システムソフトウェア

2021年度

第8回 (10/28)

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

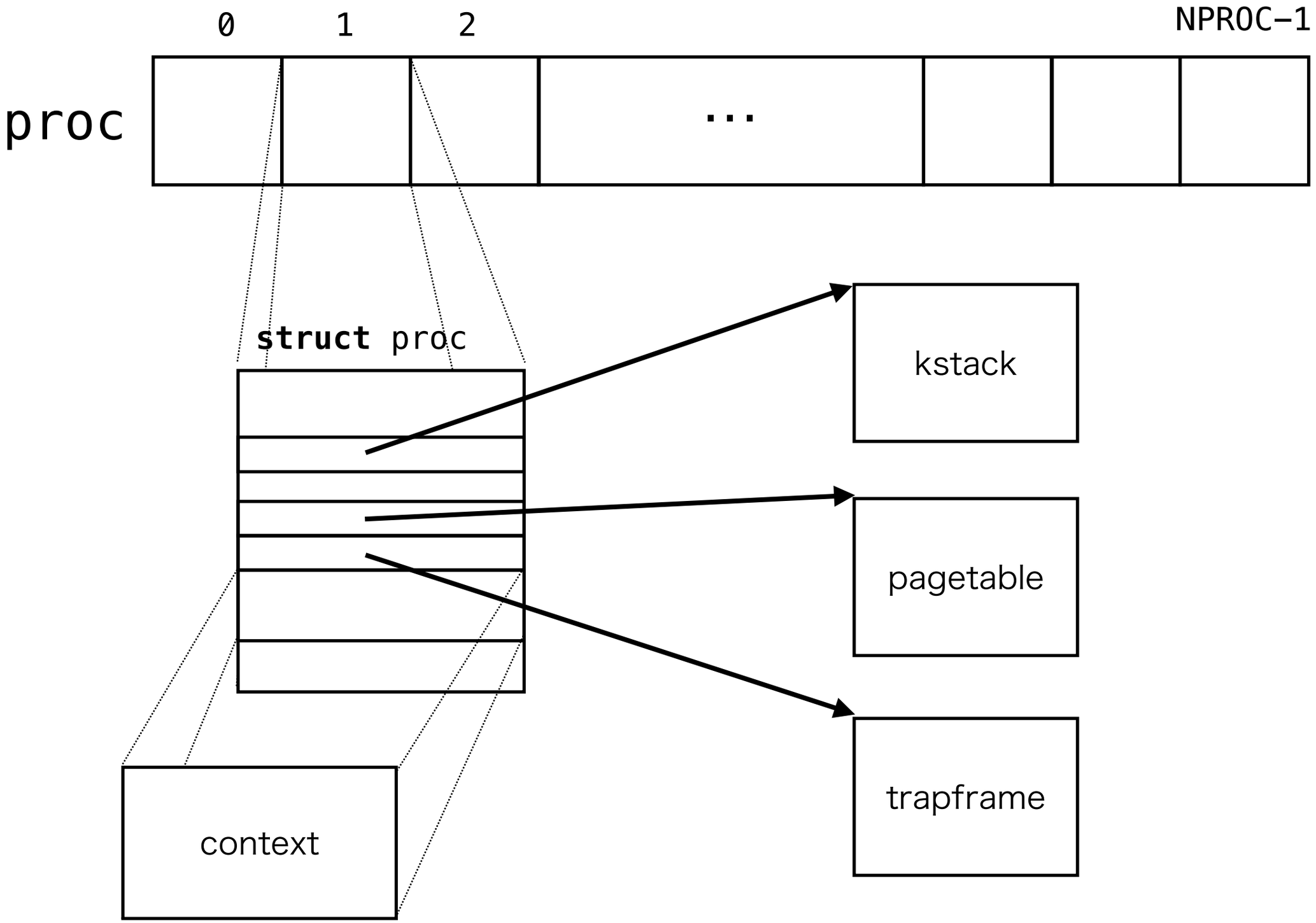
本日のメニュー

- xv6のプロセス(2)

スケジューラ[再]

