CS 1550

Week 7 – Lab 3

Interrupts

Part 2

Teaching Assistant

Henrique Potter

CS 1550 – Dues Dates

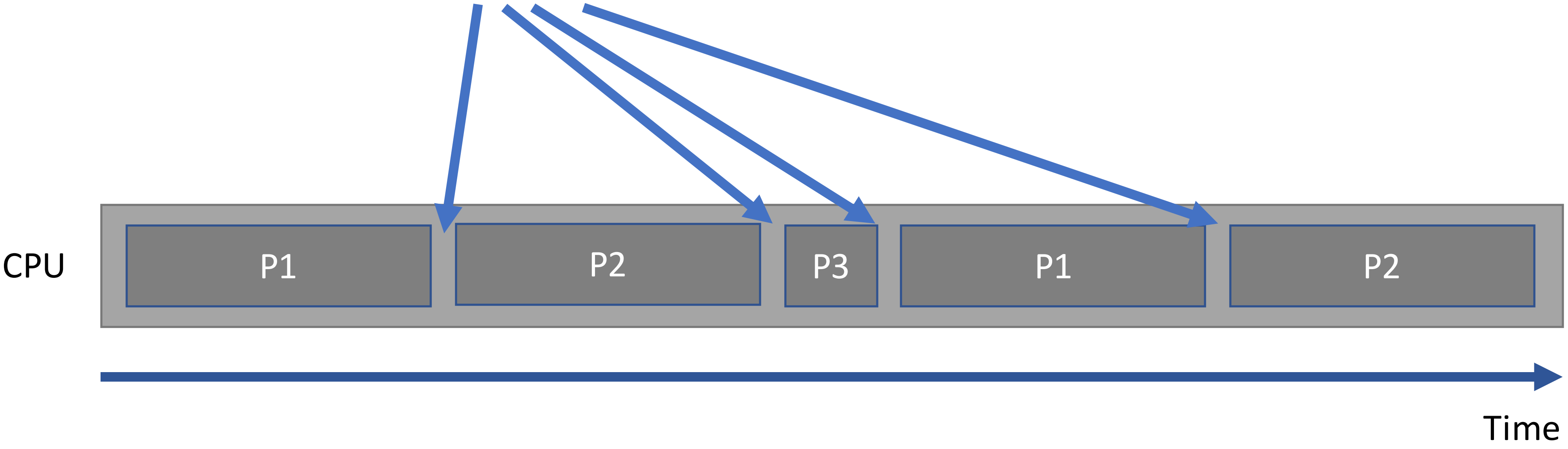
- Lab 3: Monday, March 9 @11:59pm
- Project 2: Tuesday, March 3 @11:59pm

CS 1550 – Dues Dates

- Lab 3: Monday, March 9 @11:59pm
- ~~Project 2: Tuesday, March 3 @11:59pm~~
- Project 2: Friday, March 6 @11:59pm

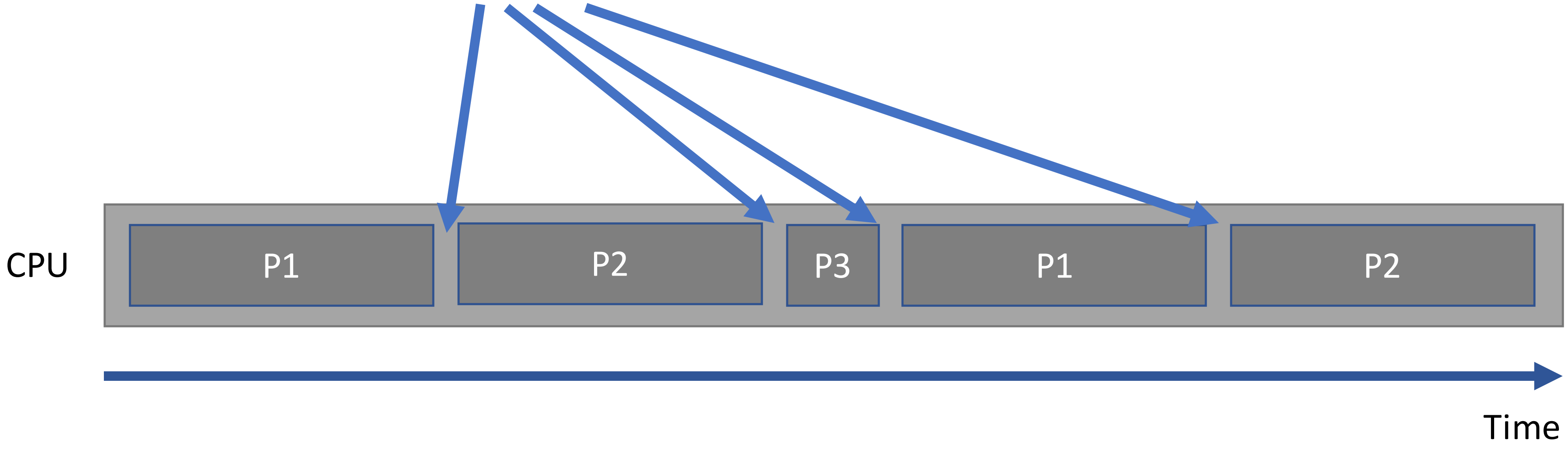
Scheduling of processes

- In xv6, an interrupt for the scheduler is generated on every clock tick
- The scheduler is called, and a new process is selected



Scheduling of processes

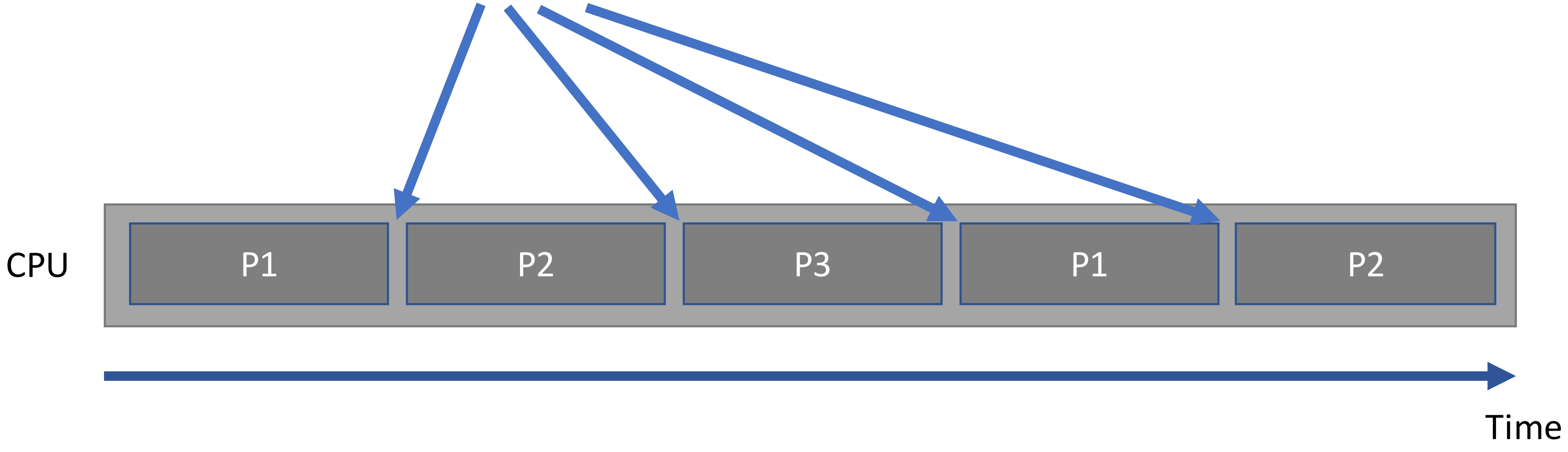
- In xv6, an interrupt for the scheduler is generated on every clock tick
- The scheduler is called, and a new process is selected



If the scheduler selects new processes in a round robin fashion. What's is wrong with this picture?

Scheduling of processes

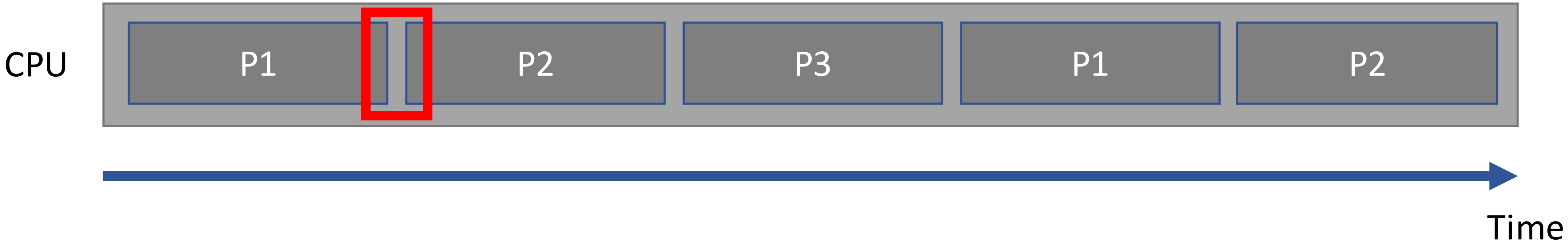
- In xv6, an interrupt for the scheduler is generated on every clock tick
- The scheduler is called, and a new process is selected



Processes should look
more evenly distributed!

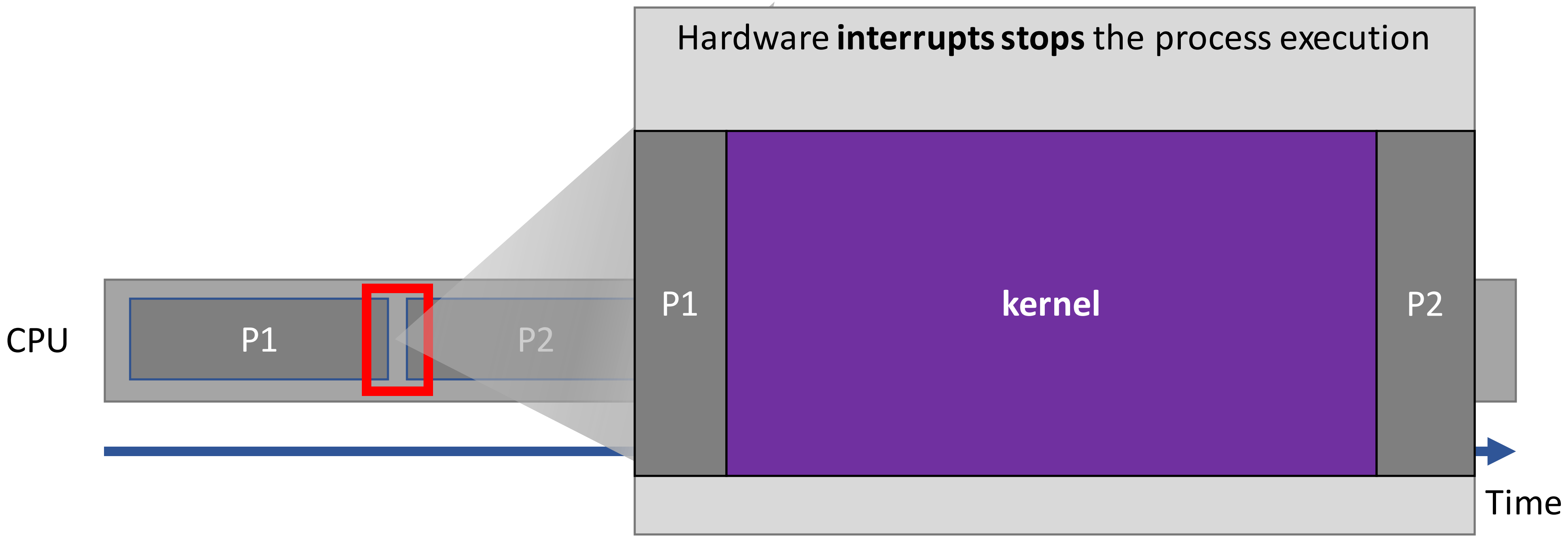
Scheduling of processes

- How processes are switched during their execution?



