システムソフトウェア

2021年度

第4回 (10/14)

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

http://titech-os.github.io

e-mail: takuo∅c.titech.ac.jp

本日のメニュー

- メモリ管理(2)
  - 仮想記憶
  - xv6のプロセスとメモリ管理(1)

仮想記憶

仮想記憶(Virtual Memory)

- 物理メモリ量という制約を緩和するための抽象化技術
  - 各プロセスが必要とするメモリ量，あるいはそれらの合計は物理メモリ量を超えることがある．
  - そこで，二次記憶を用いて見かけ上のメモリ量を増加させる

- xv6では二次記憶を使った仮想記憶は実装されていない

スワップ領域(swap area)

- 物理メモリに入りきらないメモリ領域を格納するための二次記憶上の領域
  - 読み書きはページ単位で行われる.
- ページアウト（スワップアウト）
  - 物理メモリに入りきらないページをスワップ領域に書き出すこと
- ページイン
  - スワップ領域から物理メモリ上にページを読み込むこと

仮想記憶機構の動作概略

- 物理メモリに入りきらないページはスワップ領域に書き出されているとする.
- プロセスPがページAのアドレスを参照：
  - Aが物理メモリ上にあれば，それを参照する.
  - Aが物理メモリ上にないときはページフォルトが起こる．そこでOSは以下を行う.
    - Aがスワップ領域に書き出されているならばそれを物理メモリに読み込んでページテーブルに登録し，Pに制御を戻す.
    - そうでなければ不正なメモリ参照とする.

アドレス空間記述表

- 各論理ページがスワップ領域に書き出されているか，書き出されているならば二次記憶のどこにあるかを記述した表
- ページフォルトが起きたときにOSによって参照される
- ページテーブルとは異なる

デマンドページング

- プロセス開始時に物理ページを全く割り当てないでおき，プロセスの要求にまかせて物理ページを獲得させる方式
- 最初は頻繁にページフォルトが起こるが，時間とともにプロセス実行に必要な物理ページが徐々に獲得されていく.

- 例：exec

ページアウト

- カーネルがページを確保する際，未使用の物理ページがないときはどうするのか.
- 既に割当て済みの物理ページを選んでその内容をスワップ領域に書き出し(ページアウトor スワップアウト)，空いたページを作る.
  - このときページアウトされるページを犠牲(victim)ページと呼ぶ.
- どのようにして犠牲ページを選ぶのか？

ページ置換アルゴリズム(1)

- 犠牲ページを選ぶアルゴリズム
  - システム全体の性能を左右する.
- のぞましいアルゴリズム：
  - ページイン／アウトによるページ置換の回数をなるべく少なくしたい.
  - ページアウトする際は，ページイン以降更新されていないページを選ぶようにしたい.
    - ページ内容を書き出さずにすむため.

ページ置換アルゴリズム(2)

- 例：
  - 物理ページ数がnで，ページIDが1,2,...,n+1 の順に繰り返しアクセスが起こるとする.
- 最悪のアルゴリズムの場合
  - ちょうどアクセスするページがいつもページアウトされていて，毎回ページフォルトが起こる.
- 最善のアルゴリズムの場合
  - 例えば物理ページフレームがnのページのみをページアウトする．この場合 n+1 回のアクセスに2回だけページフォルトが起こる.

FIFOアルゴリズム

- ページアウトが必要になった時点で，最も最初にメモリに読み込まれていたページを犠牲ページとするアルゴリズム
- 単純で実装も容易だが，性能は必ずしも良くない．よく参照されるページも，単に古いというだけでページアウトされてしまうため.
- 利用可能フレーム数が大きくなったときにページフォルト率が増えるという現象(Beladyの異常)が生じることがある.

Beladyの例外 (Belady's Anomaly)

- 5つのページ 0, 1, 2, 3, 4 が以下のような順にアクセスされるとする
  - 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4
- FIFOアルゴリズム
  - 物理ページフレーム数が3の場合
    - 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4
    - ページフォルトは9回
  - 物理ページフレーム数が4の場合
    - 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4
    - ページフォルトは10回

最適(optimal)アルゴリズム

- 現在の物理ページのうち，アクセスされる順番が最後のものをページアウトする.
- 一般には実現不可能
  - なぜなら，将来にわたってアクセスの順番が完全にわかっていることが前提だから.
    - Cf. Shortest Job First スケジューリング
- (もしできたとしたら)最適である.
  - 最適＝ページフォルトの回数が最も少なくなる.

