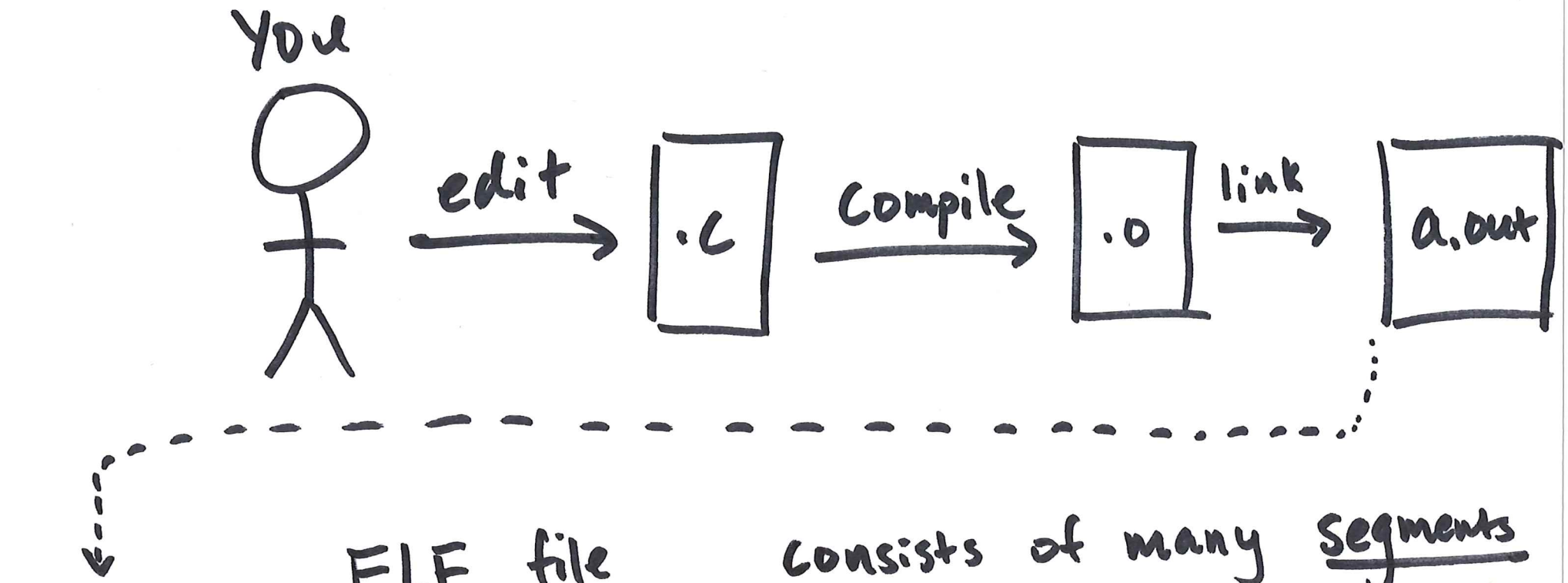
Lec 5

Processes + Concurrency



ELF file

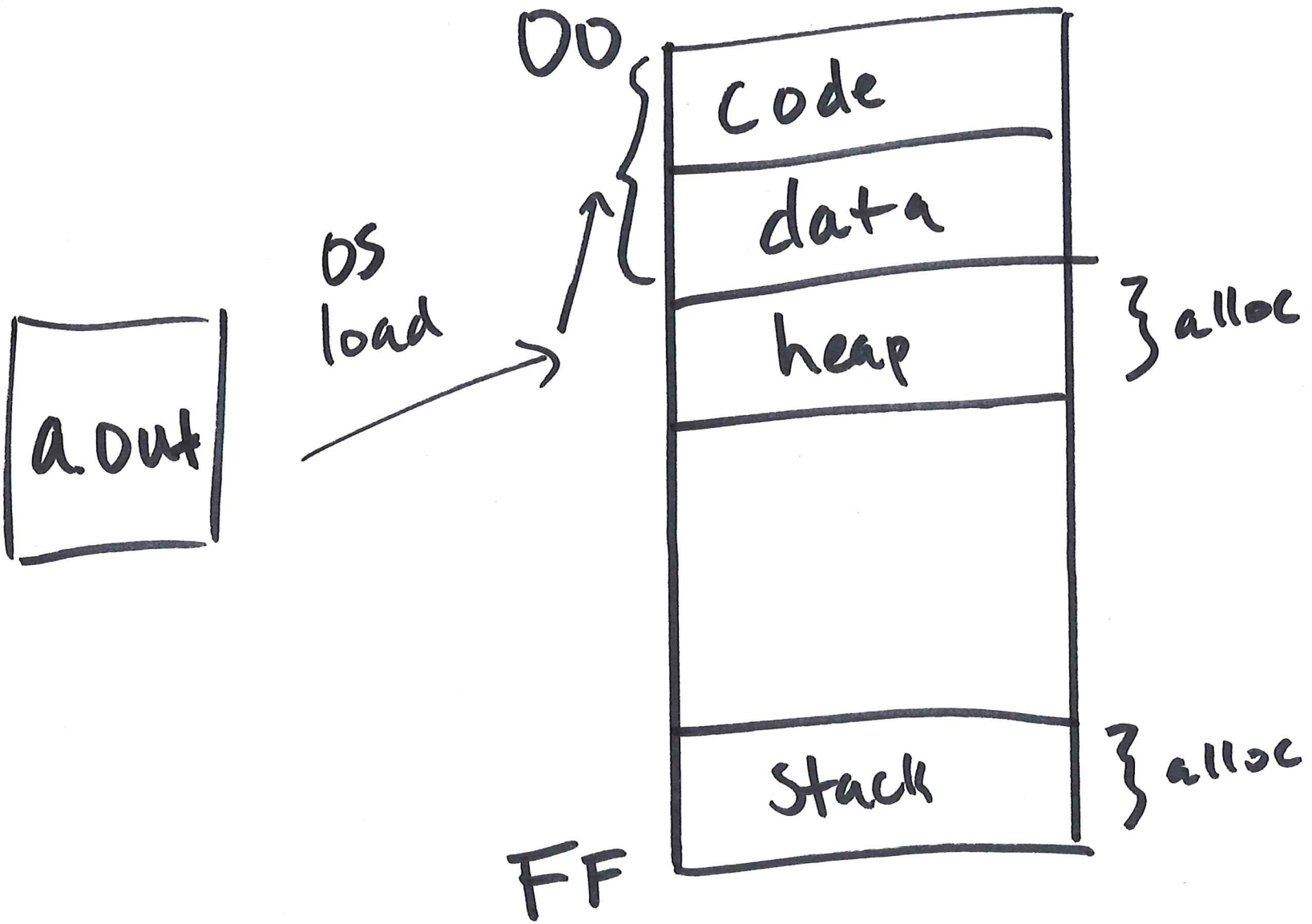
Executable and Linkable Format

consists of many segments

- code segments
- data segments

descriptions of where to put stuff in memory

new proc virtual mem



So far, only one process

- still important to isolate the Kernel
- still need to switch into Kernel mode
  - interrupts + syscalls
  - use the Kernel stack

More than one process

- isolated from each other → separate virtual addr spaces
- each might be in the middle of syscall
  - ↳ one kernel stack per process
- Process Control Block (PCB) tracks state

Questions:

- how to create new process?
- how to switch between processes?

```
enum procstate {
  // Empty space in PCB table. UNUSED entries should have NO resources.
  UNUSED,
  // This slot has been allocated with `allocproc` .
  EMBRYO,
  // Sleeping, waiting for event (should not be scheduled).
  SLEEPING,
  // Ready to be scheduled (procs only scheduled when state is RUNNABLE).
  RUNNABLE,
  // Currently running.
  RUNNING,
  // Exited, waiting to be reaped.
  ZOMBIE,
};

// Per-process state
struct proc {
  struct vspace vspace;      // Virtual address space descriptor
  char *kstack;              // Kernel stack
  enum procstate state;      // Process state
  int pid;                   // Process ID
  struct trap_frame *tf;     // Trap frame for current syscall
  struct context *context;   // swtch() here to run process
  void *chan;                // If non-zero, sleeping on chan
  int killed;                // If non-zero, have been killed
  char name[16];             // Process name (debugging)
};
```

(PCB)

```
// proc.c
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

NPROC = 64

```
// from proc.h
struct context {
  uint64_t r15;
  uint64_t r14;
  uint64_t r13;
  uint64_t r12;
  uint64_t r11;
  uint64_t rbx;
  uint64_t rbp;
  uint64_t rip;
};
```

Creating a new process

- allocate + initialize PCB
- create + initialize address space
  - allocate Kstack
- load program, alloc user heap + stack
- copy args into user stack
- arrange trapframe to "resume" at preface
  - preface.S: call main and then call exit()
- mark process as runnable

