システムソフトウェア

2020年度

第5回 (10/19) **下書き版**

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

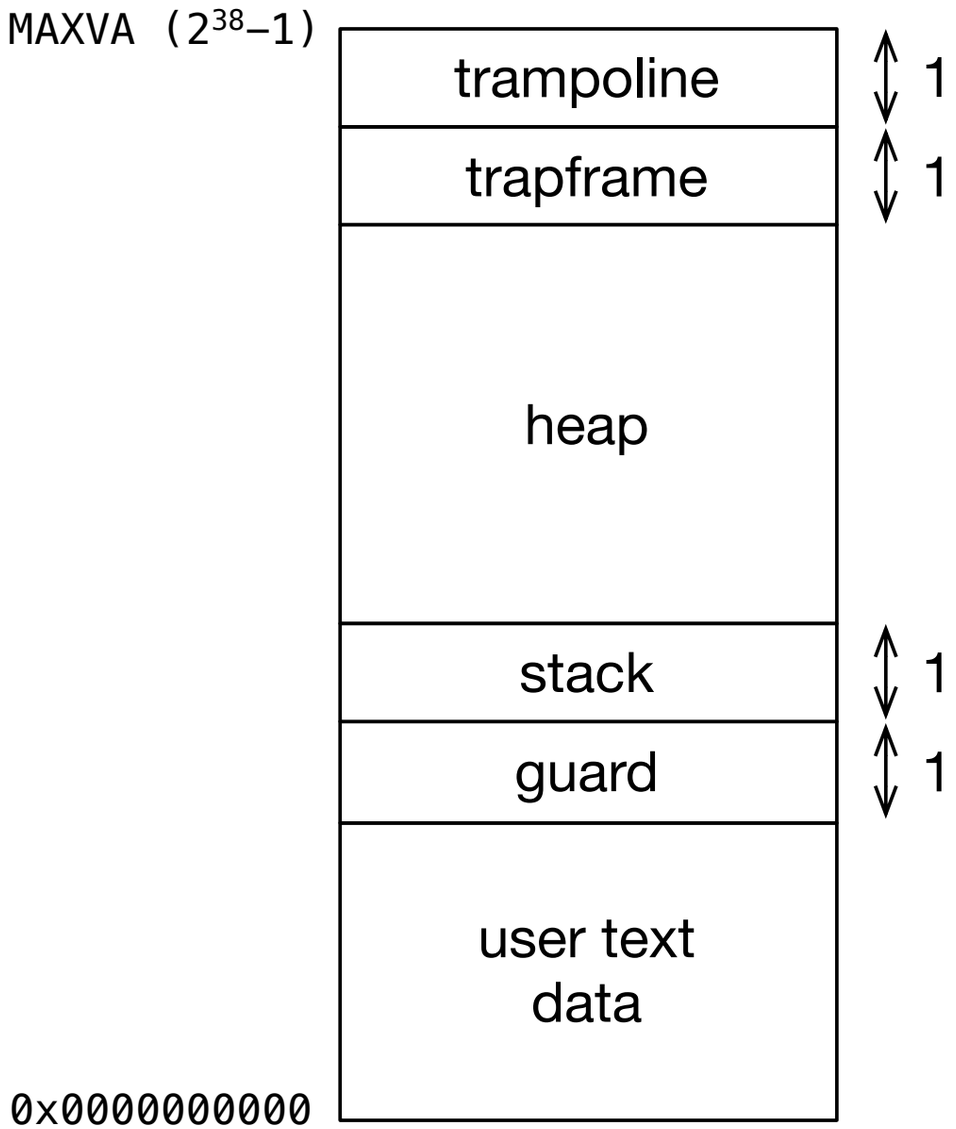
http://titech-os.github.io

e-mail: takuo∅c.titech.ac.jp

本日のメニュー

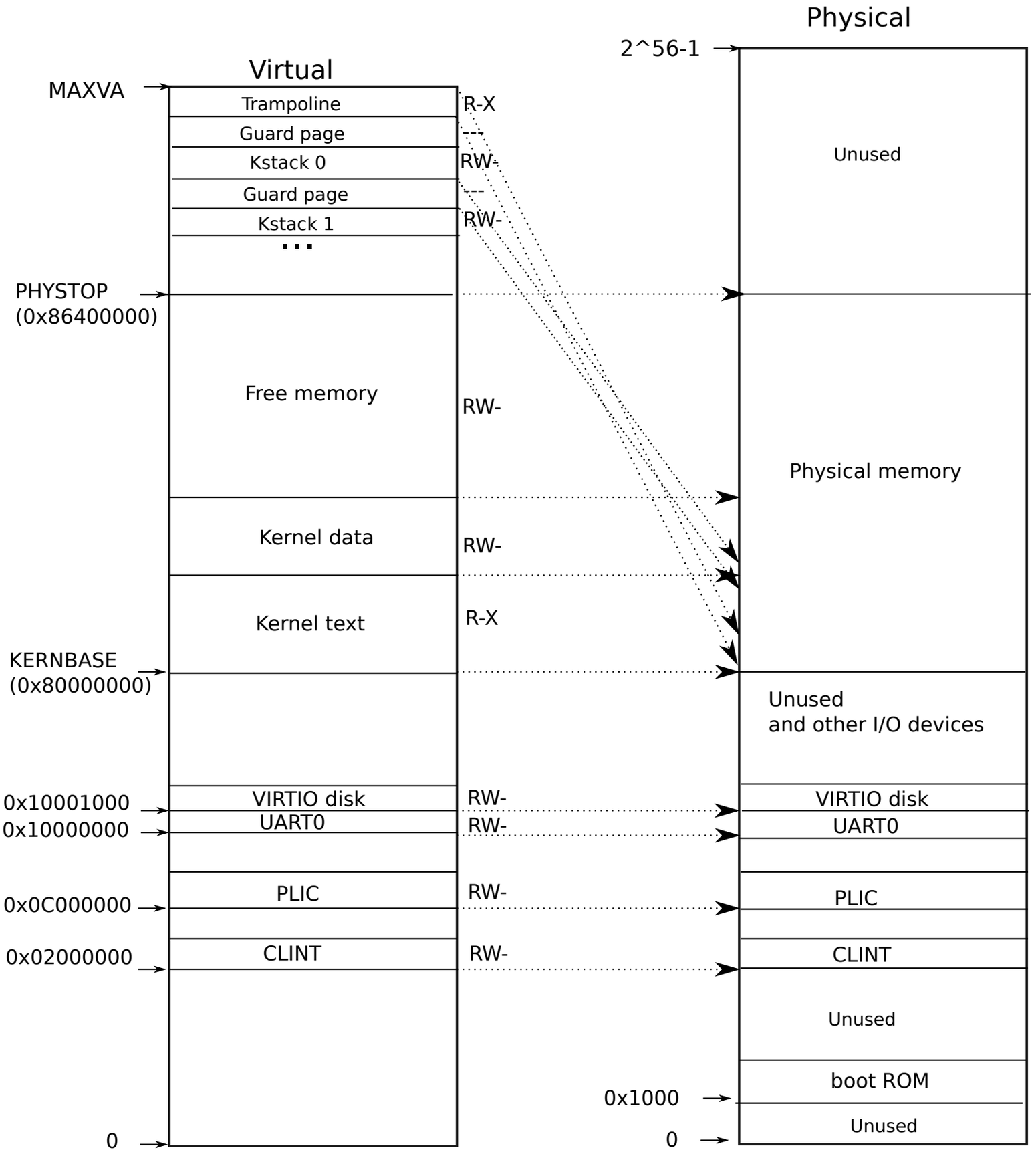
- プロセス管理(1)

プロセスのメモリ空間



- アドレス：38ビット
- user text & data
  - プログラムと定数
  - コンパイル時にサイズが決まっているデータ
- stack
  - コールスタック
- heap
  - 実行時に獲得されるメモリ
- trampoline
  - カーネル空間との切り替えに使うコード
  - カーネル空間と共有
- trapframe
  - システムコールの引数等

カーネルのメモリ空間


Physical
2^56-1
Virtual
MAXVA
Trampoline
R-X
Guard page

Kstack 0
RW-
Guard page

Kstack 1
RW-
...
PHYSTOP
(0x86400000)
Free memory
RW-
Kernel data
RW-
Kernel text
R-X
KERNBASE
(0x80000000)
0x10001000
VIRTIO disk
RW-
0x10000000
UART0
RW-
0x0C000000
PLIC
RW-
0x02000000
CLINT
RW-
0
Unused
Physical memory
Unused
and other I/O devices
VIRTIO disk
UART0
PLIC
CLINT
Unused
boot ROM
0x1000
Unused
0

xv6のPCB(proc構造体)

