



CS 1550

Week 8 – Lab 3

Priority Scheduling with xv6

Teaching Assistant

Henrique Potter

CS 1550 – Lab 3

- **Due:** Monday, March 9th @11:59pm

Scheduling of processes

- Important feature of OS's is allowing concurrent execution of processes
- Better utilization of resources
 - While a process waits for I/O another one can execute

Scheduling of processes

- Important feature of OS's is allowing concurrent execution of processes
- Better utilization of resources
 - While a process waits for I/O another one can execute
- In **xv6**, processes are scheduled in a round-robin fashion

The logo for xv6, consisting of the text "xv6" in a bold, yellow, sans-serif font, centered within a solid black square.

xv6

Scheduling of processes

- Important feature of OS's is allowing concurrent execution of processes
- Better utilization of resources
 - While a process waits for I/O another one can execute
- In **xv6**, processes are scheduled in a round-robin fashion

The logo for xv6, consisting of the text "xv6" in yellow on a black square background.

xv6

However, how
does the scheduler
work in xv6?

Scheduling of processes

- In xv6, an interrupt for the scheduler is generated on every clock tick



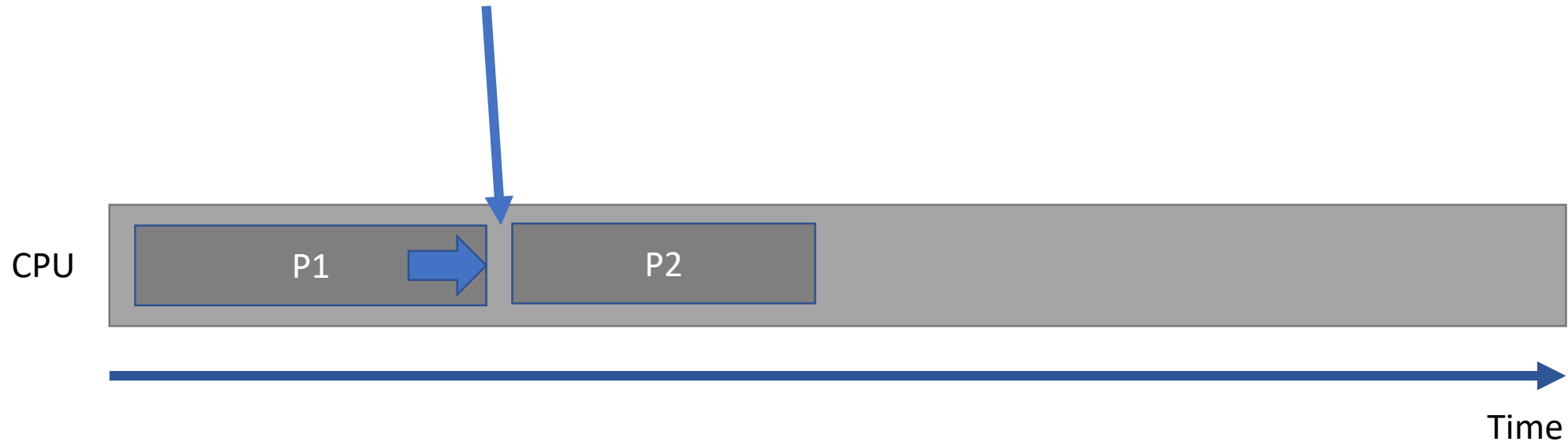
Scheduling of processes

- In xv6, an interrupt for the scheduler is generated on every clock tick
 - A 100Mhz processor does 100 Million clocks ticks per second