Scheduling of processes

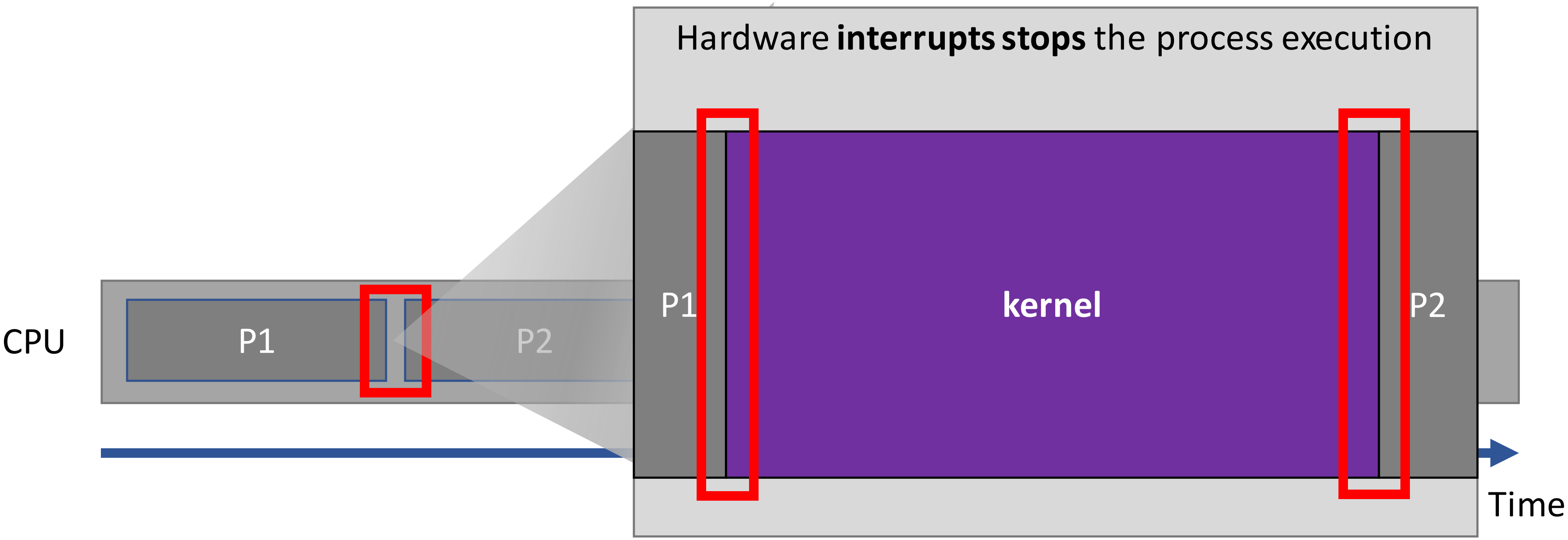
- How processes are switched during their execution?



Hardware **interrupts stops** the process execution

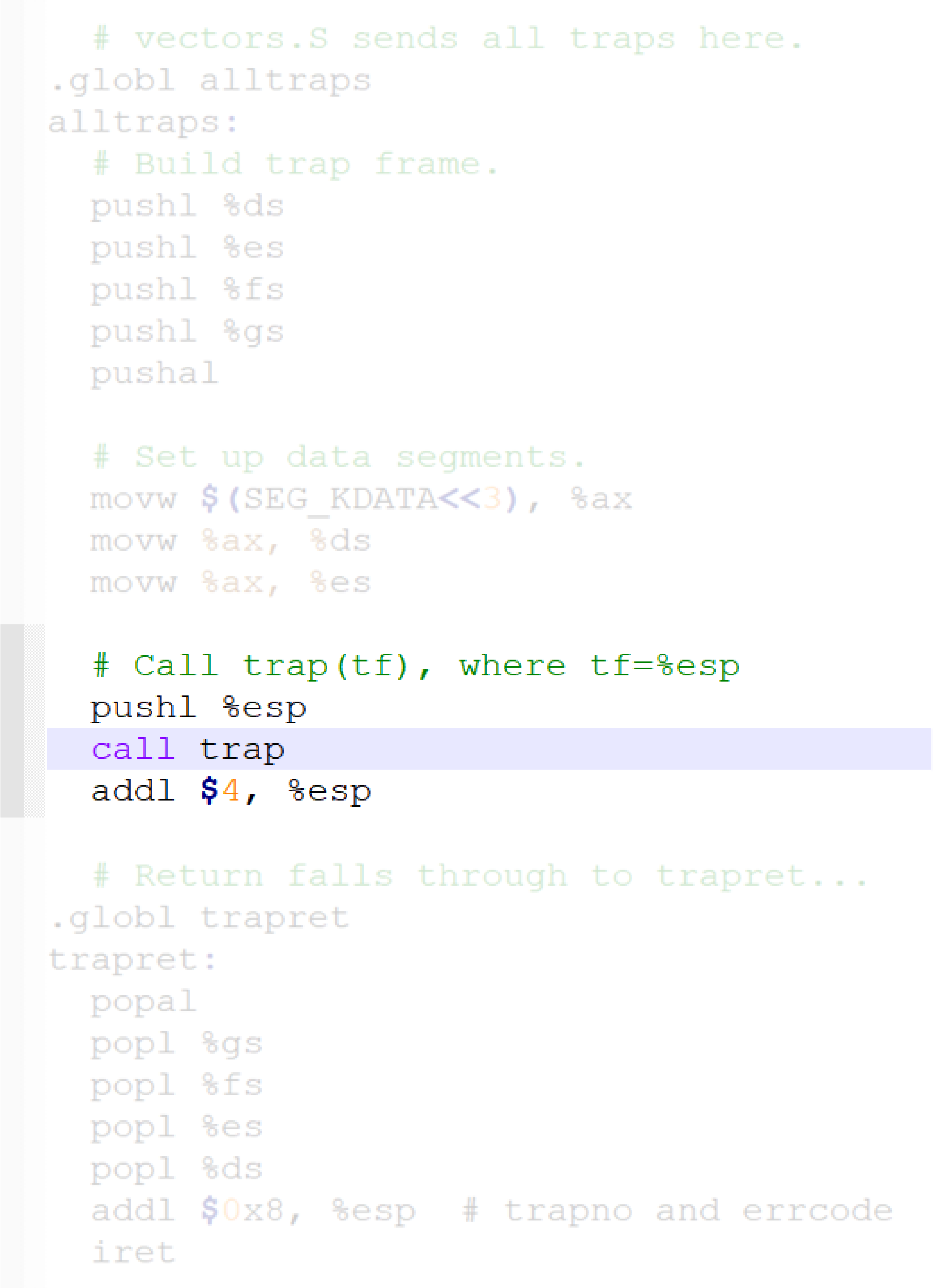
Scheduling of processes

- How processes are switched during their execution?



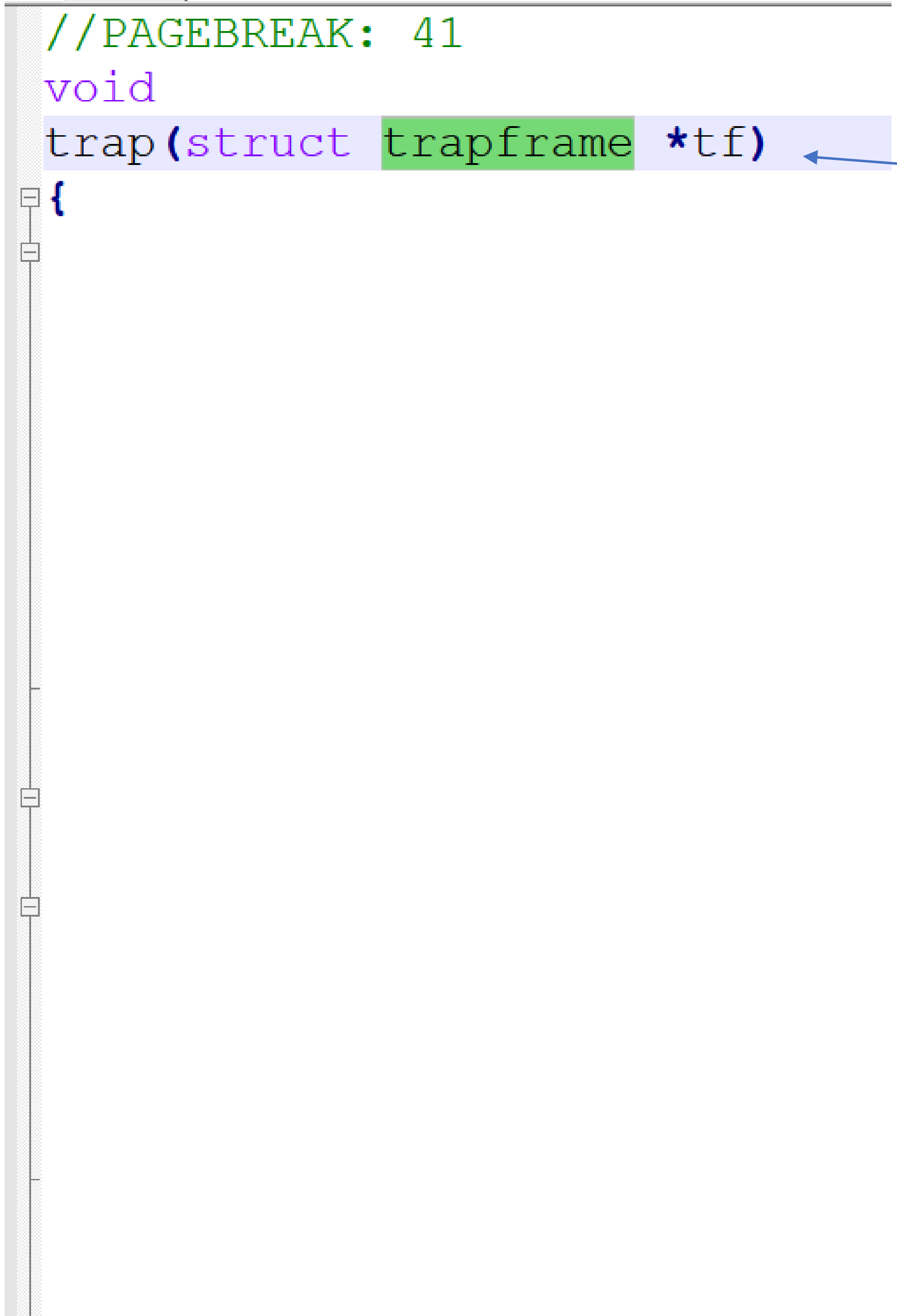
Let's take a deeper look at how
the interrupts work!

- **trapasm.S** file



```
  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

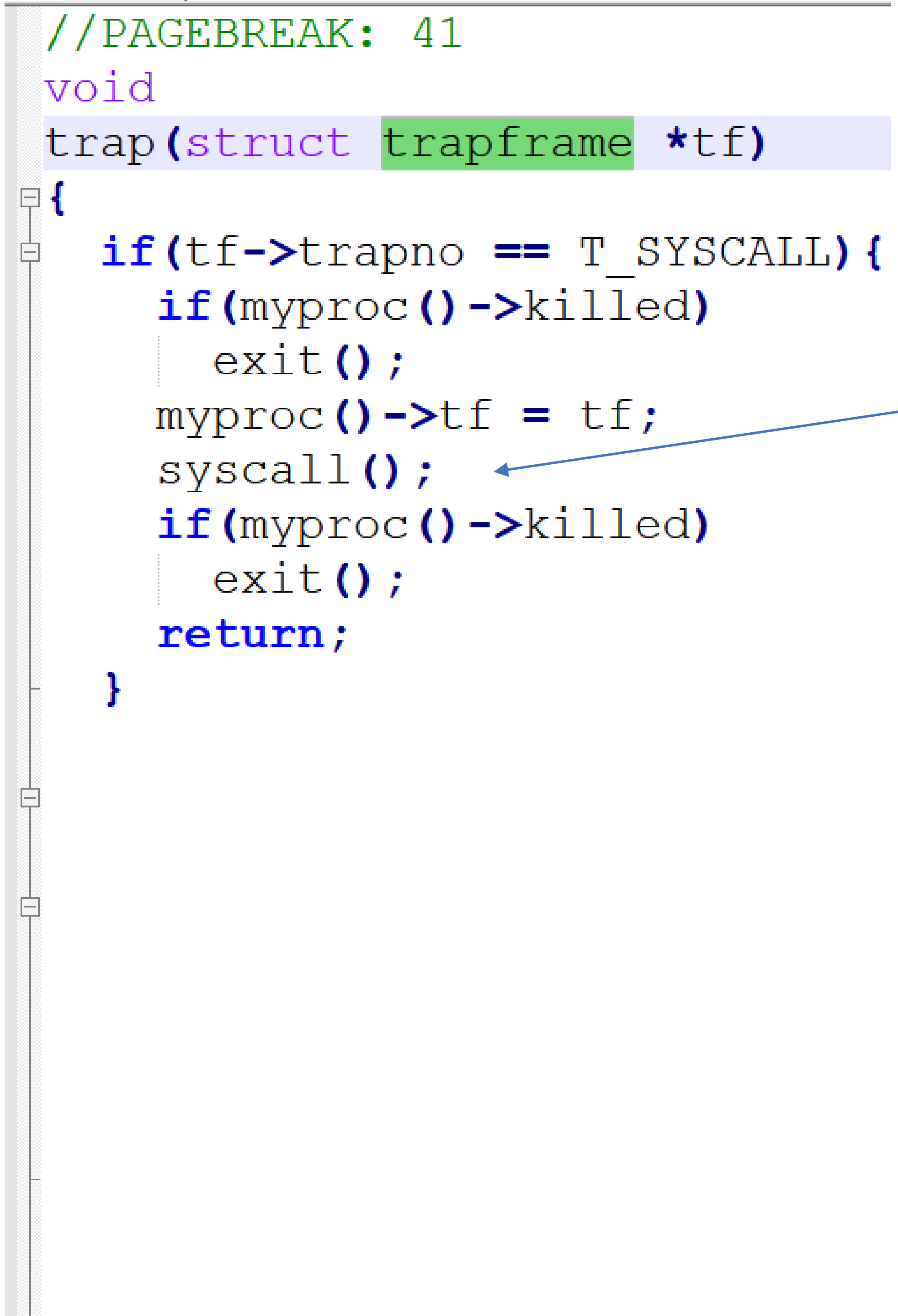
- **trap.c** file