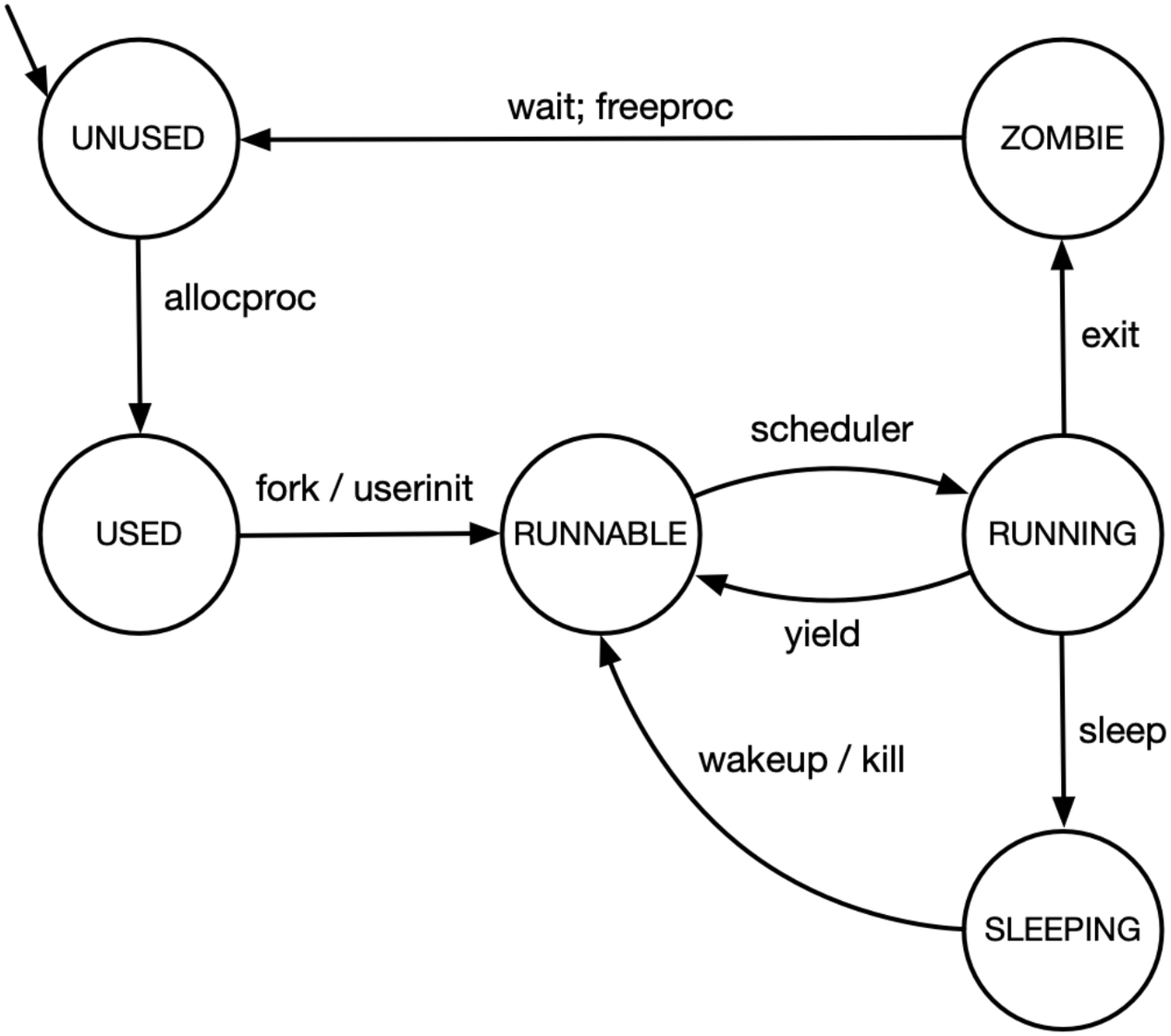
- scheduler (proc.c)
  - 各CPUが実行
  - プロセステーブル(proc)をみてモードが RUNNABLEになっているものがあったら RUNNINGに切り替え, swtchで制御を移す