メモリアクセスの時間的局所性

- 実際のページ置換アルゴリズムでは，将来のアクセスの順序を予測する.
- そのためにメモリアクセスの時間的局所性という性質を用いる.
  - 多くのプログラムは，最近アクセスしたメモリアドレス(およびその近傍)を近い将来ふたたびアクセスするという性質を持つ.
  - つまり，最近アクセスされたページは近い将来再びアクセスされる可能性が高いと考える.

LRU (Least Recently Used)

- 最近最も使われていないページをページアウトするアルゴリズム
- 実現方法
  - メモリアクセスが起こるたびに，その参照が起こった物理ページのアクセス時刻の記録(タイムスタンプ)を取る.
  - 犠牲ページを選ぶ際は，タイムスタンプが最小のものを選べばよい.

LRUの正確な実現は難しい

- メモリアクセス毎のタイムスタンプの記録はオーバーヘッドが高い.
  - ハードウェアによる支援が必要
- 犠牲ページを選ぶ際，タイムスタンプが最小のものを探さなければならない.

LRUの近似的な実現(1)

- LRUを近似的に実現するための機構
- ページテーブルのエントリに参照ビットと汚れビットと呼ばれるフィールドを設ける.
  - 参照ビット(reference bit)
    - ページを読んだときにセットされる.
  - 汚れビット(dirty bit)
    - ページに書き込んだときにセットされる.

LRUの近似的な実現(2)

- OSは定期的（例えば1秒おき）に以下を行う
  - 各ページテーブルエントリに up, Rp, Dp というビットが確保されている.
  - 各物理ページ p について以下を実行：
    - up = Rp | Dp;
    - Rp = 0;
    - Dp = 0;
      - Rp, Dp はページ p の参照ビットと汚れビット
- ページ置換：
  - up, Rp, Dp 全てが0のものを犠牲ページとする.
  - そのようなページがなければ，Rp, Dp が共に0であるものを犠牲ページとする

ページ置換をどう設計するか

- 局所的 vs 大域的
  - 局所的：ページフォルトを起こしたプロセスのページから犠牲ページを選ぶ
  - 大域的：全てのプロセス中から選ぶ.
  - 一般に大域的のほうがよい.
- プロセスをどのくらいロードするか
- ページサイズ
- write-through vs write-back

応用：メモリマップドファイル

- mmap: メモリマップドファイル
  - スワップ領域の代わりに好きなファイルを指定
  - ページインでファイルの内容をメモリに読み込み, 書き込んだ内容はページアウトでファイルに書き出される.

- 大きなファイルの読み書きを行う場合，read でいったんバッファ(仮想メモリ内)に読み込んでから再び書き出すのに比べると効率的

mmap (OS X)

```
void *
mmap (void *addr, size_t len, int prot,
      int flags, int fd, off_t offset);
```

- ファイルディスクリプタ fd で示されるファイルの offset バイト目からの len バイトを，アドレス [addr', addr' + len) で確保する.
- addr' (mmapの返値)
  - addr = NULL → 適当に選んだアドレス
  - addr ≠ NULL → addr (確保できれば)
    - 確保不可能 → MAP_FAILED

通常のファイルアクセスとの違い

- mmapを使ってマップされたファイルの内容は，あたかもメモリのように（実際メモリなのだが）ランダムアクセスできる.
- open/read/write/lseek を使った場合，基本的には逐次アクセスである.