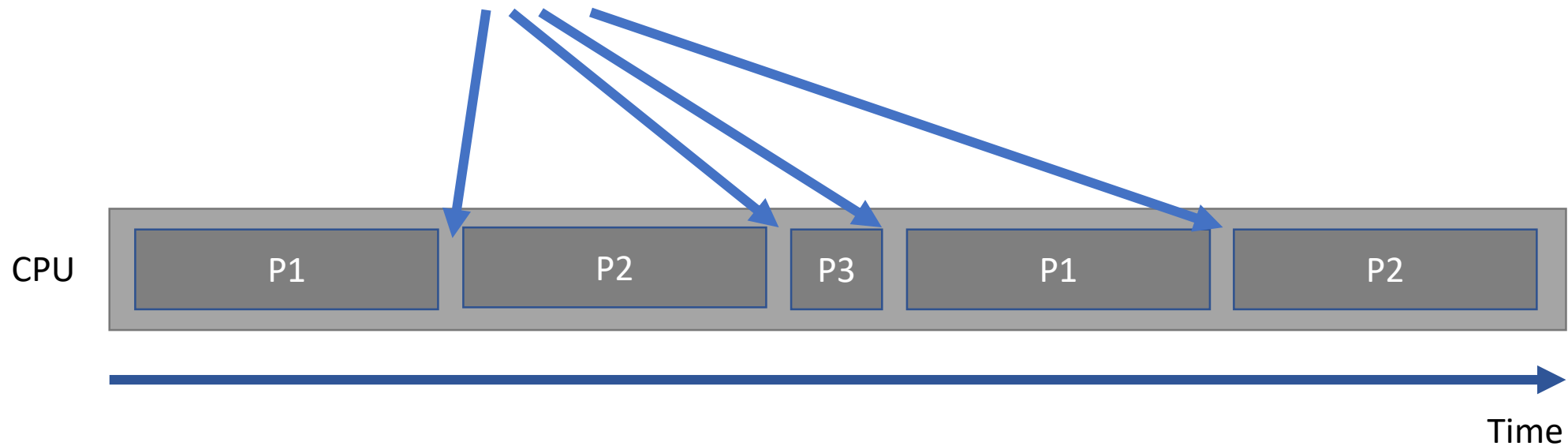
Scheduling of processes

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



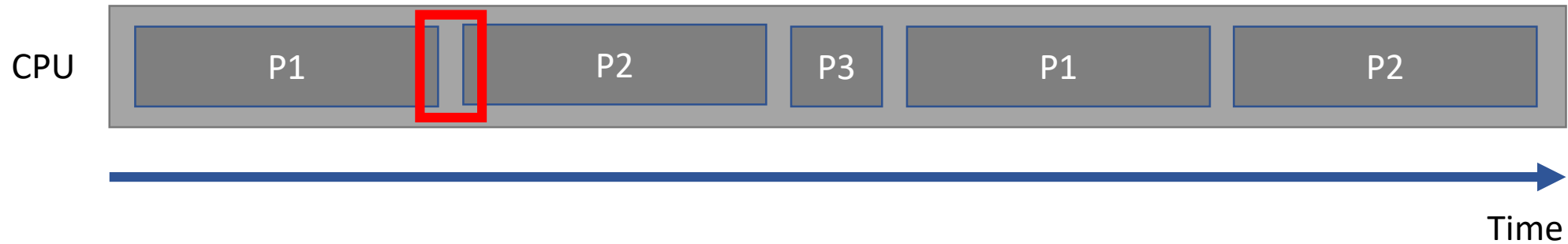
Scheduling of processes

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



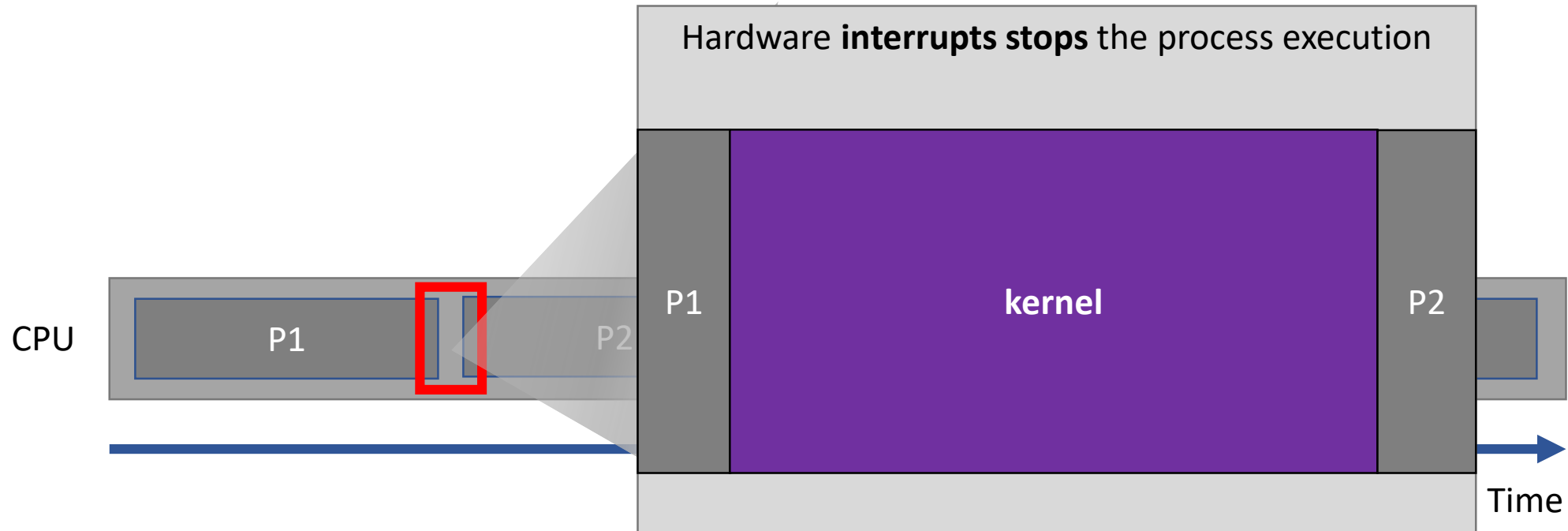
Scheduling of processes

- How processes are switched during their execution?