```
.globl preface
preface:
    call main
    mov %rax, %rdi
    call exit
```

Unix process API

- fork() create copy of curr process
- exec() replace program being run by this proc.
- wait() wait for process to finish
- exit() process is done (tell waiter)
- signal() send notification to other process

example calling fork()

```
#include <stdio.h>
#include <unistd.h>

int main() {
  int child_pid = fork();
  if (child_pid == 0) {
    // I am the child
    printf("Child: pid = %d\n", getpid());
    return 0;
  } else {
    // I am the parent
    printf("Parent: child_pid = %d\n", child_pid);
    return 0;
  }
}
```

5.10

Implementing fork/exec

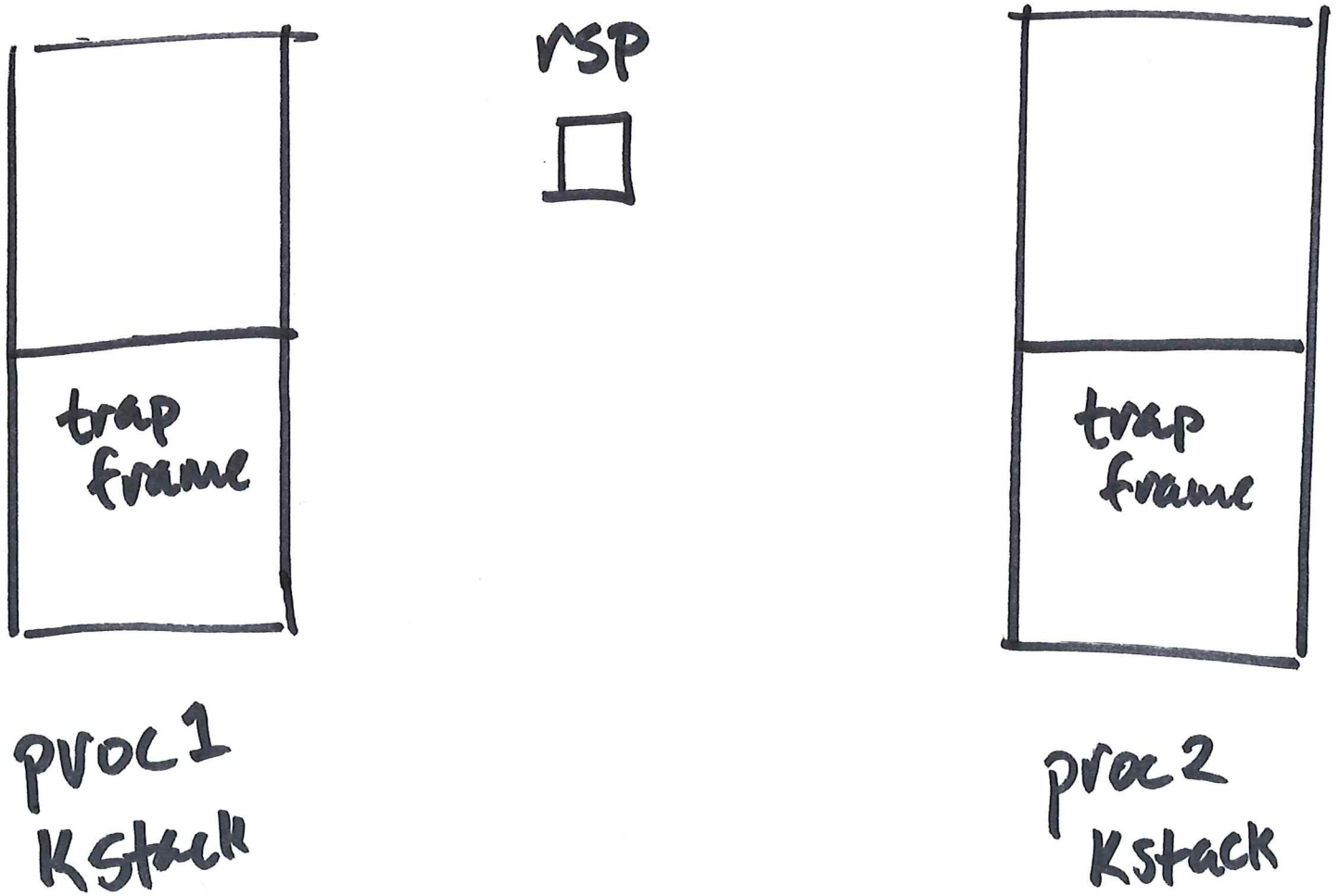
