとカーネルスレッド



(b) ユーザスレッド

M 個のスレッドで手分けをして合計を計算する様子



M 個のスレッドで合計を計算する（前半）

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N 1000                                  // 1 スレッドの担当データ数
#define M 10                                    // スレッド数
pthread_t tid[M];                               // M 個のスレッドのスレッド ID
pthread_attr_t attr[M];                         // M 個のスレッドの属性
int a[M*N];                                     // このデータの合計を求める
int total[M];                                   // 各スレッドの求めた部分和
typedef struct { int no, min, max; } Args;      // スレッドに渡す引数の型定義

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
  Args *args = arg;                             // m 番目のスレッド
  int sum = 0;                                  // 合計を求める変数
  for (int i=args->min; i<args->max; i++) {     // a[N*m ... (N+1)*m] の
    sum += a[i];                                //    合計を sum に求める.
  }
  total[args->no]=sum;                          // 担当部分の合計を記録
  return NULL;                                  // スレッドを正常終了する
}
```

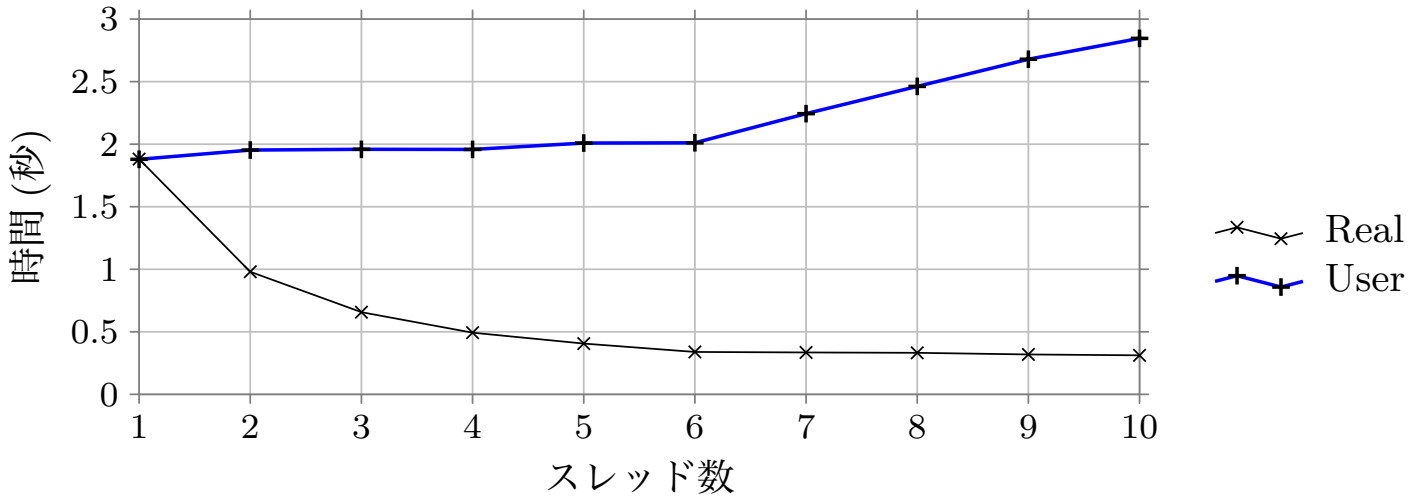
M 個のスレッドで合計を計算する（後半）

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

M 個のスレッドで合計を計算する（後半）

スレッド数 (M)・データ件数 (M*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
M*N	10,000	10,000	9,999	10,000	10,000	9,996	9,996	10,000	9,999	10,000
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 12 コアの中間的な振舞)