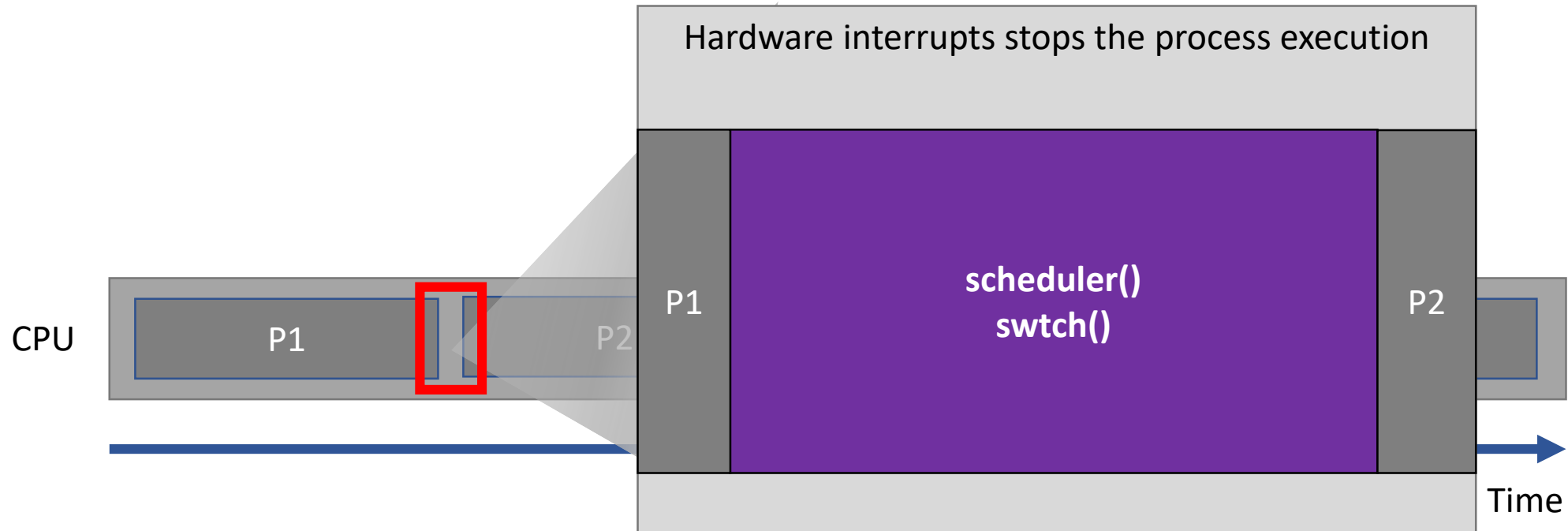
Scheduling of processes

- How processes are switched during their execution?



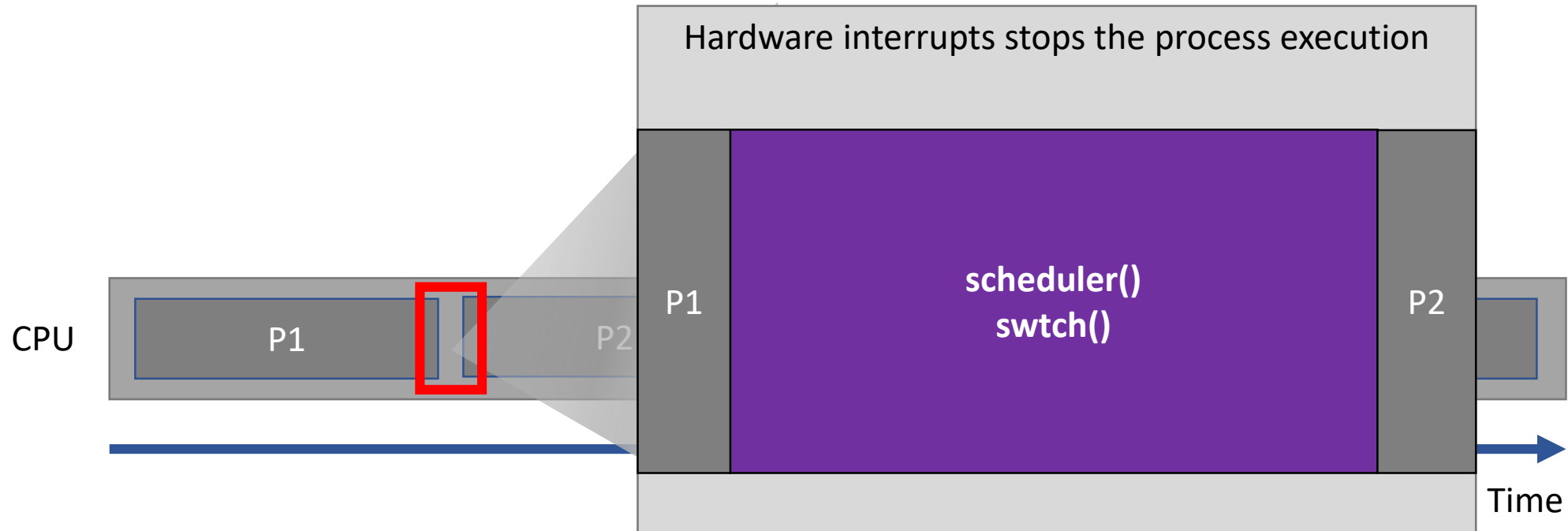
Scheduling of processes

- How processes are switched during their execution?



Scheduling of processes

- How processes are switched during their execution?