```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
```

Trapframe contains the process data

- **trap.c** file

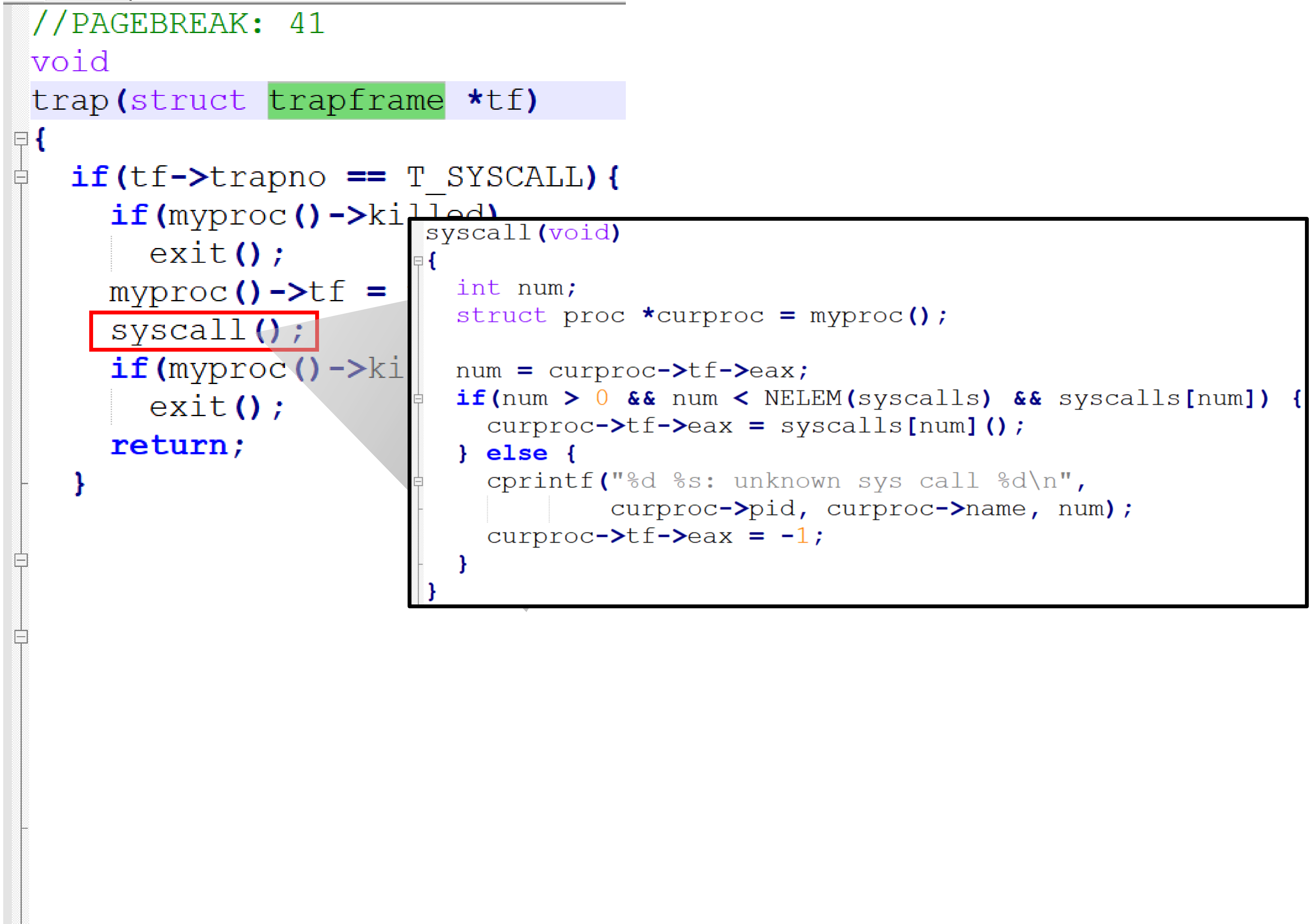


```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

Call syscall! (Lab 1)

A user syscall cause a interrupt!

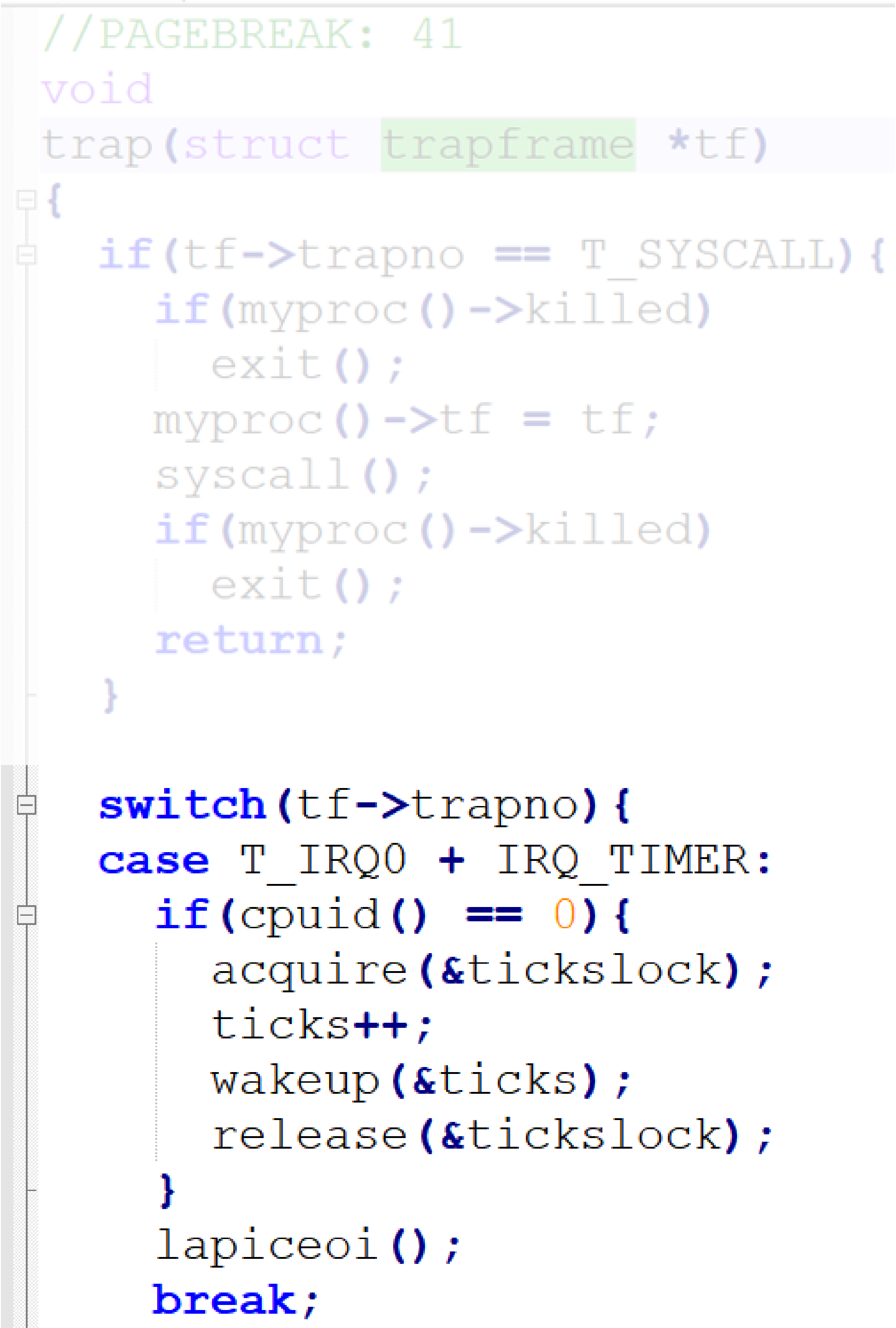
- **trap.c** file



```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf =
    syscall();
    if(myproc()->ki
      exit();
    return;
  }
```

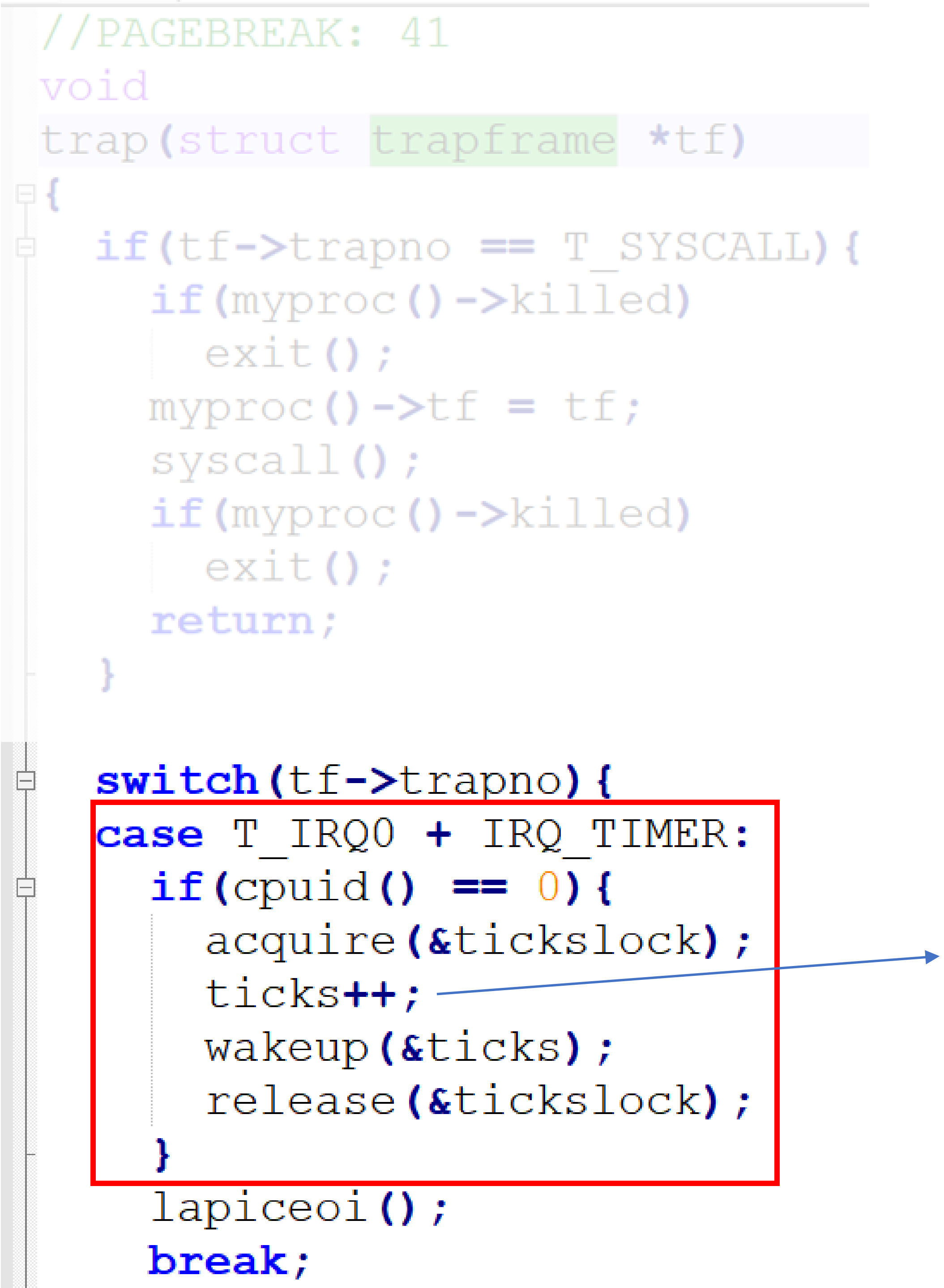
```
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

- **trap.c** file