プロセステーブル


0
1
2
NPROC−1
proc
...
struct proc
kstack
pagetable
trapframe
context

プロセスの状態(proc.h)

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



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

スケジューラの動作

```
for(;;){
  // Avoid deadlock by ensuring that devices can interrupt.
  intr_on();

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == RUNNABLE) {
      // Switch to chosen process.  It is the process's job
      // to release its lock and then reacquire it
      // before jumping back to us.
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&p->lock);
  }
}
```

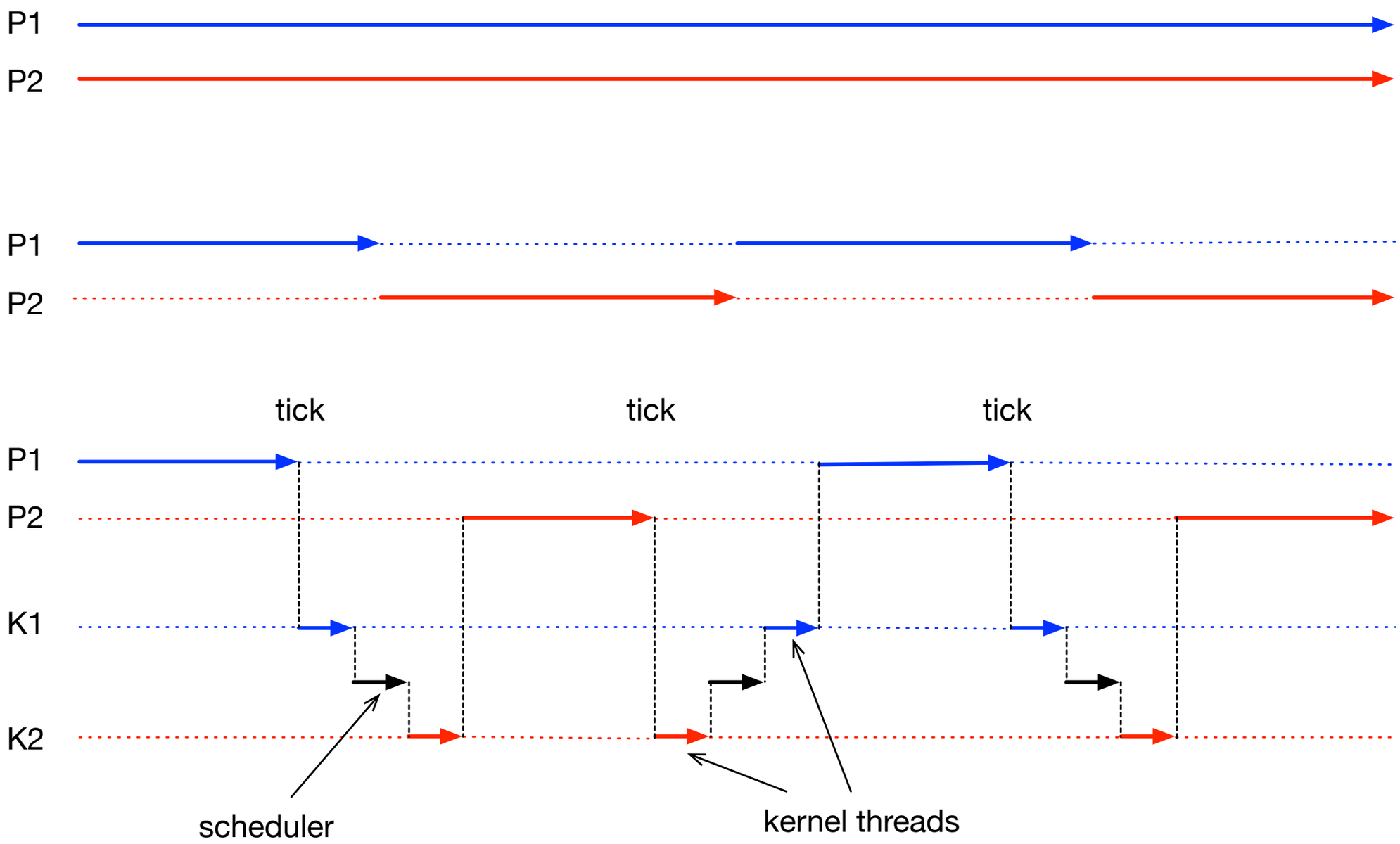
yield: swtchの行き先

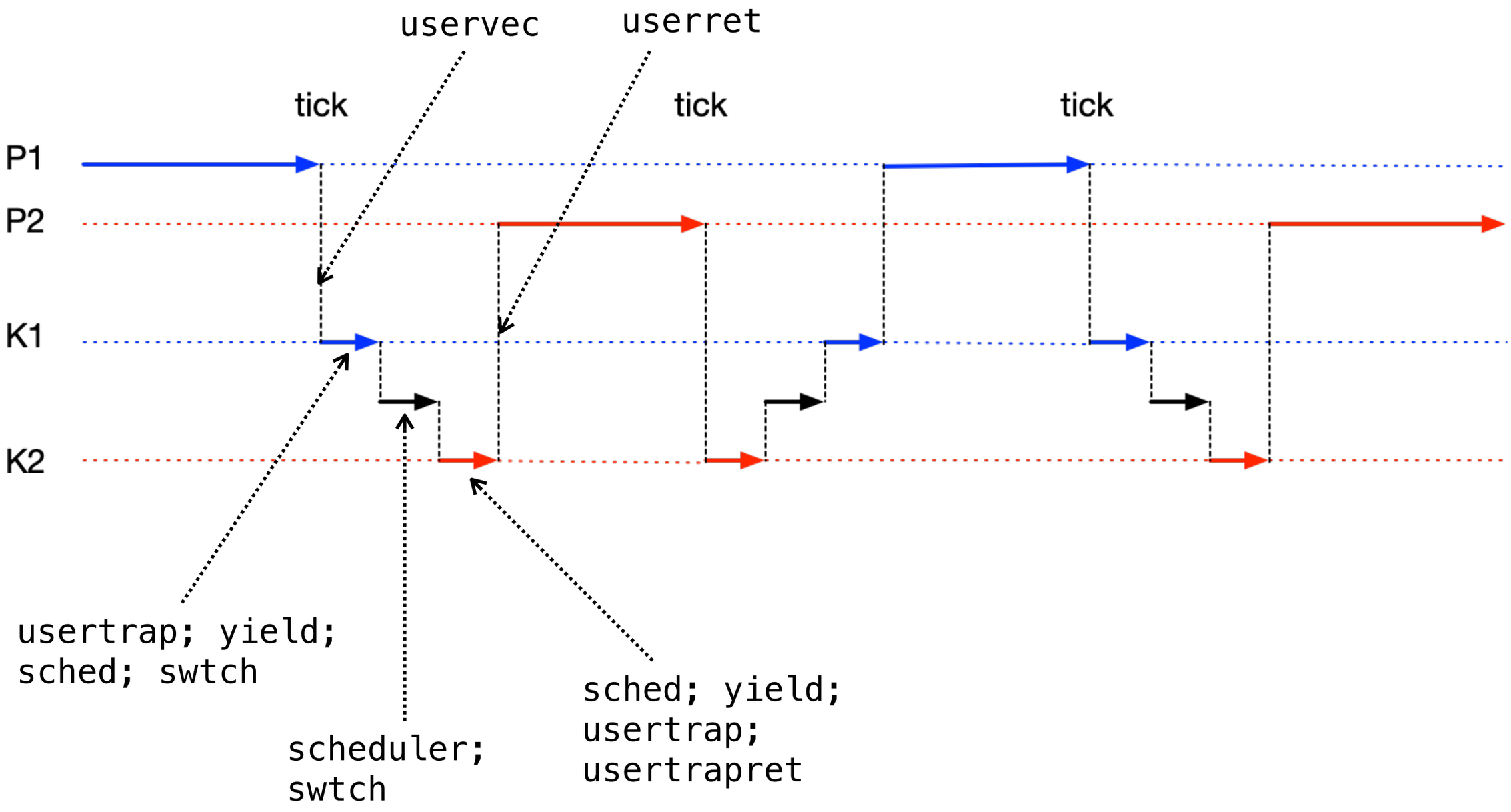
```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&p->lock))
    panic("sched p->lock");
  if(mycpu()->noff != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(intr_get())
    panic("sched interruptible");

  intena = mycpu()->intena;