proc.c implements the
scheduler function

- **proc.c** file

```
void
scheduler(void)
{
}
}
```

- **proc.c** file

The process information

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

- **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; // switch() 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
};
```


- **proc.c** file

The process state information

The cpu state information

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

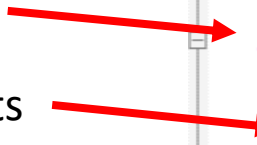
- **proc.c** file

Infinite loop

Enable interrupts

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

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

Two red arrows originate from the left side of the image. The first arrow, labeled 'Infinite loop', points to the 'for(;;){' line of the scheduler function. The second arrow, labeled 'Enable interrupts', points to the 'sti();' line within the same function.

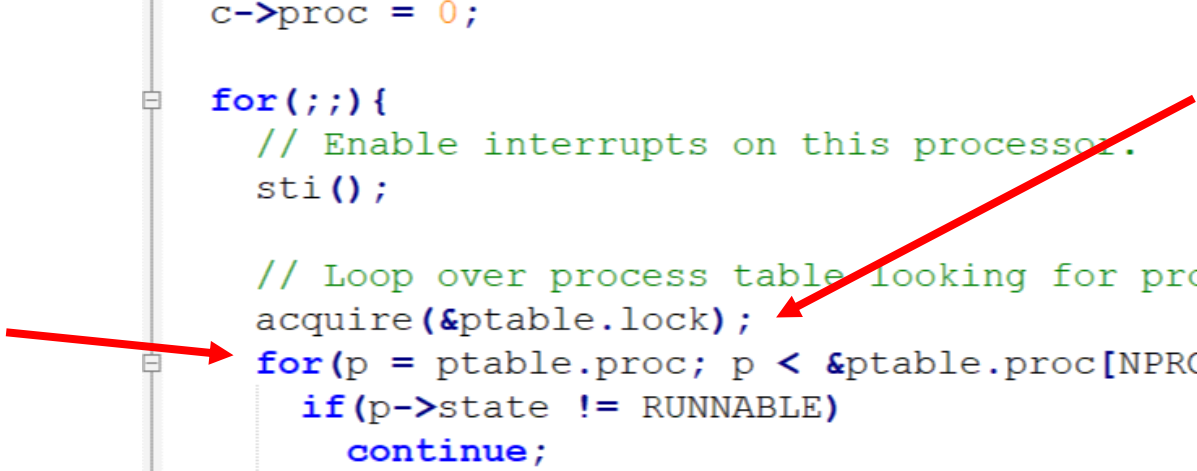
- **proc.c** file

```
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;
        }
    }
}
```



Before that get *ptable* lock

Loop over all the processes

- **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;
        }
    }
}
```

Pointer arithmetic!



- **proc.c** file

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

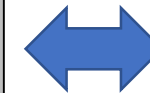
    for(;;){
        // Enable interrupts on this processor.
        sti();

        // 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;
        }
    }
}
```

Pointer arithmetic!



```
struct foobar *p;
p = 0x1000;
p++;
```



```
struct foobar *p;
p = 0x1000 + sizeof(struct foobar);
```

- **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;
            switchvm(p);
            p->state = RUNNING;
        }
    }
}
```

cpu process is set

This is what
myproc() returns

Loads the process page table

- **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;
            switchvm(p);
            p->state = RUNNING;

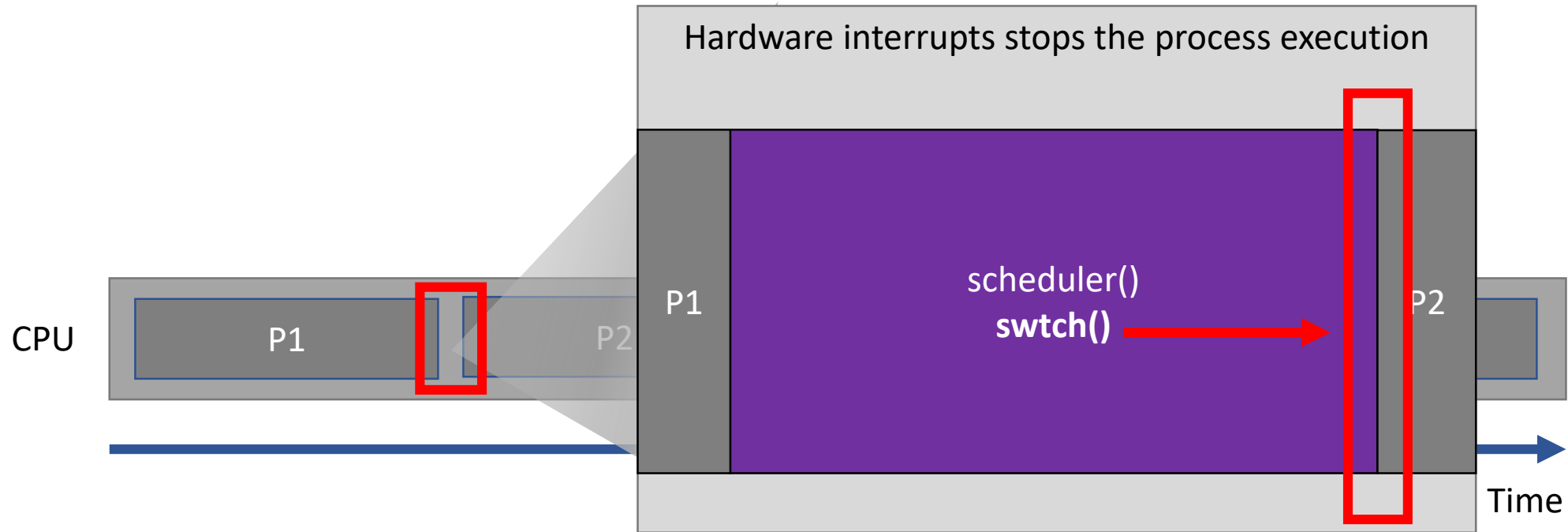
            swtch(&(c->scheduler), p->context);
            switchkvm();
        }
    }
}
```

Here the process is
switched to execute

The kernel execution will **stop here**

The process will **continue** from
wherever is stopped

Scheduling of processes



proc.c implements the
scheduler function

- **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;
            switchvm(p);
            p->state = RUNNING;

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

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

When a process is interrupted is starts from here

This loads the kernel's state information

- **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;
            switchvm(p);
            p->state = RUNNING;

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

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

This loop never ends



Yield in trap

- Yield:
 - Acquire the process table lock ptable.lock
 - Release any other locks it is holding
 - Update its own state (proc->state)
 - Call sched
- Force process to give up CPU on clock tick.
- IRQ stands for Interrupt Requests