実験：mmap vs. read

- 大きめのファイルをアクセスして，実行時間を比較してみる.
  - テキストファイル中の特定の文字を数える.
  - ランダムアクセスではなく，先頭から順にアクセスしてみる.
- 疑問：通常，ファイルの内容はファイルキャッシュと呼ばれるメモリ領域にコピーされ，読み出しはそこから行われる．さらに逐次アクセスであれば差はないのではないか？

rw_count.c

```
int main (int argc, char *argv[]) {
    char *file = argv[1];
    FILE *fp = fopen(file, "r");
    int c;
    long nz = 0;
    while ((c = getc(fp)) != EOF)
        if (c == 0) nz++;
    rewind(fp);
    long nm = 0;
    while ((c = getc(fp)) != EOF)
        if (c == 255) nm++;
    printf("nz=%ld, nm=%ld\n", nz, nm);
    fclose(fp);
    return EXIT_SUCCESS;
}
```

mmap_count.c

```
int main (int argc, char *argv[]) {
    char *file = argv[1];
    int fd = open(file, O_RDONLY);
    size_t size = (size_t)lseek(fd, 0, SEEK_END);
    lseek(fd, 0, SEEK_SET);
    signed char *a =
      mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    long nz = 0;
    for (off_t i = 0; i < size; i++)
        if (a[i] == 0) nz++;
    long nm = 0;
    for (off_t i = 0; i < size; i++)
        if (a[i] == -1) nm++;
    printf("nz=%ld, nm=%ld\n", nz, nm);
    munmap(a, size);
    close(fd);
    return EXIT_SUCCESS;
}
```

実行時間(CPU時間，単位は秒)

ファイルサイズ	rw_count	mmap_count
10M	0.14	0.02
50M	0.74	0.13
100M	1.48	0.26
500M	7.43	1.34

CPU: 2×2.66GHz Dual-Core Intel Xeon, Memory: 16GB, Compiler: llvm-gcc 4.2.1, OS: Mac OS X 10.7.2

実行時間(CPU時間，単位は秒)

ファイルサイズ	rw_count	mmap_count
10M	0.19	0.01
50M	0.81	0.06
100M	1.64	0.14
500M	8.2	0.66

CPU: 2.93GHz Intel Xeon X5670 × 2 (12 core)，Memory: 54GB (limited to 6GB), Compiler: Intel ICC 11.1, OS: SUSE Linux Enterprise Server 11 SP1

プライベート／共有マップ

- mmapの第4引数(flags)
  - MAP_PRIVATE
    - プロセス毎に固有の物理ページを持つ.
      - 読み出し専用であれば共有されることもある.
    - 書き込み結果はファイルに反映されず，他のプロセスとも共有されない.
  - MAP_SHARED
    - 複数のプロセスが同じ物理ページを共有する.
    - 書き込み結果はファイルに反映し，他のプロセスとも共有される.

メモリ割当機構としてのmmap

- mmapによって特別なファイル (/dev/zero) をMAP_PRIVATEでマップすると，特定のファイルに結びつかないメモリ領域を得ることができる.
- mallocの実装などで用いられている.
  - e.g., OpenBSD malloc, glibc malloc, etc.

Copy on Write (CoW)

- メモリ内容（論理ページ）のコピーをさぼり，かわりに物理ページを共有する.
- 当該ページの書き込みを禁止しておく.
- 当該ページに書き込みをすると例外が発生するので，そのときに実際のコピーを行う.
- 例：
  - mmapにおけるプライベートマッピング
  - forkでのメモリコピー

forkにおけるCoW

- 子プロセスの生成＝ページテーブルとアドレス空間記述表のコピー
  - 子プロセス生成直後は，物理メモリを親子で共有しておく．
  - 各ページは書き込み不可にしておく．
- 書き込まれたページのみ，その時点でコピーを生成する(CoW).
- 子プロセスがexecveを実行したときに，子プロセスのマッピングを除去する．

共有メモリ

- 物理メモリへのマッピングを利用して，プロセス間の共有メモリを実現する.

xv6のプロセスとメモリ管理(1)

プロセス

- 実行中のプログラム
  - メモリ上にコピーされたプログラムと，それを実行するために割り当てられたメモリやCPUなどの計算資源に関する諸情報として実現される.
  - CPUのコア数よりも多くのプロセスが（見かけ上）同時に実行できる.
    - マルチタスキング

マルチタスキング

- （1つのCPUコア内で）複数個のプロセスを切り替え，あたかも同時に実行しているように見せる仕組み.
- マルチタスキングの種類
  - プリエンプティブ(preemptive)
    - ハードウェアタイマーで割り込みをかけ，そのタイミングで切り替える．プログラミングの際に切り替えを意識しなくてよい.
  - ノンプリエンプティブ
    - システムコール実行時，あるいはプログラム中で明示的にプロセスを切り替える.