```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

- **trap.c** file



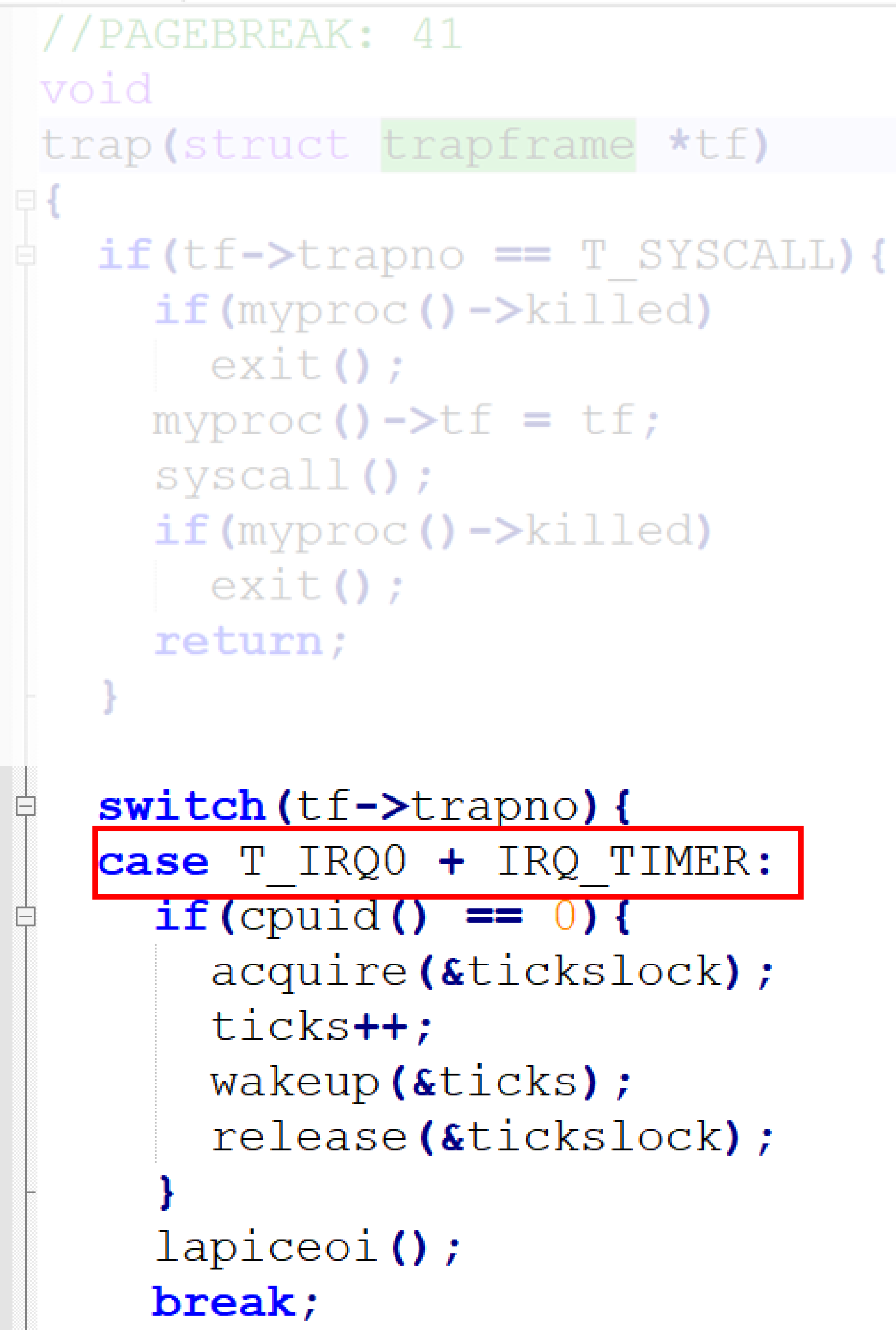
```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

Timer interrupt

Incrementing **ticks**

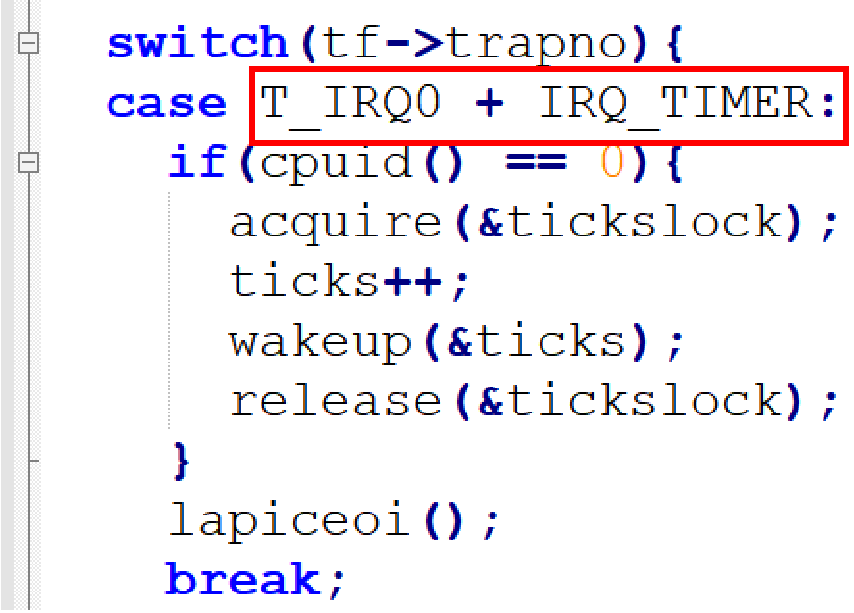
Allow time keeping

- **trap.c** file



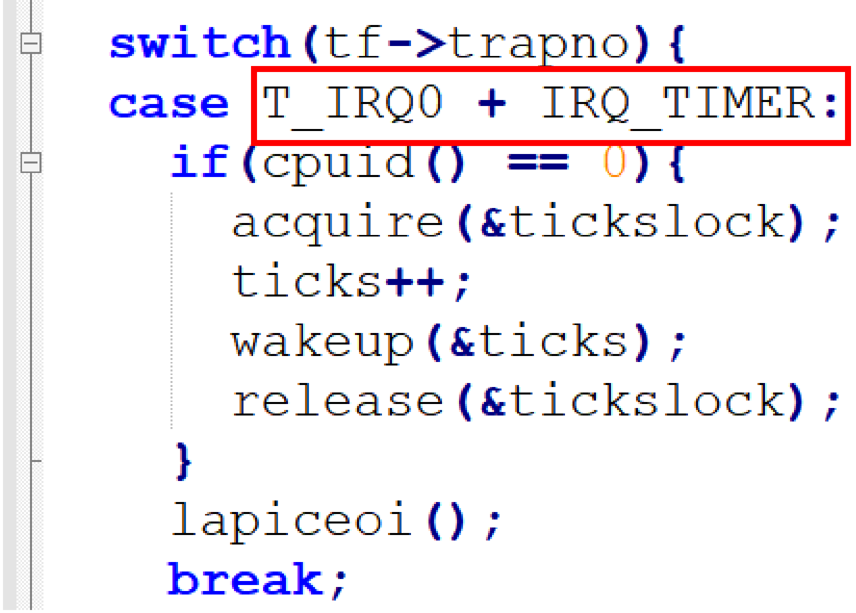
```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

- **trap.c** file



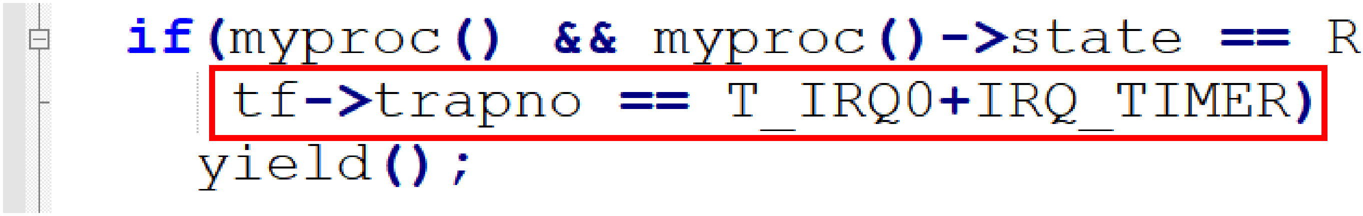
```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

- **trap.c** file



```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

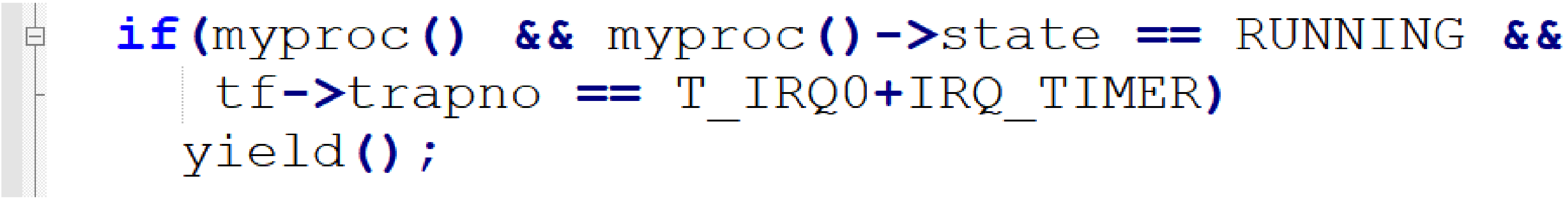
⋮



```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

Clock interrupts update cpu ticks **and** attempts to rescheduled a new process!

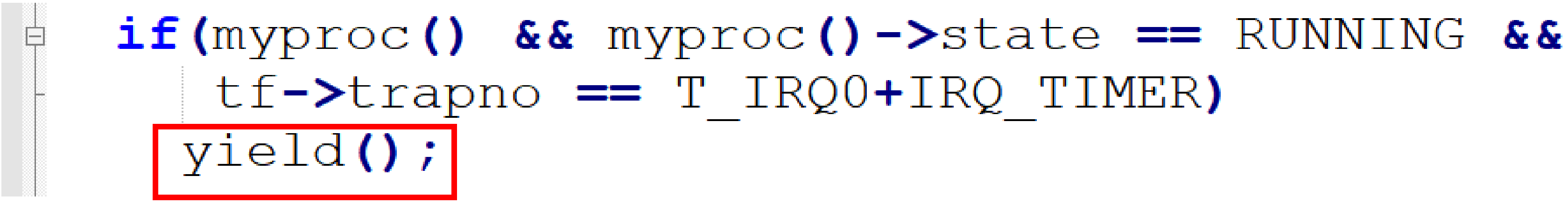
- **trap.c** file



```
if(myproc()  && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

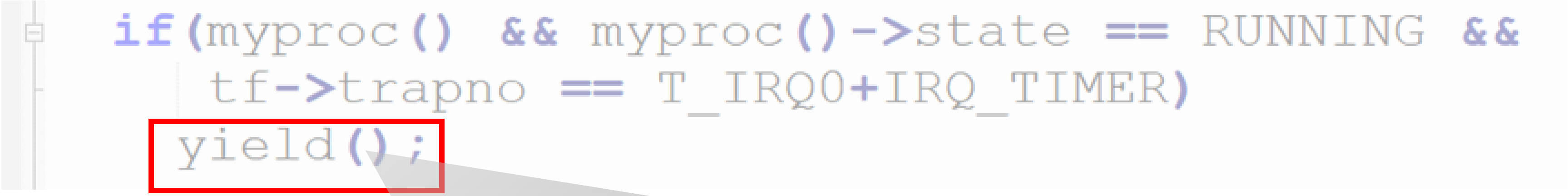
Let's take a deeper
look at yield()