```
//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;
    case T_IRQ0 + IRQ_IDE:
        ideintr();
        lapiceoi();
        break;
    case T_IRQ0 + IRQ_IDE+1:
        // Bochs generates spurious IDE1 interrupts.
        break;
    case T_IRQ0 + IRQ_KBD:
```

Yield in trap

- Yield:
 - Acquire the process table lock ptable.lock
 - Release any other locks it is holding
 - Update its own state (proc->state)
 - Call sched
- Force process to give up CPU on clock tick.
- IRQ stands for Interrupt Requests

In trap.c:

```
// Force process to give up CPU on clock tick.  
// If interrupts were on while locks held, would need to check nlock.  
if(myproc() && myproc()->state == RUNNING &&  
    tf->trapno == T_IRQ0+IRQ_TIMER)  
    yield();
```

Yield in trap

- Yield:
 - Acquire the process table lock ptable.lock
 - Release any other locks it is holding
 - Update its own state (proc->state)
 - Call sched
- Force process to give up CPU on clock tick.
- IRQ stands for Interrupt Requests

In trap.c:

```
// Force process to give up CPU on clock tick.  
// If interrupts were on while locks held, would need to check nlock.  
if(myproc() && myproc()->state == RUNNING &&  
    tf->trapno == T_IRQ0+IRQ_TIMER)  
    yield();
```

Yield in trap

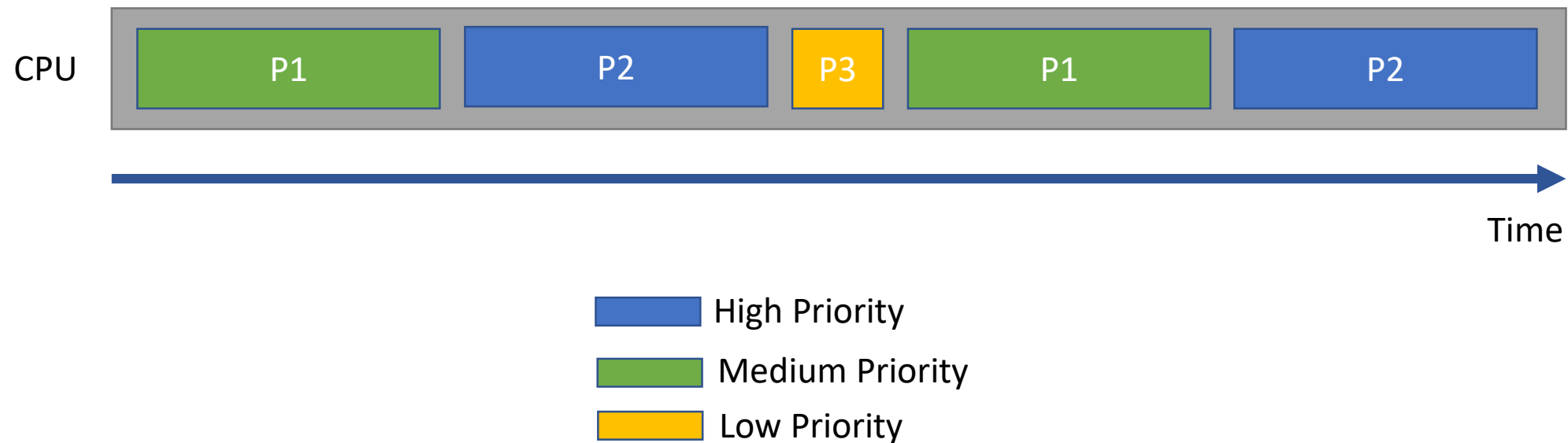
- Yield:
 - Acquire the process table lock ptable.lock
 - Release any other locks it is holding
 - Update its own state (proc->state)
 - Call sched
- Force process to give up CPU on clock tick.
- IRQ stands for Interrupt Requests

In proc.c:

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

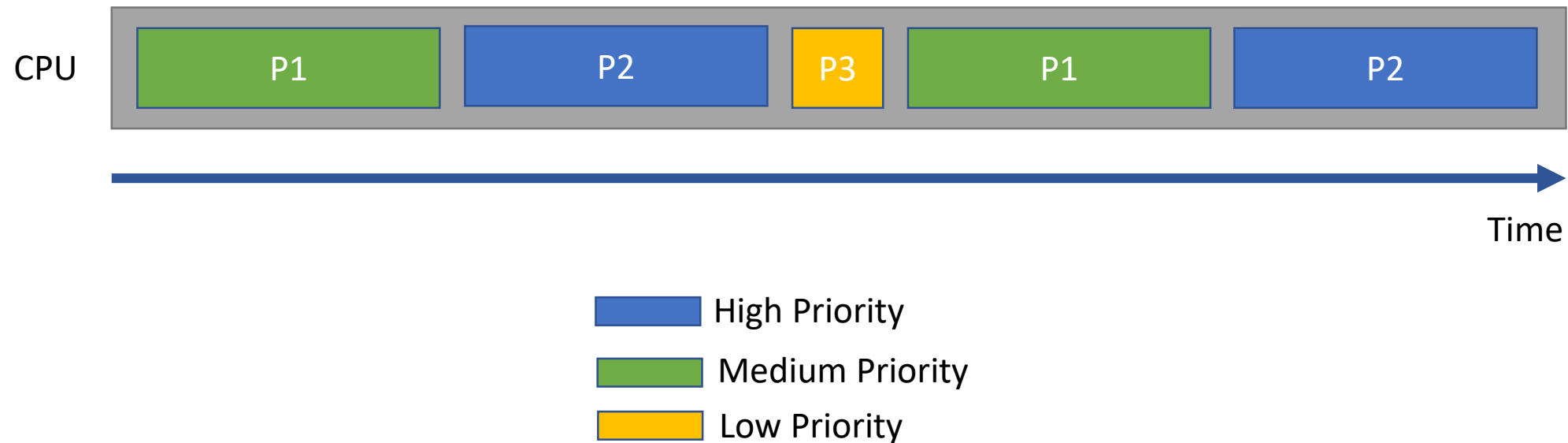
Priority scheduling of processes

- In lab 3 we will implement priority queue in xv6.



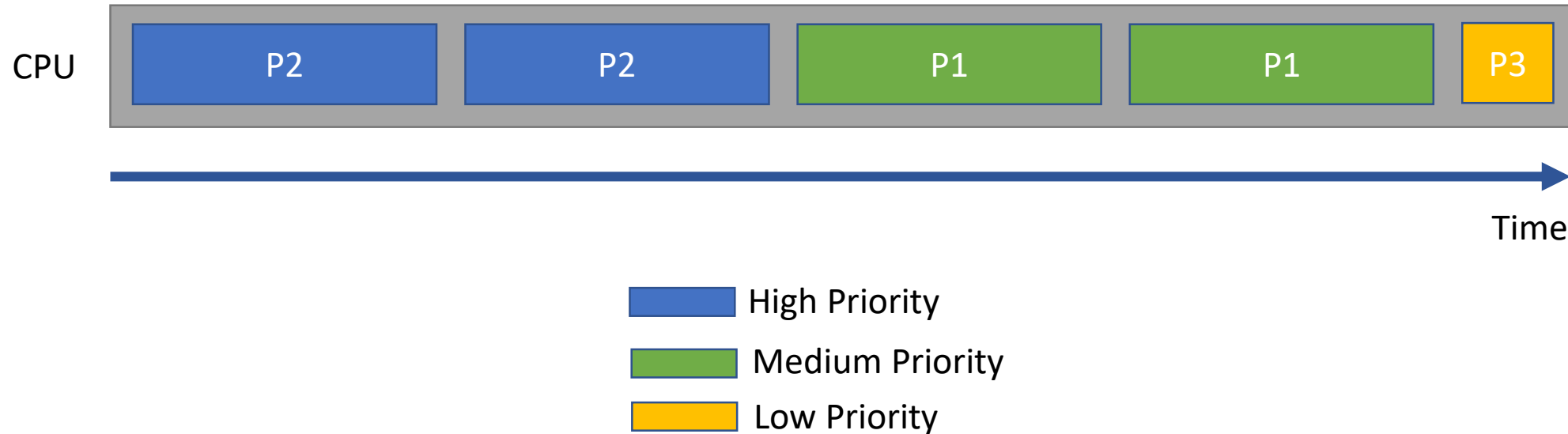
Priority scheduling of processes

- What if processes have different priorities?



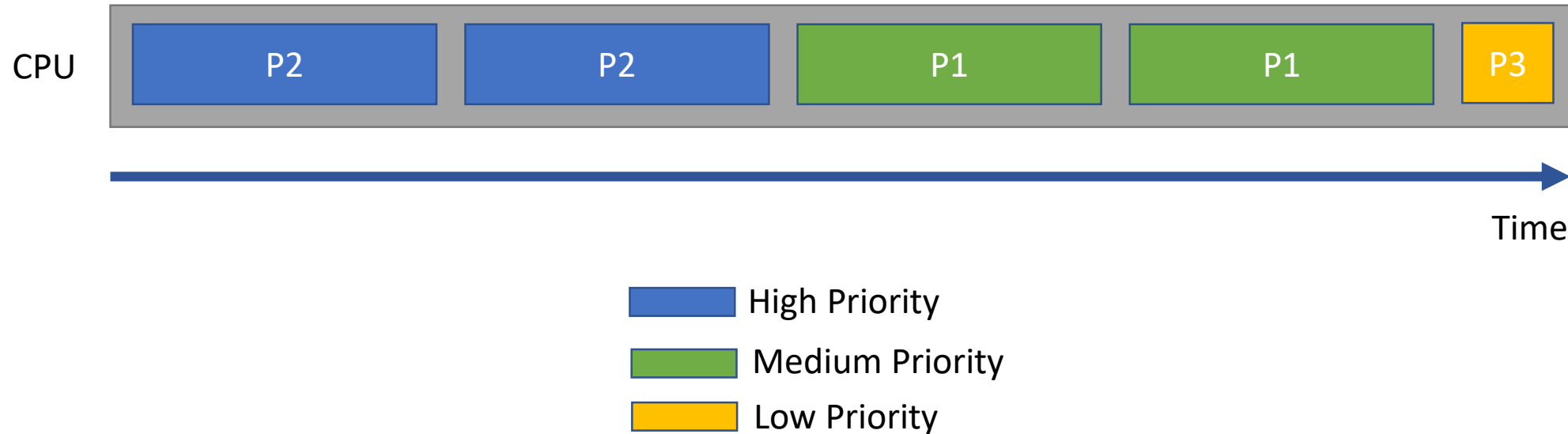
Priority scheduling of processes

- Let all the higher priority processes finish before moving to lower priority ones



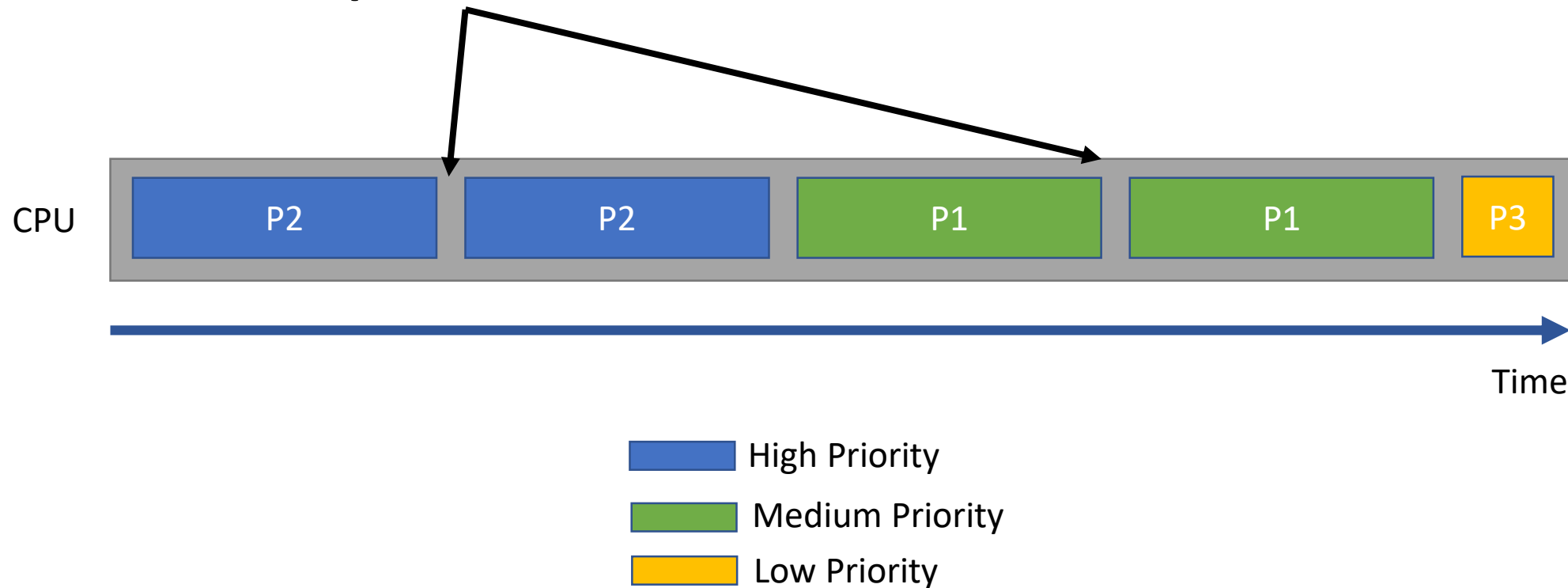
Priority scheduling of processes

- Let all the higher priority processes finish before moving to lower priority ones
- What is the **problem here?**



Priority scheduling of processes

- Let all the higher priority processes finish before moving to lower priority ones