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}
```

```
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

yield: タイマー割り込みで他のカーネルスレッドに実行を譲る.

OSによるコンテクスト切り替え




uservec
userret
tick
tick
tick
P1
P2
K1
K2
usertrap; yield;
sched; swtch
scheduler;
swtch
sched; yield;
usertrap;
usertrapret

プロセスの「種」の作成

- allocproc (proc.c)
  - プロセステーブル(proc)からUNUSEDなものをひとつ見つけて初期化
    - pidの割り当て
    - trapframeとページテーブルの設定
    - コンテクストの設定
      - カーネルスタックの底をコンテクストのspに設定
      - forkretをコンテクストの戻りアドレスに設定

プロセスの生成(fork)

- fork (proc.c)
  - allocprocでプロセスの「種」を生成
  - 親プロセスのメモリ空間をコピー
  - 親プロセスのレジスタ(trapframe)をコピー
  - ファイル等の情報をコピー
  - 子プロセスをRUNNABLEにする
    - あとはスケジューラに任せる

プロセスの生成(init)[再]

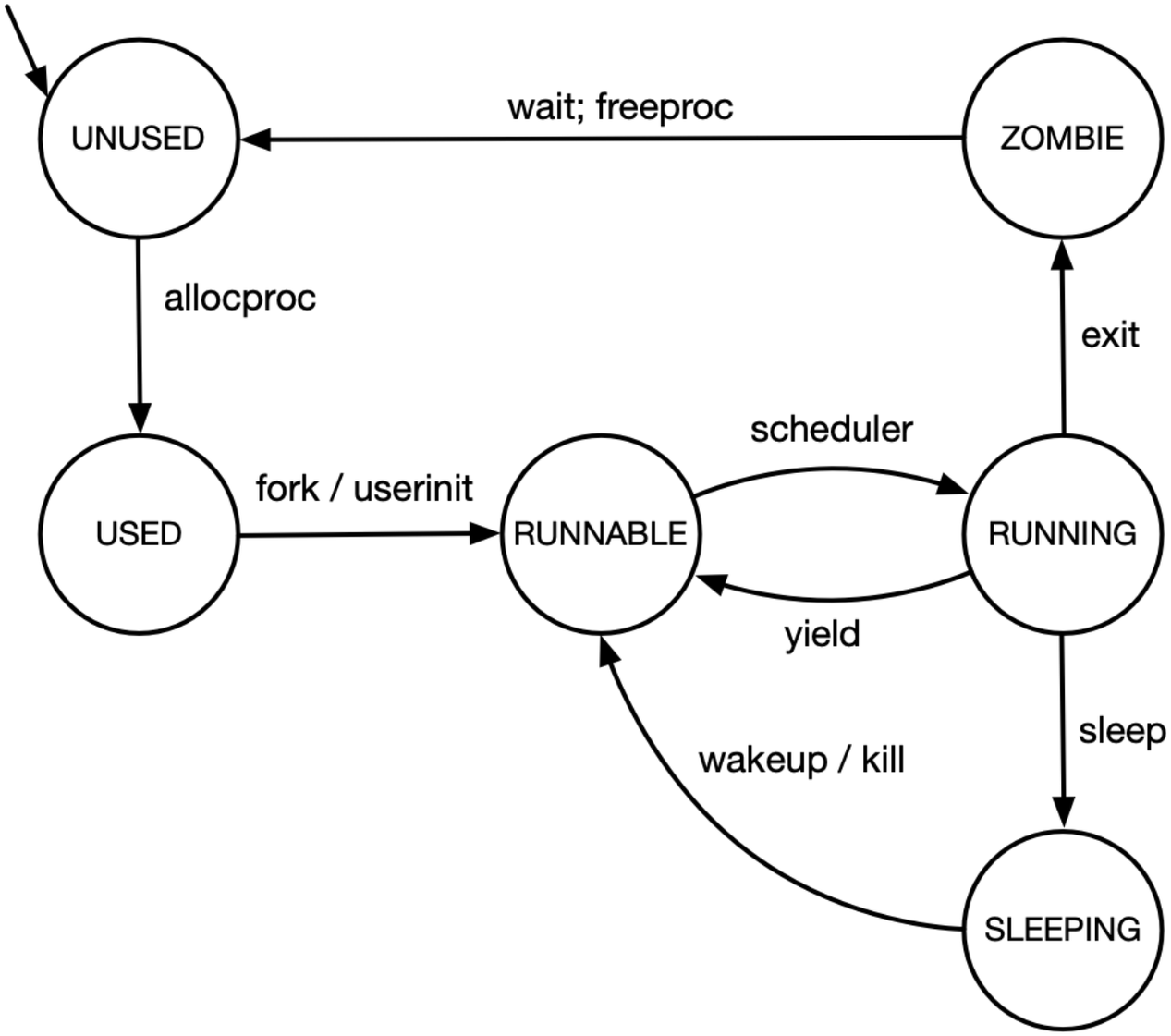
- userinit (proc.c)
  - allocprocでプロセスの「種」を作る
  - 1ページだけのメモリ空間を作り，initcodeの内容をコピーする
    - initcodeは exec("init") を実行するだけのコード
  - 当該ページの先頭(0)をpcに，末尾をスタックポインタとするようなtrapframeを作る
  - RUNNABLEにする
    - あとはスケジューラに任せる

forkret

- forkやuserinitで作成された新しいプロセスが初めて動き出すときの，スケジューラからの「戻り先」として機能する