- **trap.c** file

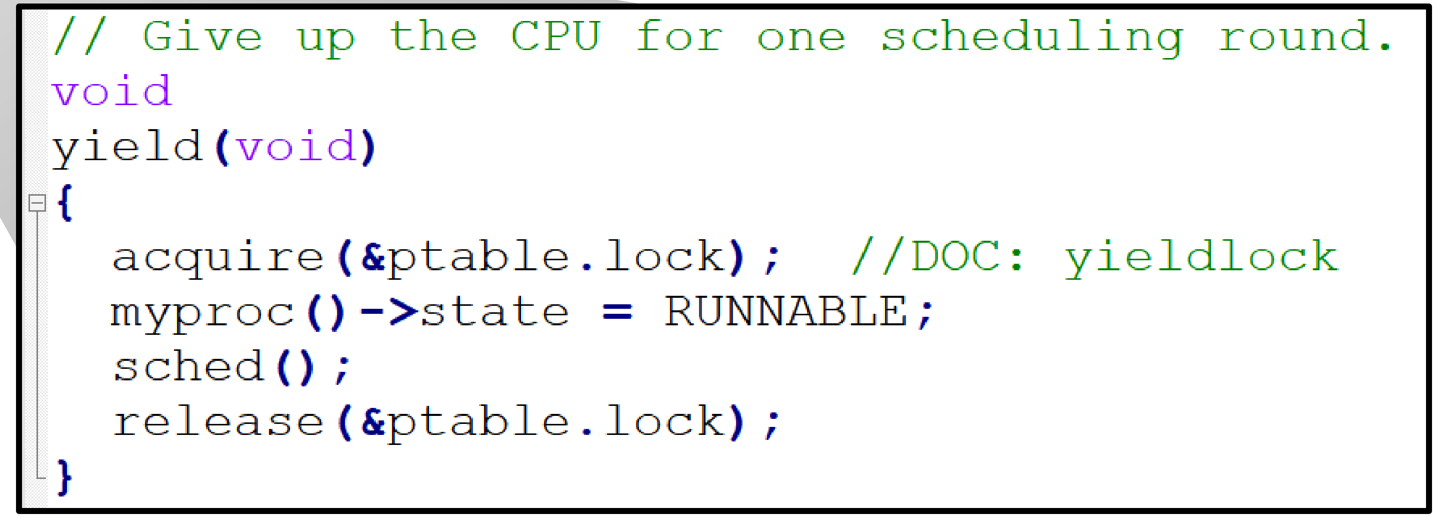


```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

- **trap.c** file

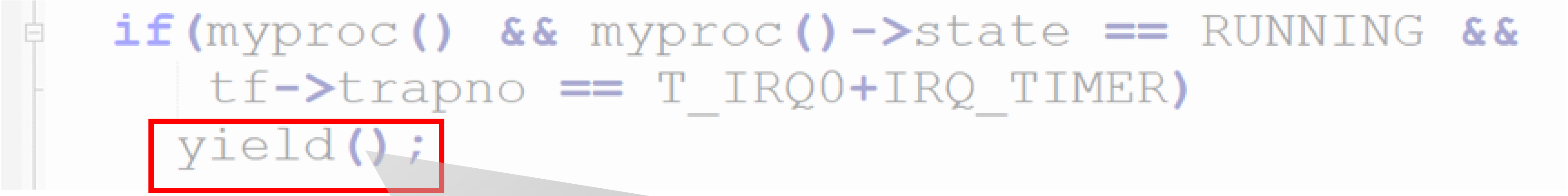


```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

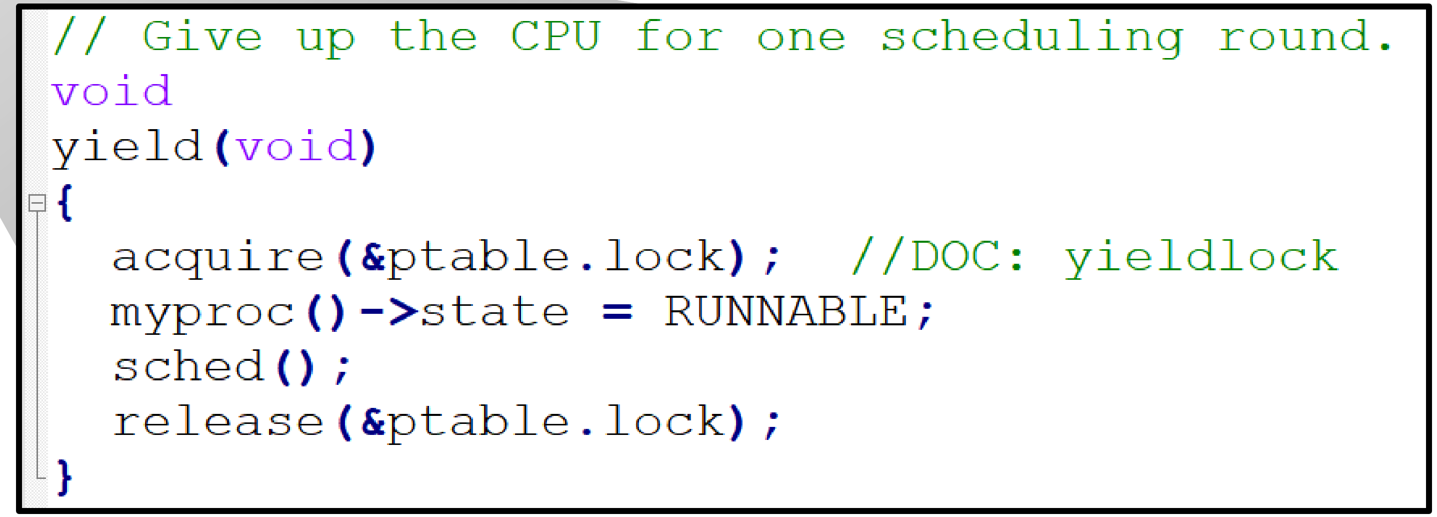


```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

- **trap.c** file



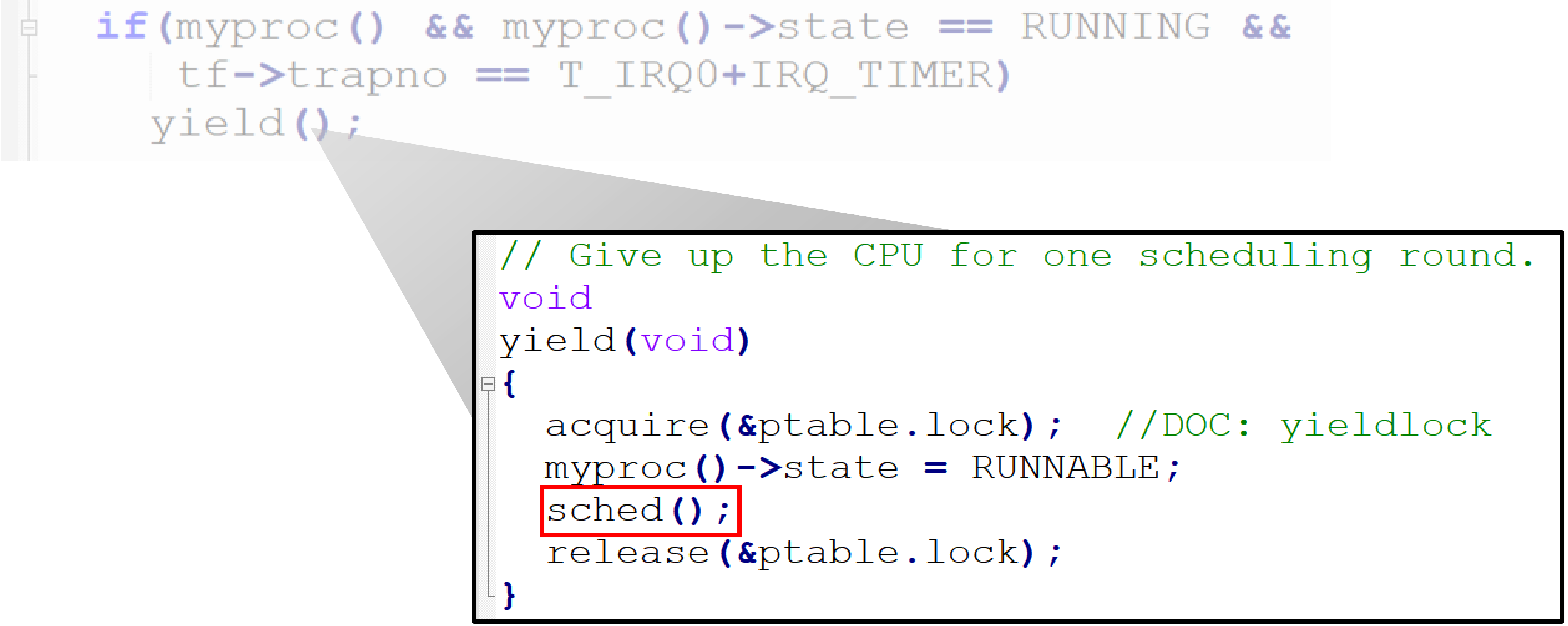
```
if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
   yield();
```



```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

Why do we change the process state to **runnable**?

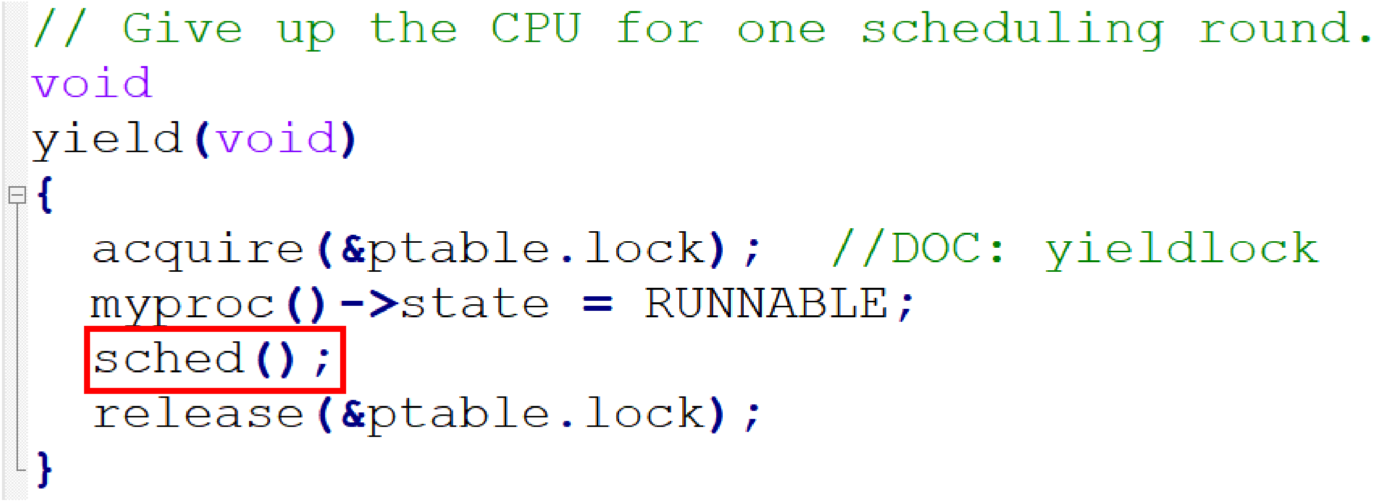
- **trap.c** file



```
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
```

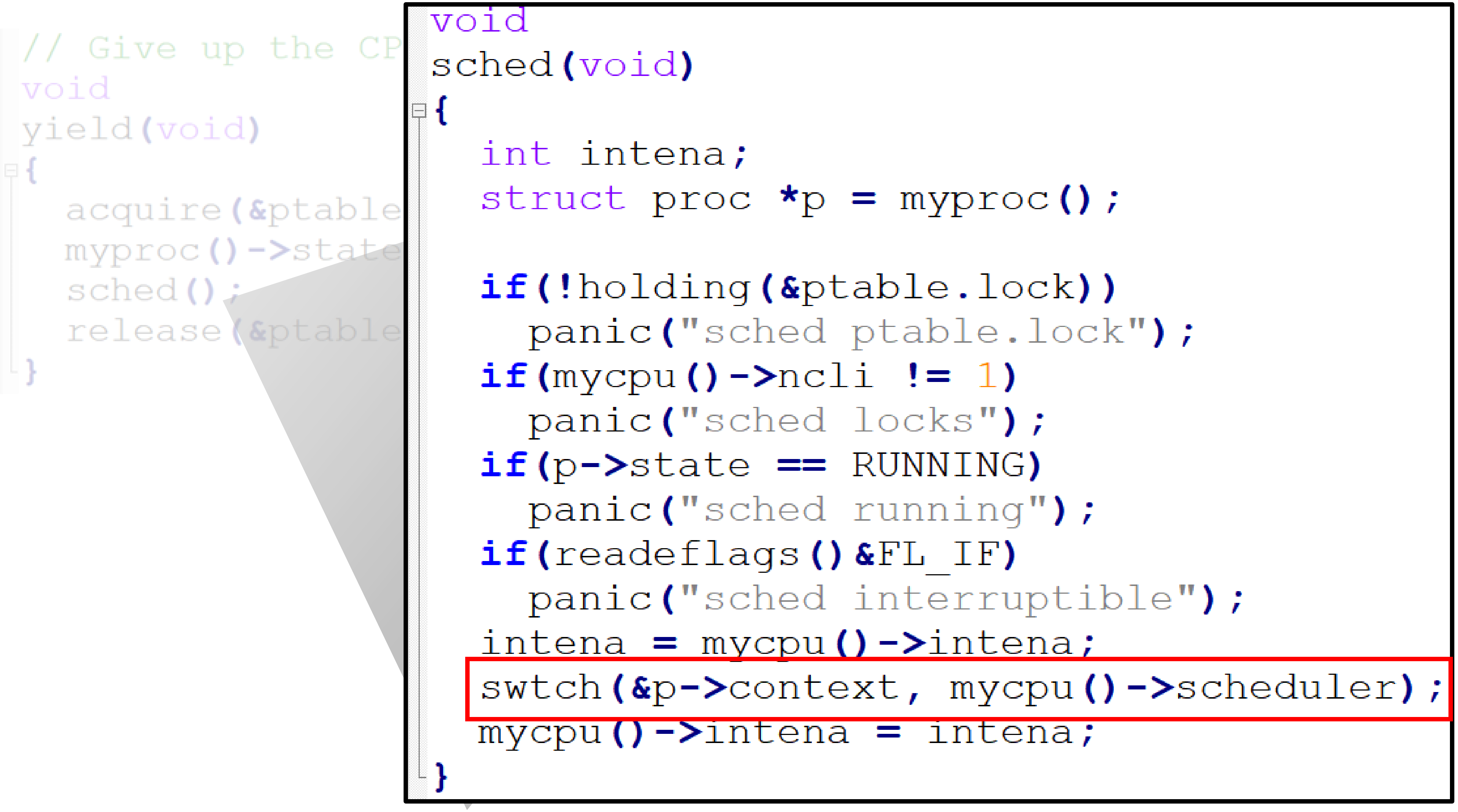
```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

- **trap.c** file