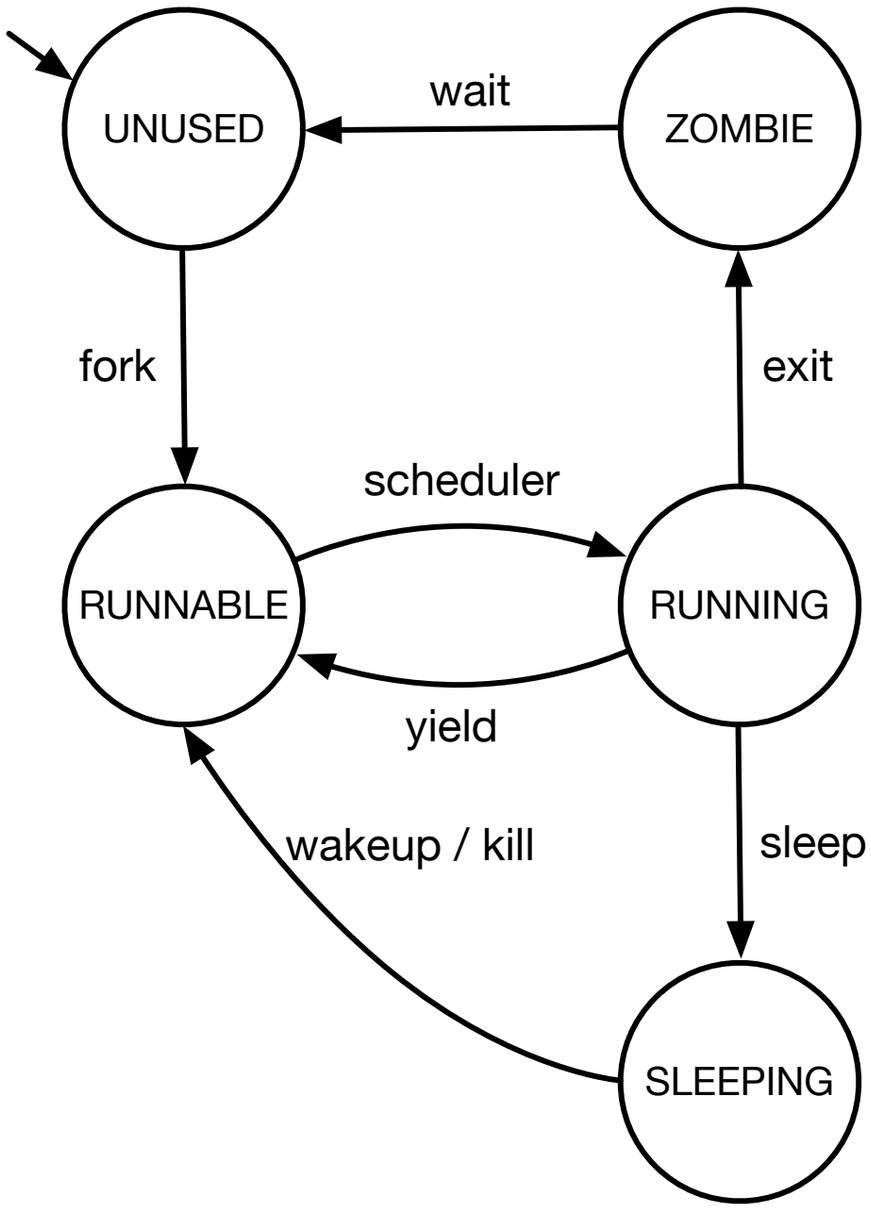
```
// Per-process state
struct proc {
  struct spinlock lock;

  // 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;               // Bottom of kernel stack for this process
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // Page table
  struct trapframe *tf;        // 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)
};
```