関数swtchによるユーザレベルでのマルチタスキング

- swtch：xv6カーネルで用いられているコンテクストスイッチ関数

```
void swtch(struct context *old, struct context *new);
```

- old, new: context構造体へのポインタ
- 現在実行中のコンテキストがoldに保存され, newが表すコンテクストの実行を「再開」する
- コンテクスト：実行の状態・スナップショット
  - Schemeの継続(continuation)とほぼ同様

swtchの動作

```
void f() {
  printf("F1\n");
  swtch(&f_context, &g_context);
  printf("F2\n");
  swtch(&f_context, &g_context);
  printf("F3\n");
}

void g() {
  printf("G1\n");
  swtch(&g_context, &f_context);
  printf("G2\n");
  swtch(&g_context, &f_context);
  printf("G3\n");
}

int main() { ... f(); ... }
```

実行結果

```
F1
G1
F2
G2
F3
G3
```

context構造体

```
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

- ra
  - 関数の戻り番地
- sp
  - スタックポインタ
- s0, ..., s11
  - callee-save レジスタ
  - 関数呼び出しの際，呼び出された関数側で保存すべきレジスタ

RISC-Vでの関数呼び出し

- 引数：レジスタ x10, ..., x17 (別名 a0, ..., a7)
- 返値：レジスタ x10 (別名 a0)
- 戻り番地：レジスタ x1 (別名ra)
- スタックポインタ：レジスタ x2 (別名 sp)
- その他
  - x5, x6, x7, x28, ..., x31 (t0, ..., t6) (caller save)
  - x8, x9, x18, ..., x27 (s0, ..., s11) (callee save)
  - x0 ゼロレジスタ（常に0）

swtch

defs.h

```
void swtch(struct context *, struct context *);
```

swtch.S

```
.globl swtch
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

swtchのテストプログラム

```
void foo() {
    uint64 c = 0;
    for (;;) {
        printf("foo : %lu\n", c);
        swtch(&foo_context, &bar_context);
        c += 1;
    }
}

void bar() {
    uint64 c = 0;
    for (;;) {
        printf("bar : %lu\n", c);
        swtch(&bar_context, &baz_context);
        c += 2;
    }
}

void baz() {
    uint64 c = 0;
    for (;;) {
        printf("baz : %lu\n", c);
        swtch(&baz_context, &foo_context);
        c += 3;
    }
}
```

swtchを呼ぶたびに
foo→bar→baz→…
と実行が切り替る

実行例(xv6)

```
$ swtest
foo: 0
bar: 0
baz: 0
foo: 1
bar: 2
baz: 3
foo: 2
bar: 4
baz: 6
foo: 3
bar: 6
baz: 9
...
```

swtest.c

```c
#include "kernel/types.h"
#include "user/user.h"

// Saved registers for kernel context switches. (from kernel/proc.h)
struct context { ... };

// swtch.S (from kernel/defs.h)
void swtch(struct context*, struct context*);

struct context foo_context;
struct context bar_context;
struct context baz_context;

#define STACK_DEPTH 512
uint64 bar_stack[STACK_DEPTH];
uint64 baz_stack[STACK_DEPTH];

void foo() { ... }
void bar() { ... }
void baz() { ... }

int main() {
    // setting up initial contexts
    bar_context.ra = (uint64)bar;
    bar_context.sp = (uint64)(bar_stack + STACK_DEPTH);
    baz_context.ra = (uint64)baz;
    baz_context.sp = (uint64)(baz_stack + STACK_DEPTH);
    // start from foo
    foo();
    return 0;
}
```