```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

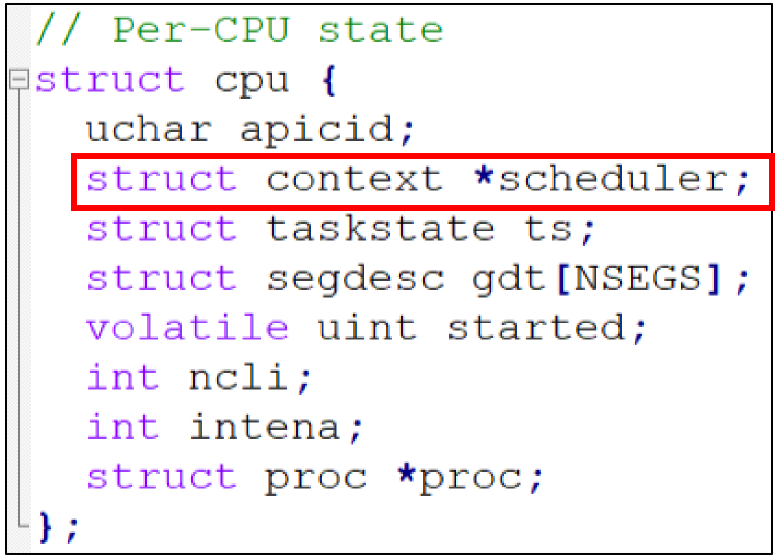
- **trap.c** file



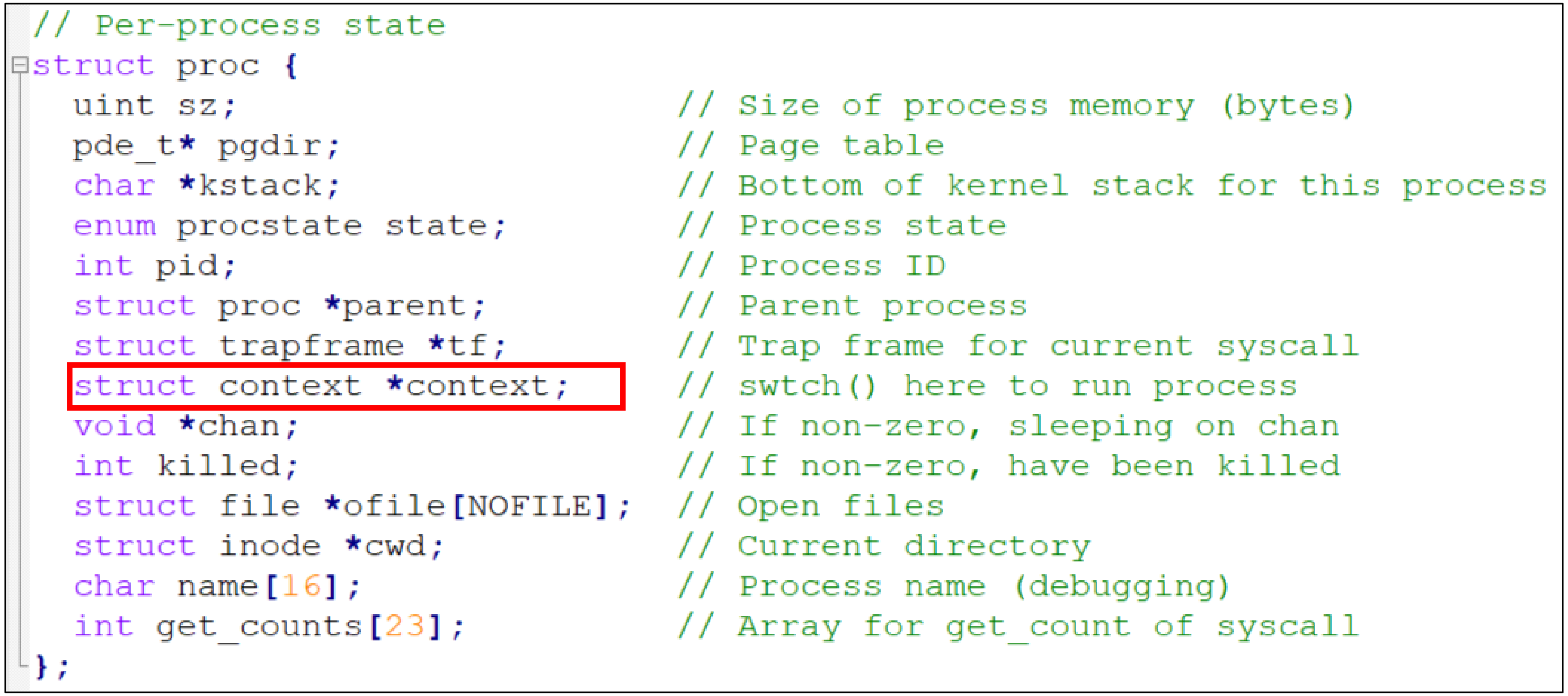
```
// Give up the CP
void
yield(void)
{
  acquire(&ptable
  myproc()->state
  sched();
  release(&ptable
}
```

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

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

- **proc.h** file

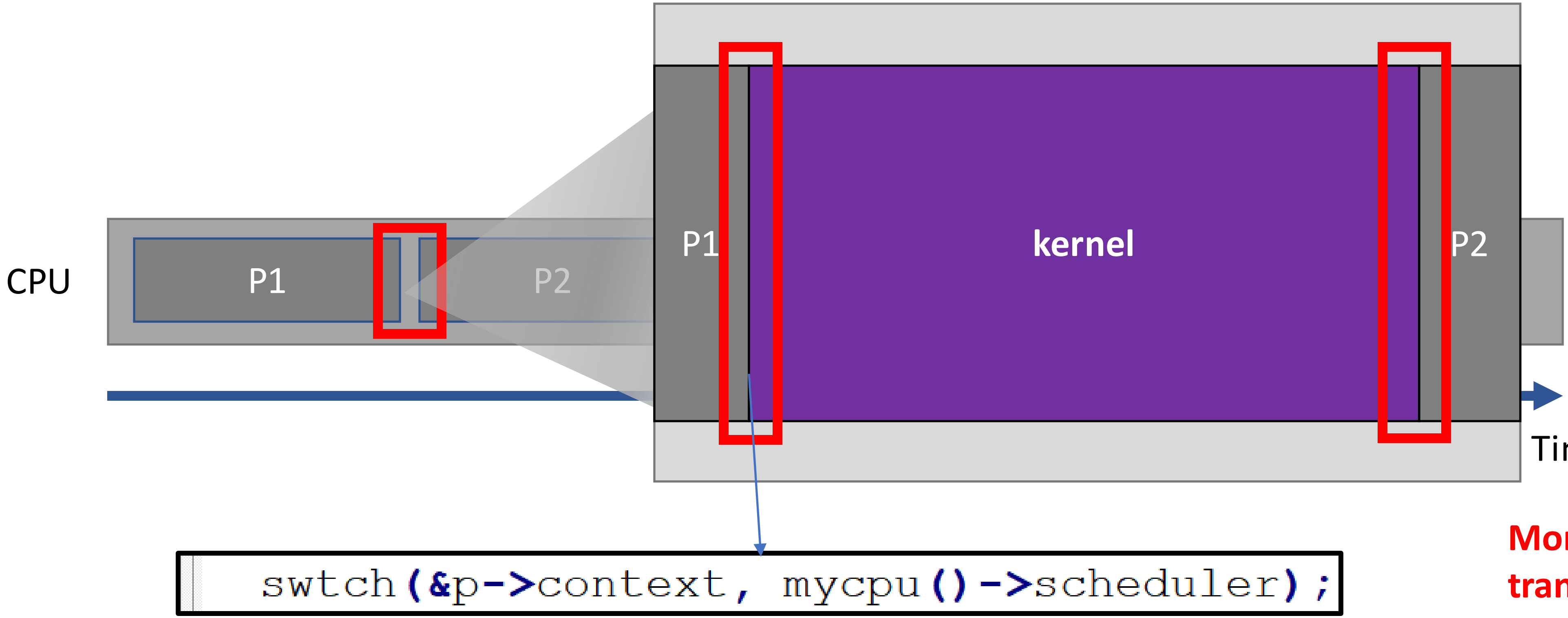


```
// Per-CPU state
struct cpu {
  uchar apicid;
  struct context *scheduler;
  struct taskstate ts;
  struct segdesc gdt[NSEGS];
  volatile uint started;
  int ncli;
  int intena;
  struct proc *proc;
};
```



```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *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
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int get_counts[23];          // Array for get_count of syscall
};
```

Scheduling of processes

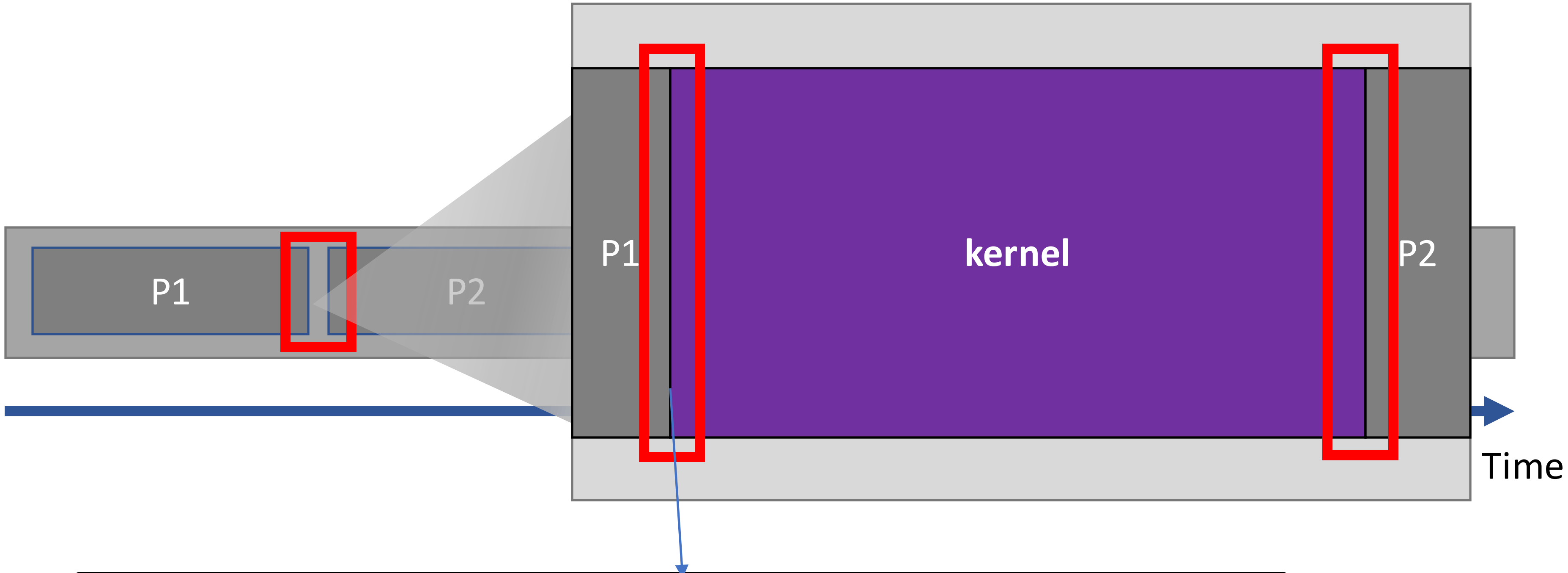


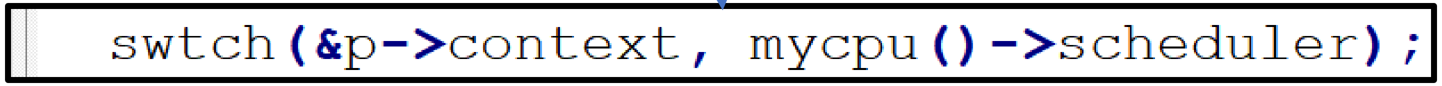
```
swtch(&p->context, mycpu()->scheduler);
```

Moment of transition into the kernel!