- What is the **problem here?**



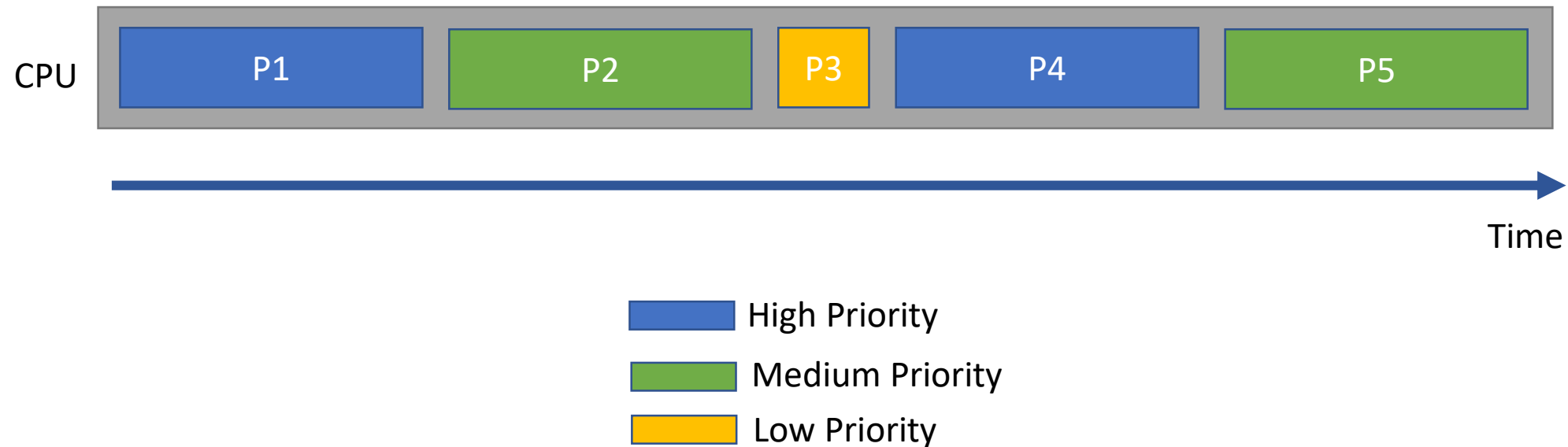
Priority scheduling of processes

- Even better: Don't **yield** if the current process is the **only one** of its priority
- This is the **bonus** part of your lab



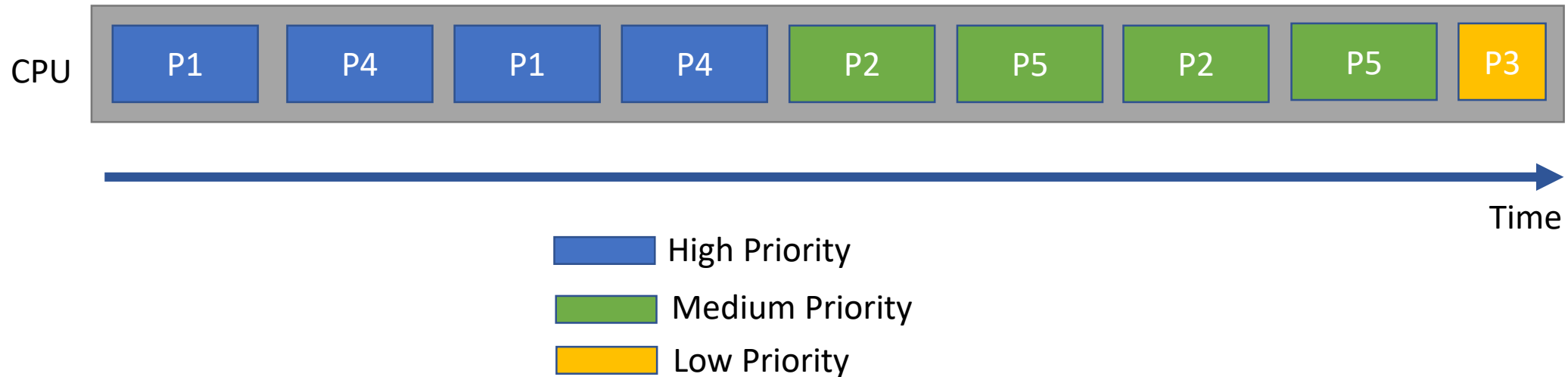
Processes with same priorities

- What if different processes have the same priorities?



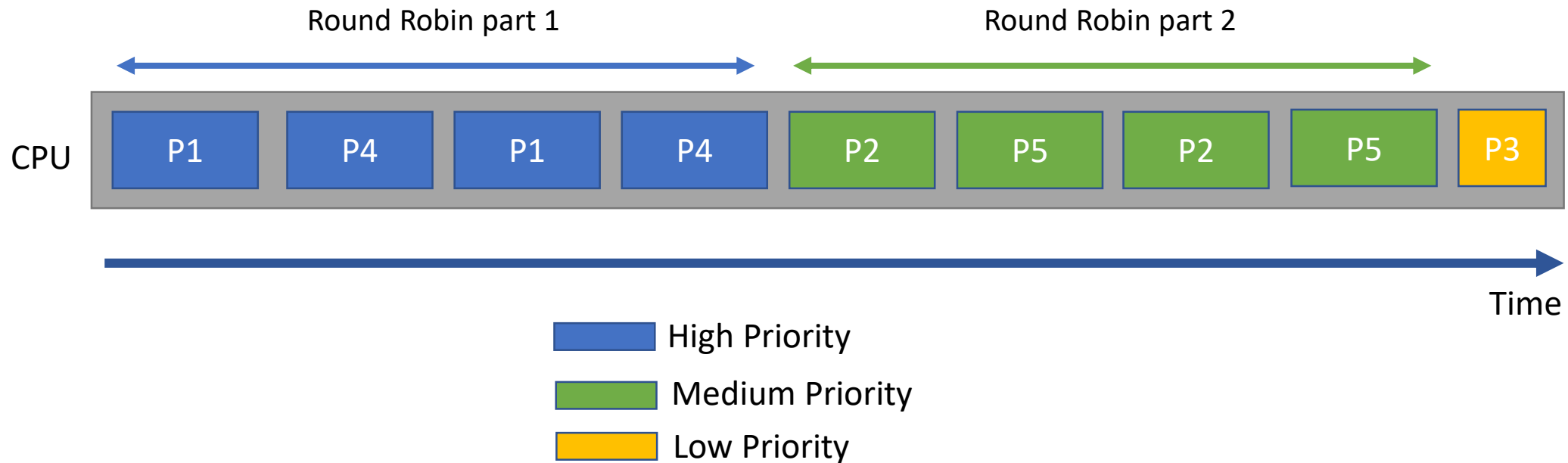
Processes with same priorities

- What if **different processes** have the **same** priorities?



Processes with same priorities

- Group processes with the same priorities together!
 - Use round robin!



Lab 3 – part 1: 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.



CS 1550

Week 8 – Lab 3

Teaching Assistant

Henrique Potter