実行してみたい場合は，講義サイトにあるxv6-riscvのswtestブランチをチェックアウトする（すでにcloneしてある場合は git pull してから git checkout swtest）.

```
$ git clone https://github.com/titech-os/xv6-riscv.git
$ git checkout swtest
$ make
$ make qemu
```

おまけ：x86_64版

```
        .globl swtch
swtch:
        ## save old callee-save registers
        pushq %rbp
        pushq %rbx
        pushq %r12
        pushq %r13
        pushq %r14
        pushq %r15

        ## push argument register
        pushq %rdi

        ## switch stacks
        movq %rsp, (%rdi)       # *old = sp
        movq %rsi, %rsp         # sp = new

        ## load argument register
        popq %rdi

        ## load new callee-save registers
        popq %r15
        popq %r14
        popq %r13
        popq %r12
        popq %rbx
        popq %rbp

        ret
```

```
        .globl swtch
swtch:
        push { r0, r4-r7, lr }
        mov r2, r8
        mov r3, r9
        mov r4, r10
        mov r5, r11
        mov r6, r12
        push { r2-r6 }

        mov r2, sp
        str r2, [r0]
        mov sp, r1

        // movs r2, #0
        // mov lr, r2

        pop { r2-r6 }
        mov r8, r2
        mov r9, r3
        mov r10, r4
        mov r11, r5
        mov r12, r6
        pop { r0, r4-r7, pc }
```