Scheduling of processes

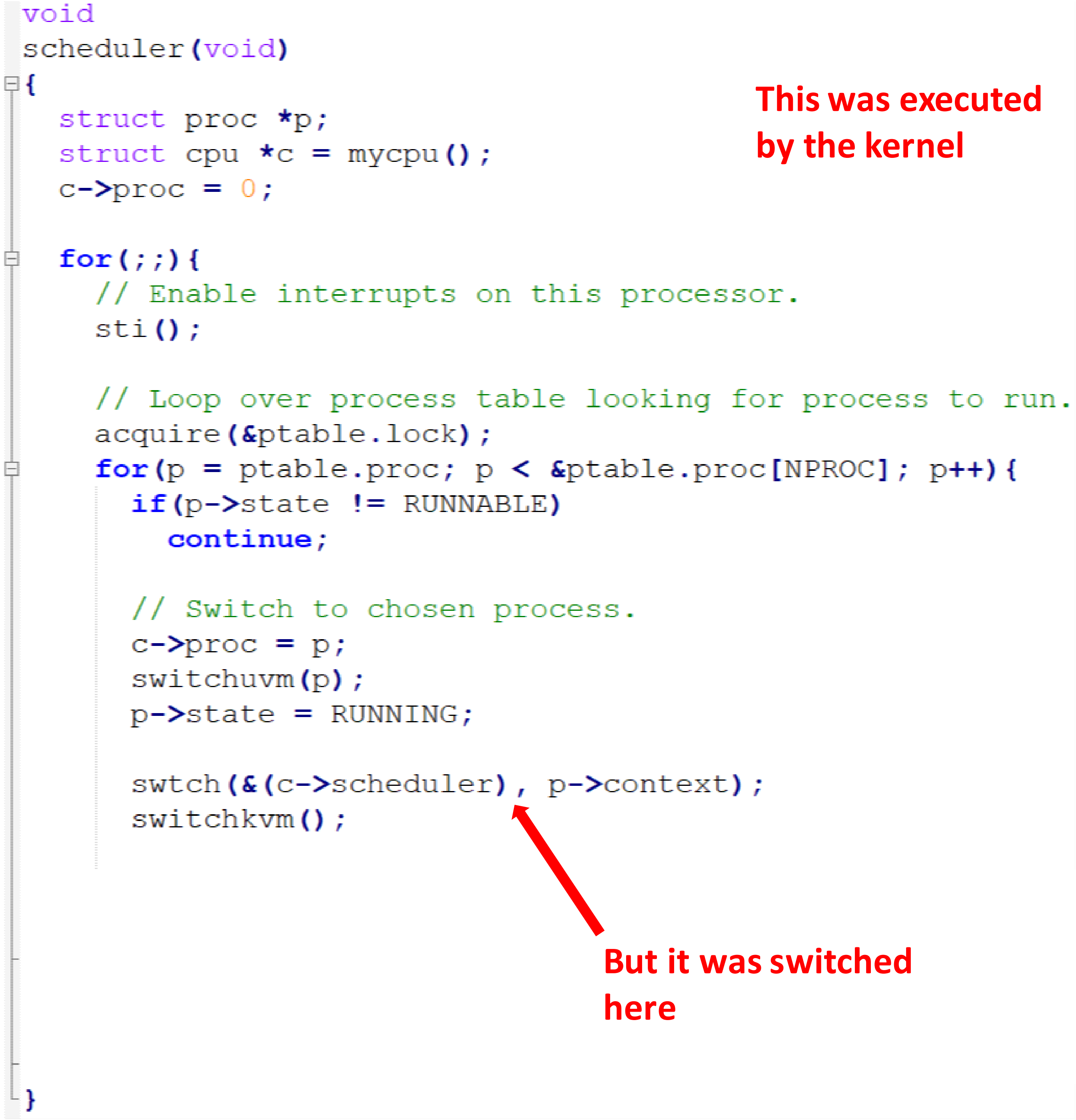
1. What was the context of the trap() execution ?





```
swtch(&p->context, mycpu()->scheduler);
```

- **proc.c** file

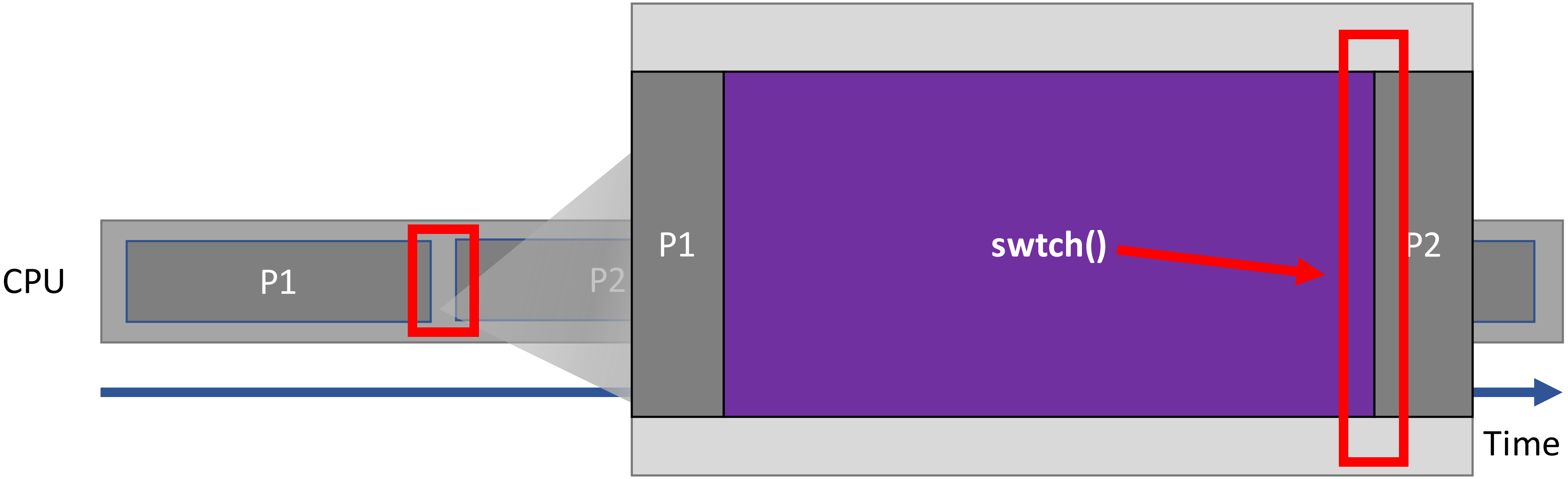


```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();




}
```

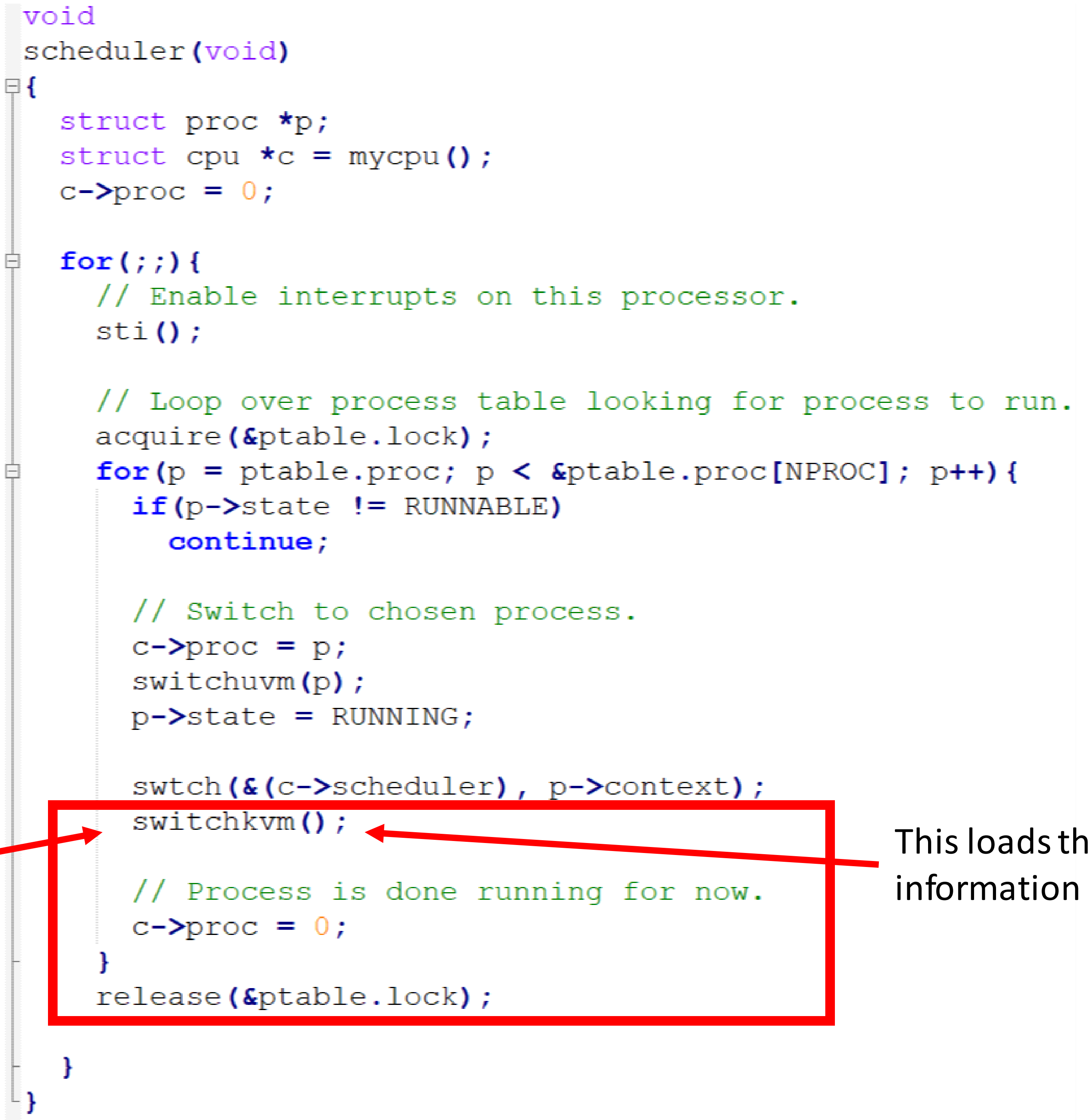
This was executed by the kernel

But it was switched here

Scheduling of processes



- **proc.c** file

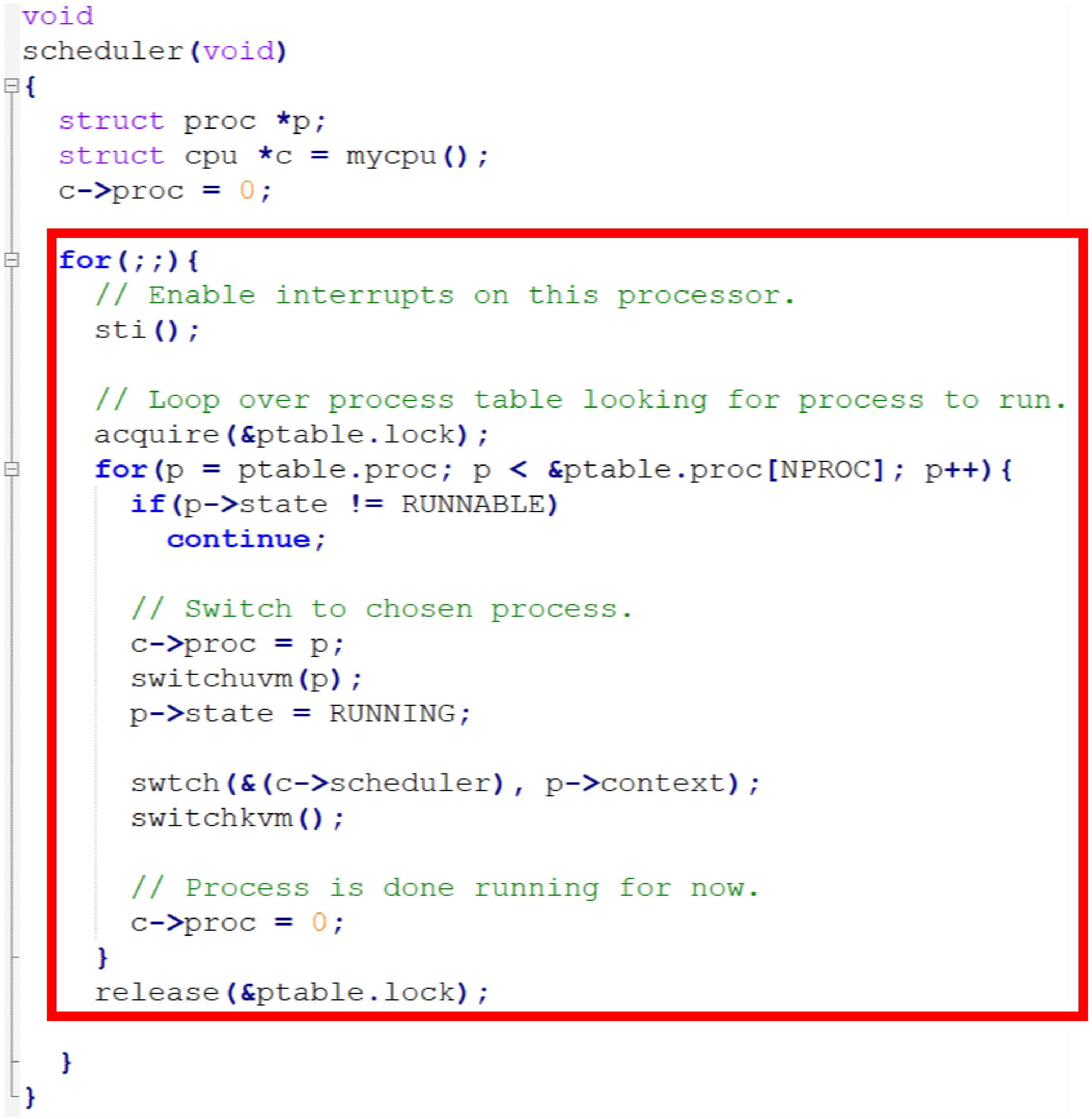


```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

The kernel **starts from here** since it stopped at the previous line!

This loads the kernel’s information

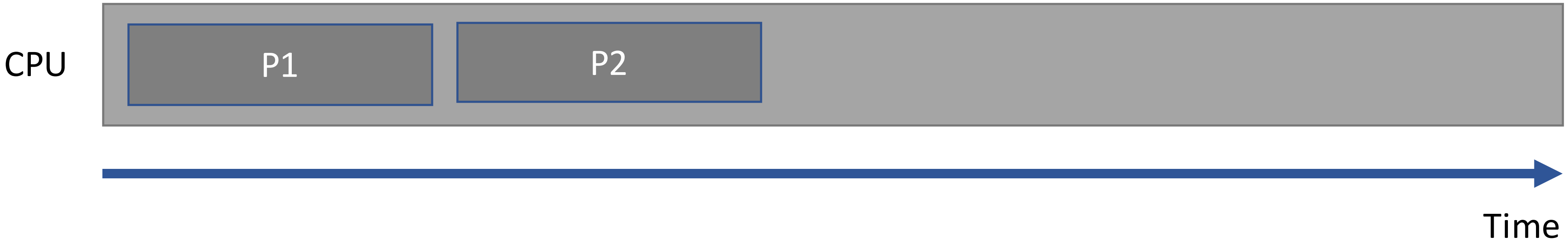
- **proc.c** file