fork: copy address space of parent (!)
- in child, arrange to return 0
- in parent, arrange to return child PID

exec: load new program into address space
- arrange trapframe to resume at preface

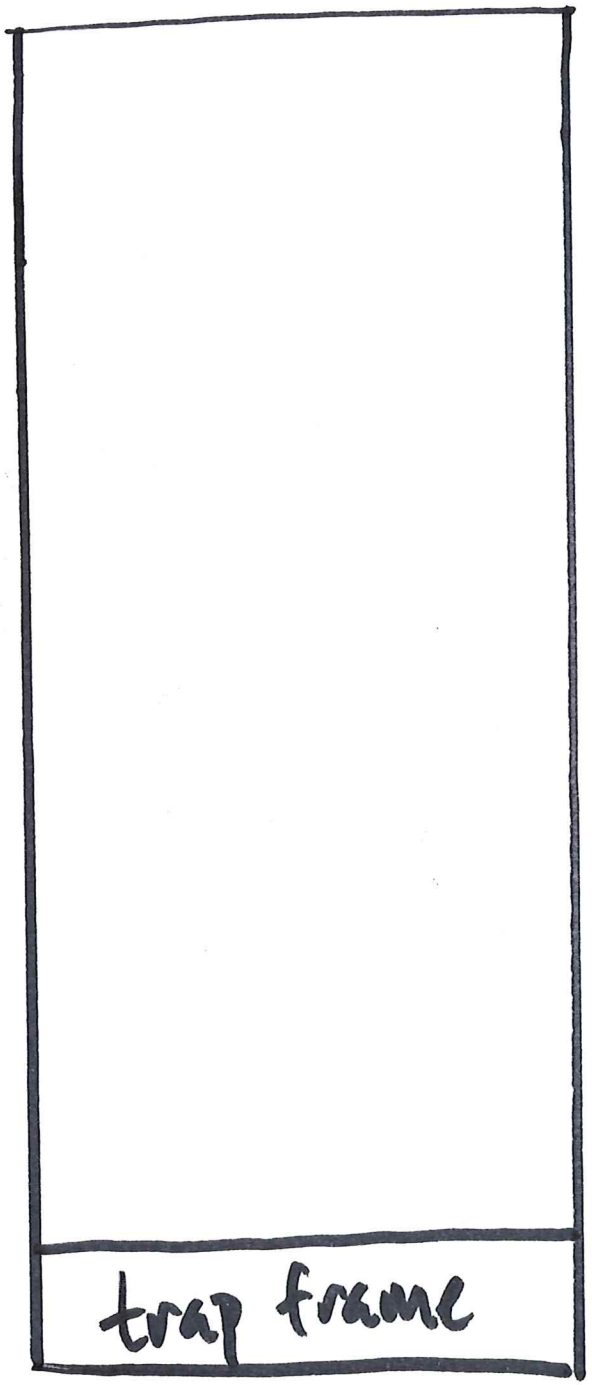
Is this slow?

How to switch between processes?

idea: timer interrupt handler "resumes" to different process (!!!)



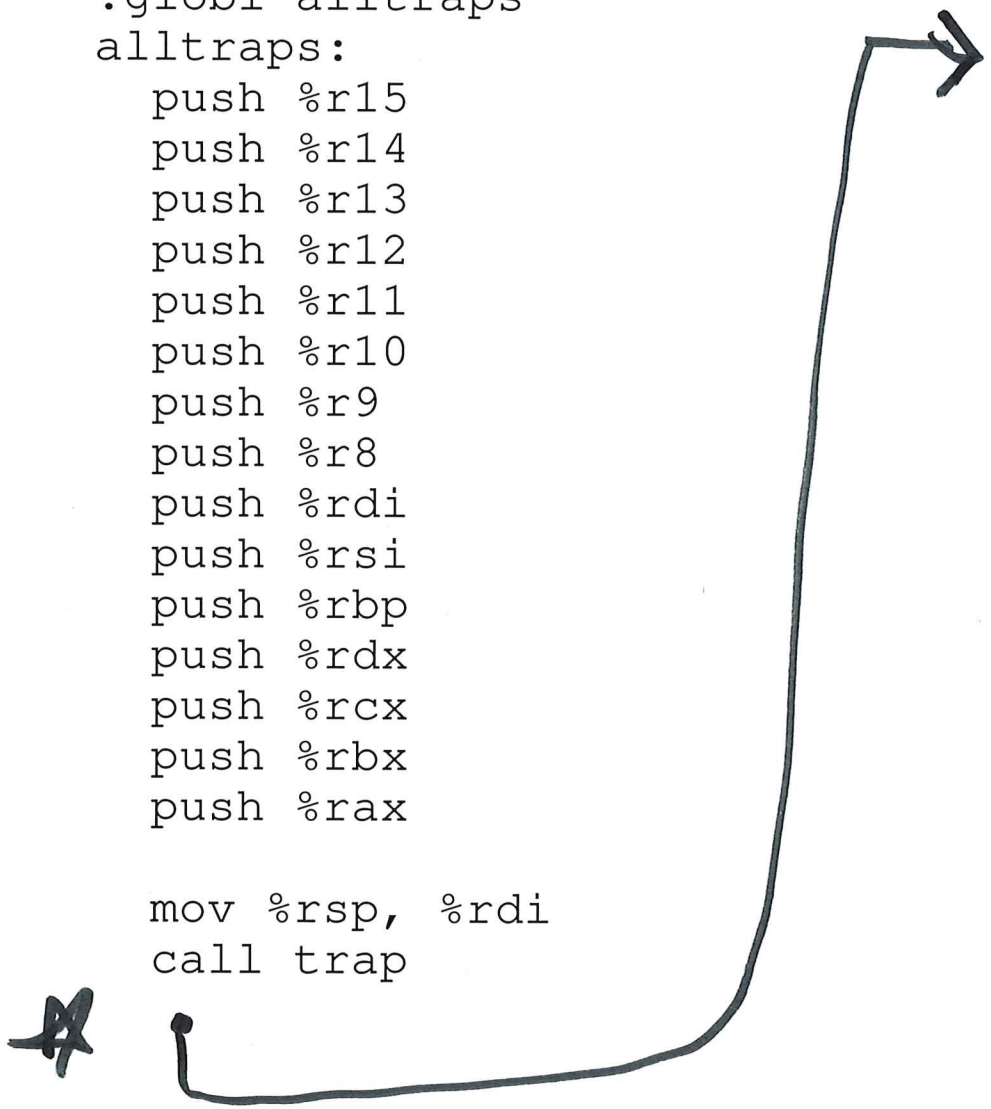
Context switch in xK (see 3.10 + 5.13½



- hw calls alltraps trapasm.h
- calls trap.c trap()
- calls yield (proc.c)
- calls sched
- calls swtch (swtch.S)
- switches to scheduler
- finds runnable proc
- calls swtch
- switches to new proc
- returns from swtch (!) ...