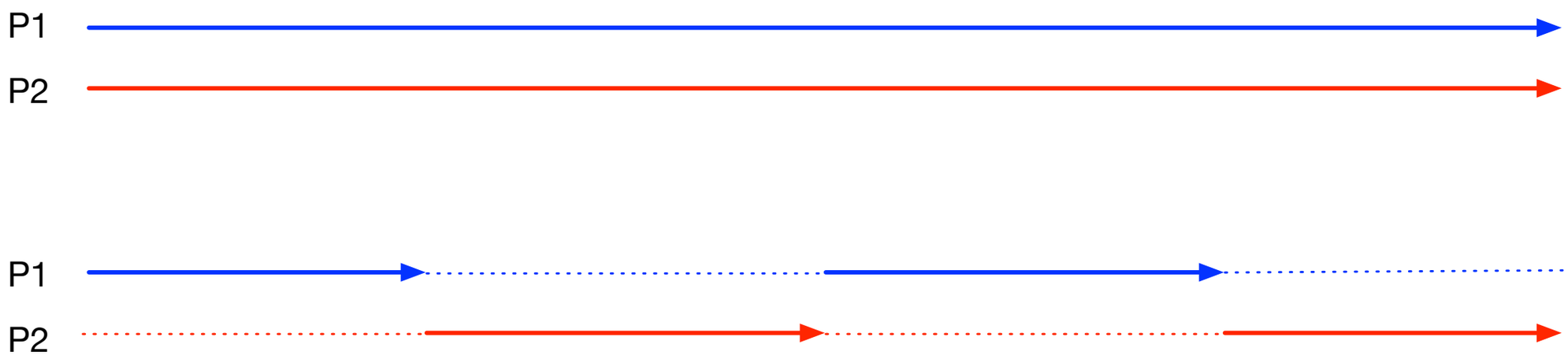
xv6のプロセスの状態

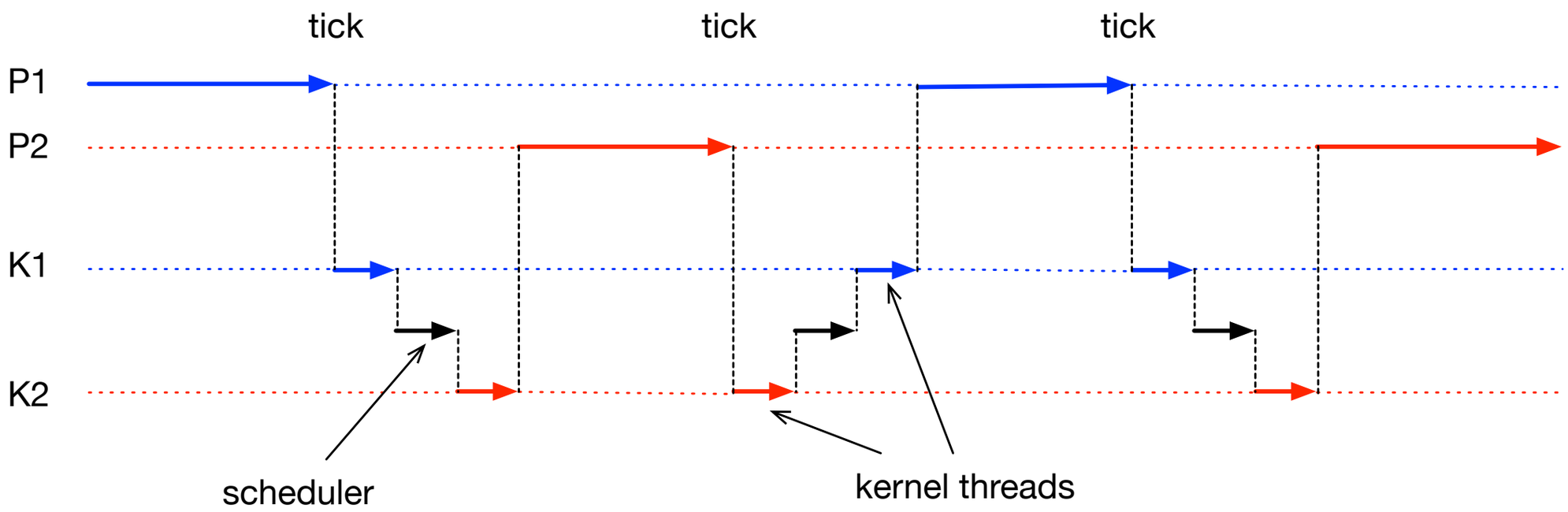
```
enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```



- 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

コンテクストスイッチングの動作





プロセステーブルとその初期化(proc.c)

```
struct proc proc[NPROC];
```

```
// initialize the proc table at boot time.
void
procinit(void)
{
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  for(p = proc; p < &proc[NPROC]; p++) {
      initlock(&p->lock, "proc");
      p->kstack = KSTACK((int) (p - proc));
  }
}
```

allocproc (proc.c)

- プロセス(の種)を作成する
  - 配列procを順に見て，状態(stateフィールド)がUNUSEDになっている要素を1つ見つける
  - カーネルスタックをセット
    - トラップフレーム

- すべてのプロセスは構造体procの配列procで管理されている.

fork

- allocprocでプロセス(の種)を確保
- 親プロセスのレジスタをコピー
- 関数copyuvmで親プロセスのメモリ空間をコピーする
- 開いているファイルの情報をコピー
- 状態をRUNNABLEにする

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

swtchによるコンテキスト切替

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

- old, new: context構造体へのポインタ
- switchを実行すると，現在実行中のコンテキストはoldが指すcontext構造体に格納され，newが指すcontext構造体の内容が新たな実行コンテキストになる.

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
```

まとめ

- プロセスとスレッド(1)
  - プロセスの生成と終了
    - fork, exec
  - プロセス構造体(PCB)
  - マルチタスキング
    - プリエンプティブ／ノンプリエンプティブ
    - コンテクストスイッチング
  - スケジューリング
  - xv6でのコンテクストスイッチングの実装