```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

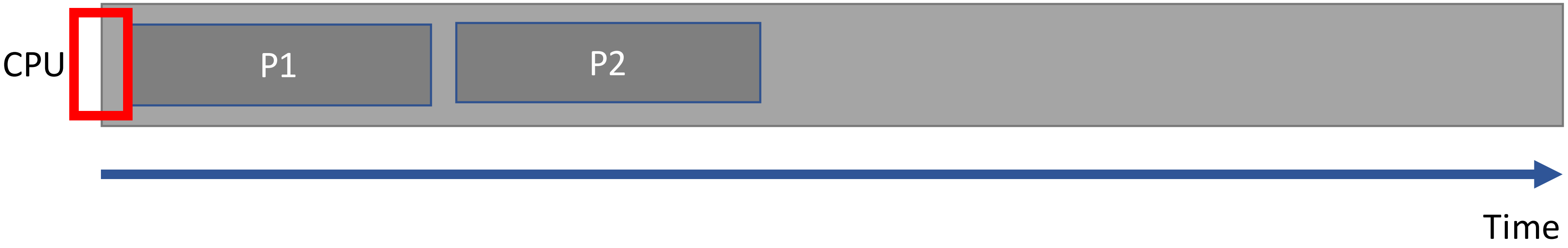
If this loop is infinite and never breaks **when did it start?**

Scheduling of processes



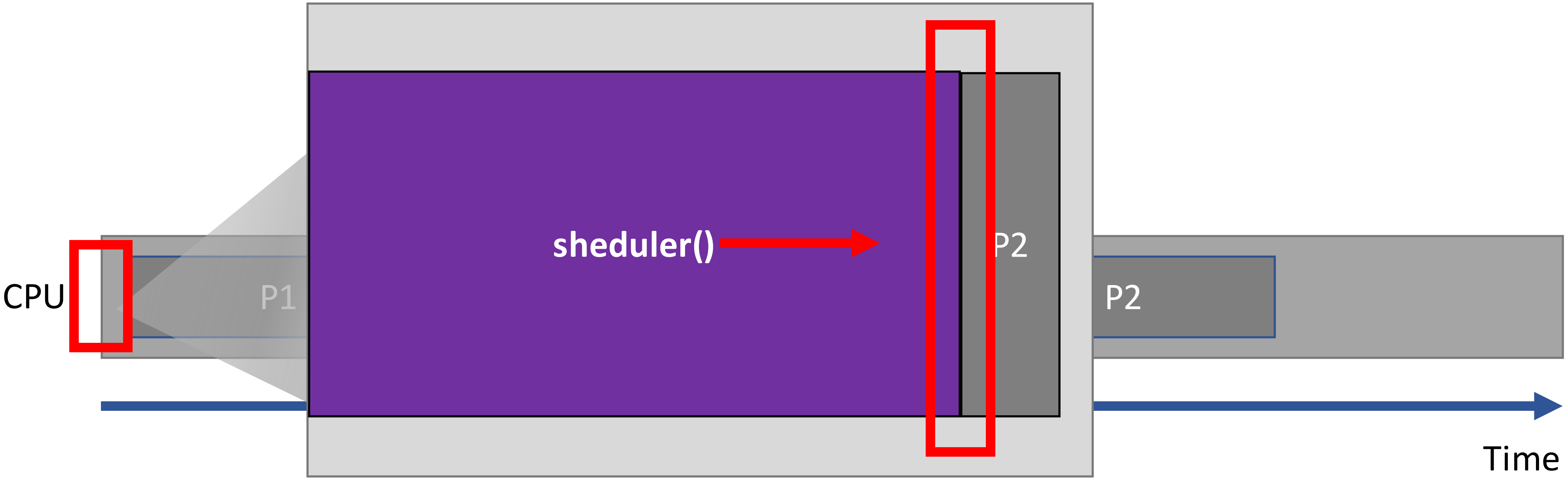
Scheduling of processes

What is the first program the executes on system boot?



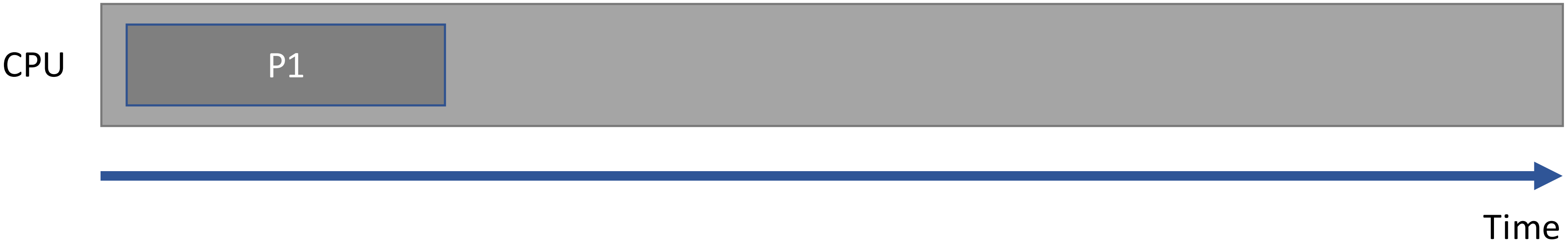
Scheduling of processes

The scheduler started on the system boot process.



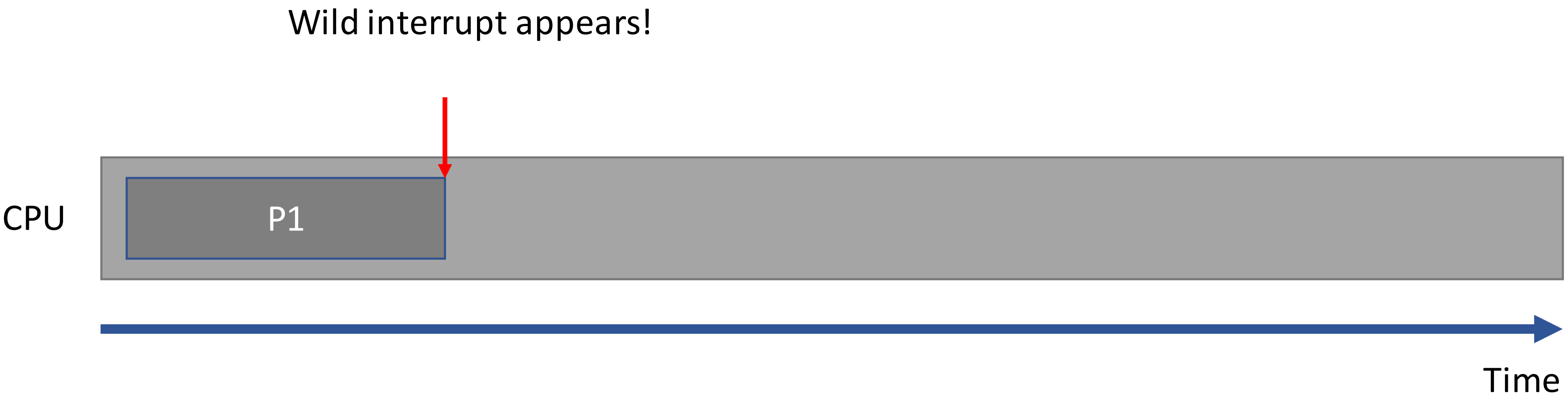
The scheduler eventually choses a process to run.

Scheduling of processes

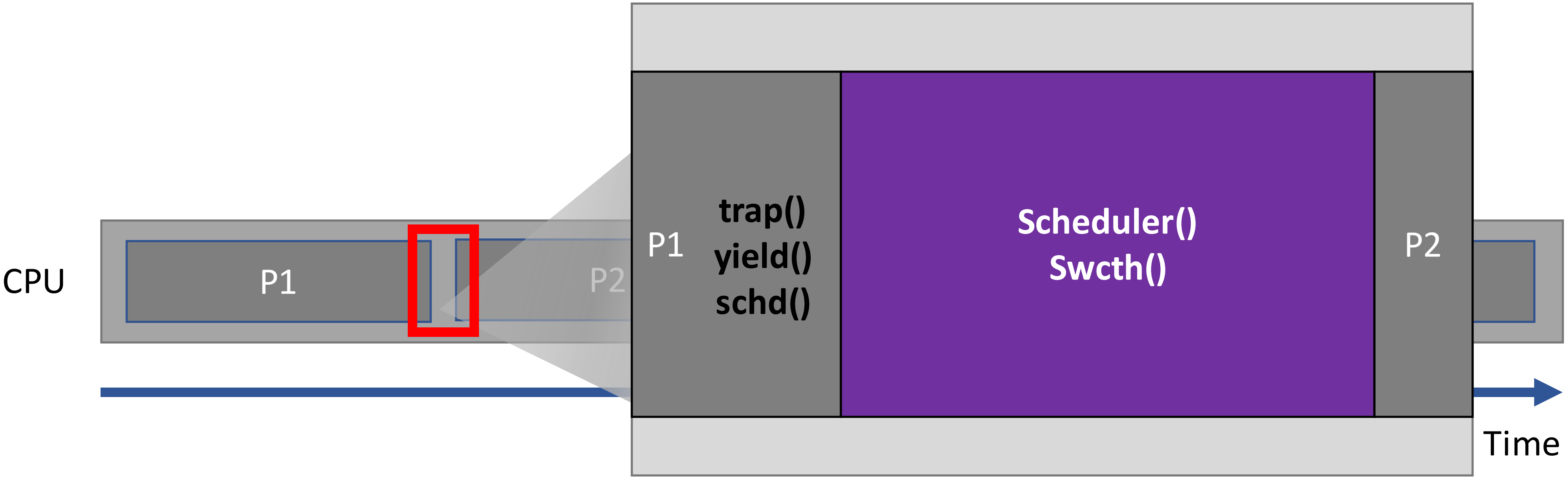


Scheduling of processes

Interrupts will stop the process at any point!



Scheduling of processes



Lab 3 – Priority-based scheduler for XV6

- The valid priority for a process is in the range of 0 to 200.
- The smaller value represents the higher priority.
- Default priority for a process is 50.
- proc.h:
  - Add an **integer** field called ***priority*** to struct proc.
- proc.c:
  - allocproc function:
    - Set the default priority for a process to 50
  - Scheduler function:
    - Replace the scheduler function with your implementation of a priority-based scheduler.

Lab 3 – part 2: add a syscall to set priority

- Add a new syscall, ***setpriority***, for the process to change its priority.
- Changes the current process’s priority and returns the old priority.
- Review lab1 to refresh steps to add a new syscall.