```
.globl alltraps
alltraps:
  push %r15
  push %r14
  push %r13
  push %r12
  push %r11
  push %r10
  push %r9
  push %r8
  push %rdi
  push %rsi
  push %rbp
  push %rdx
  push %rcx
  push %rbx
  push %rax

  mov %rsp, %rdi
  call trap
```



```
.globl trapret
trapret:
  pop %rax
  pop %rbx
  pop %rcx
  pop %rdx
  pop %rbp
  pop %rsi
  pop %rdi
  pop %r8
  pop %r9
  pop %r10
  pop %r11
  pop %r12
  pop %r13
  pop %r14
  pop %r15
  add $16, %rsp
  iretq
```

5.13½

```
// trap.c
void trap(struct trap_frame *tf) {
  uint64_t addr;

  if (tf->trapno == TRAP_SYSCALL) {
    ...
    syscall();
    ...
    return;
  }

  ...

  // Force process to give up CPU on clock tick.
  if (myproc() && myproc()->state == RUNNING &&
      tf->trapno == TRAP_IRQ0 + IRQ_TIMER)
    yield();
    ●
  ...
}
```

5.14

```
// proc.c

void yield(void) {
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}

void sched(void) {
  int intena;

  ...

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

%rdi, %rsi

5.15

```
.globl swtch
swtch:
  push %rbp
  push %rbx
  push %r11
  push %r12
  push %r13
  push %r14
  push %r15

  mov %rsp, (%rdi)
  mov %rsi, %rsp

  pop %r15
  pop %r14
  pop %r13
  pop %r12
  pop %r11
  pop %rbx
  pop %rbp

  ret
```

%rdi : current Kthread context

%rsi : next Kthread context

switches stacks (!)

```
void scheduler(void) {
  struct proc *p;

  for (;;) {
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      mycpu()->proc = p;
      vspaceinstall(p);
      p->state = RUNNING;
      swtch(&mycpu()->scheduler, p->context);
      vspaceinstallkern();

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

5.17

A suspended process