- usertrapretを実行してユーザ空間に「戻る」
  - forkの場合は親プロセスからコピーしたtrapframeを利用
  - userinitの場合は人工的に作ったtrapframeを利用

プロセスの状態(proc.h)

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



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

SLEEPING

- プロセスが何かを待って寝ている状態
  - 何か：I/Oの完了などのイベント
- RUNNABLEとは異なり，他のプロセスによって起こされない限りは実行されない

- sleep(chan, lock)
  - 実行したプロセスをSLEEP状態にする
  - chan: 何に対して待っているかを表すデータ
    - 任意のデータへのポインタ
    - モニタの条件変数に相当
  - lock: chanの排他制御のためのスピンロック

sleep (proc.c)

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  // Must acquire p->lock in order to
  // change p->state and then call sched.
  // Once we hold p->lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup locks p->lock),
  // so it's okay to release lk.

  acquire(&p->lock);  //DOC: sleeplock1
  release(lk);

  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  release(&p->lock);
  acquire(lk);
}
```

wakeup (proc.c)

```
// Wake up all processes sleeping on chan.
// Must be called without any p->lock.
void
wakeup(void *chan)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    if(p != myproc()){
      acquire(&p->lock);
      if(p->state == SLEEPING && p->chan == chan) {
        p->state = RUNNABLE;
      }
      release(&p->lock);
    }
  }
}
```

スピンロックによるセマフォ

```
struct semaphore {
  struct spinlock lock;
  int count;
};

void P(struct semaphore *s) {
  while (s->count == 0);
  acquire(&s->lock);
  s->count -= 1;
  release(&s->lock);
}

void V(struct semaphore *s) {
  acquire(&s->lock);
  s->count += 1;
  release(&s->lock);
}
```

sleep/wakeupによるセマフォ

```
struct semaphore {
  struct spinlock lock;
  int count;
};

void P(struct semaphore *s) {
  acquire(&s->lock);
  while (s->count == 0)
    sleep(s, &s->lock);
  s->count -= 1;
  release(&s->lock);
}

void V(struct semaphore *s) {
  acquire(&s->lock);
  s->count += 1;
  wakeup(s);
  release(&s->lock);
}
```

まとめ

- xv6のプロセス(2)