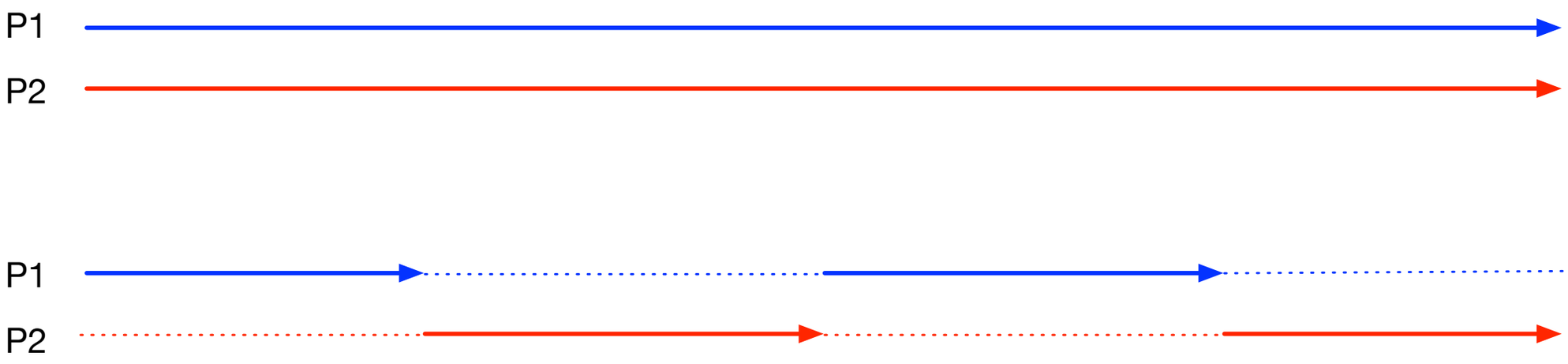
おまけ：armv7l版

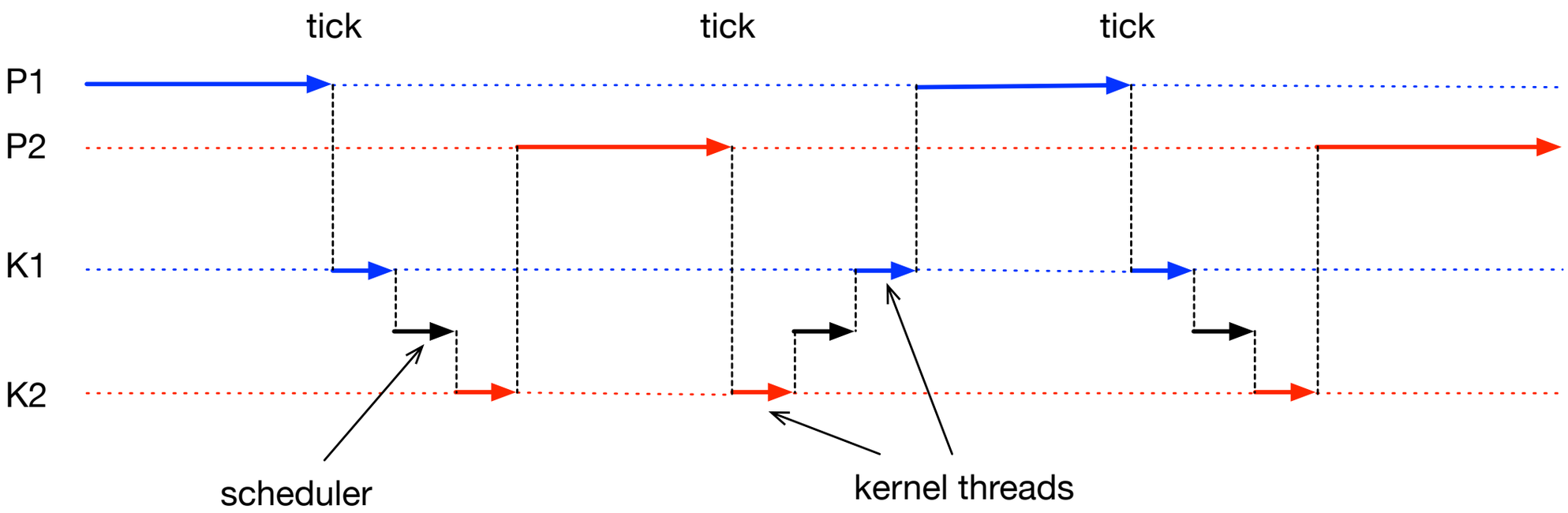
32ビット版ARM

swtestとOSプロセスの違い

- swtest
  - ユーザレベルでの擬似マルチタスキング
    - 関数呼び出しの代わりにswtchを使うことでコンテクスト切り替えを行っている
    - 明示的にswtchを呼ばないとコンテクストスイッチは発生しない
    - コルーチン(coroutine)が行っていることとほぼ同じ
- OSのプロセス
  - コンテクスト切り替えはカーネルが行う
  - ユーザプログラムからはその様子はわからない

OSによるコンテクスト切り替え





カーネルスレッド

- ユーザプロセスの制御を行うカーネル内のスレッド
  - ユーザプロセスの実行は割り込みによりカーネルスレッドに切り替わる.
  - カーネルスレッドはスケジューラを介して（他の）ユーザプロセスの実行に切り替わる.
- struct proc *myproc();
  - カーネルスレッドが担当しているユーザプロセスのプセス構造体へのポインタを返す.

プロセスコントロールブロック(PCB)

- プロセスを管理するためのデータ構造
  - カーネルが作って管理する
  - 主な内容(OSによって異なる)：
    - プロセスid
    - プロセスの状態
    - 退避されたCPUレジスタの内容
      - あるいはスレッド情報(へのポインタ)
    - スタック
    - メモリ管理情報
    - アカウント情報
    - I/O関連の情報

コンテクストスイッチング

- 実行中のプロセス(スレッド)を他に切り替える作業のこと.
  - 手順 (プロセス1→プロセス2)
    - CPUレジスタの内容をプロセス1のPCBにコピーする(退避する).
    - プロセス1のPCBの状態をRUNNABLEにする.
    - プロセス2のPCBの状態をRUNNINGにする.
    - プロセス2のPCBに保存してあったレジスタの内容をCPUレジスタにコピーする(復帰する).
  - CPUの全レジスタを退避・復帰することで, メモリ管理情報も同時に退避・復帰できる.

xv6のPCB(proc構造体)

kernel/proc.h

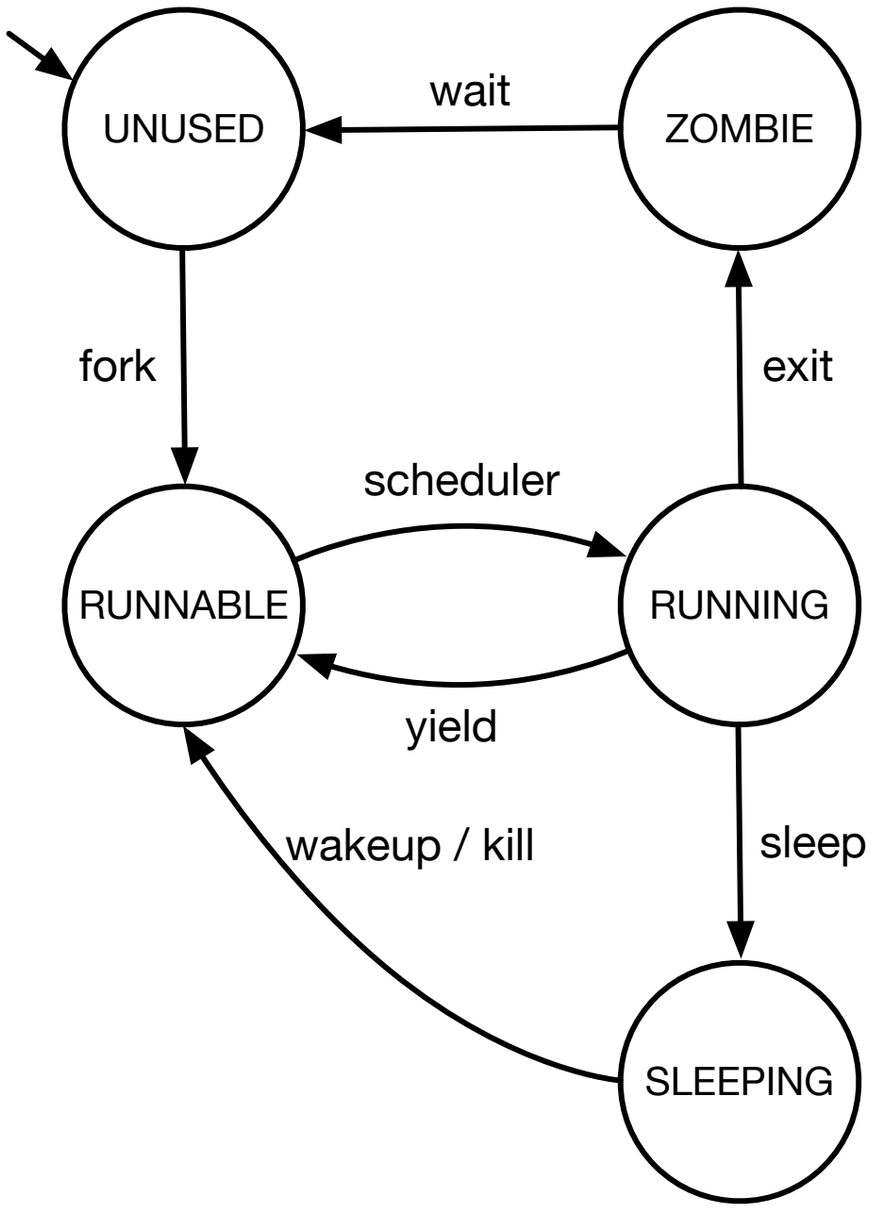
```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

xv6のプロセスの状態

```
enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```



- UNUSED
  - 構造体が未使用状態
- SLEEPING
  - I/O待ち等
- RUNNABLE
  - 実行可能だがCPUは割り当てられていない
- RUNNING
  - 実行中
- ZOMBIE
  - 終了準備中

プロセススケジューリング

- 関数schedulerは，procを順にみて状態がRUNNNABLEであるプロセスを見つけたらRUNNINGにし，そのプロセスに制御を移す.
- このとき，関数swtchによって現在実行中のプロセススタックを差し替えている

まとめ

- メモリ管理(2)
  - 仮想記憶
    - スワップ領域，アドレス記述表，デマンドページング，ページ置換アルゴリズム，mmap, copy-on-write
  - xv6のプロセスとメモリ管理(1)
    - マルチタスキング，コンテクストスイッチ，カーネルスレッド，