CS 1550

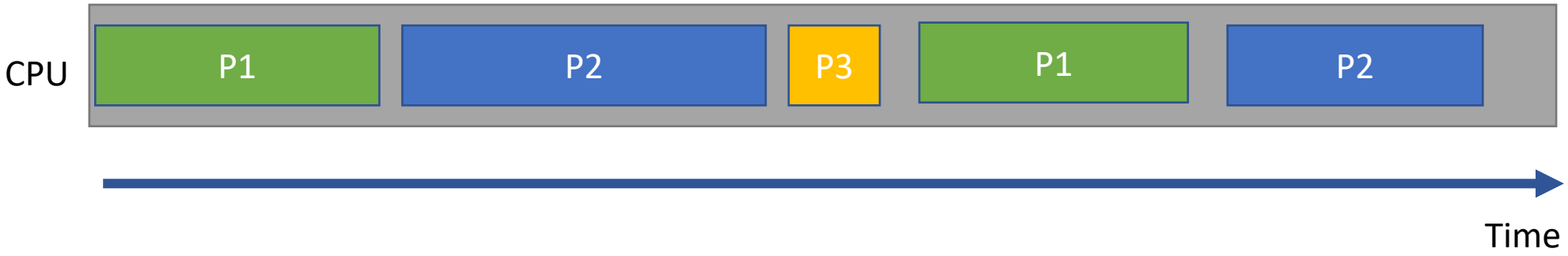
Week 7 – Priority Scheduling with xv6

Teaching Assistant

Maher Khan

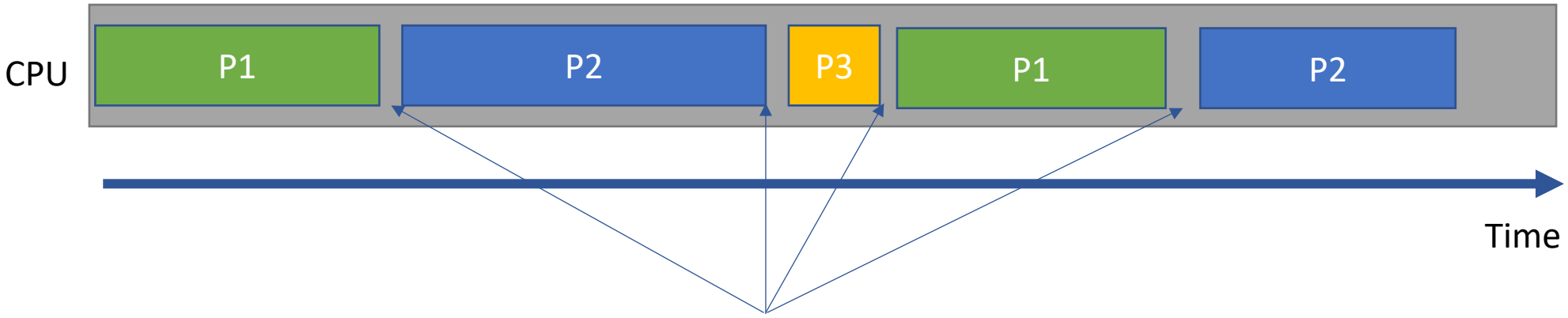
Scheduling of processes

- Switch processes during their execution.
- Currently, processes run in Round Robin inside your XV6



Scheduling of processes

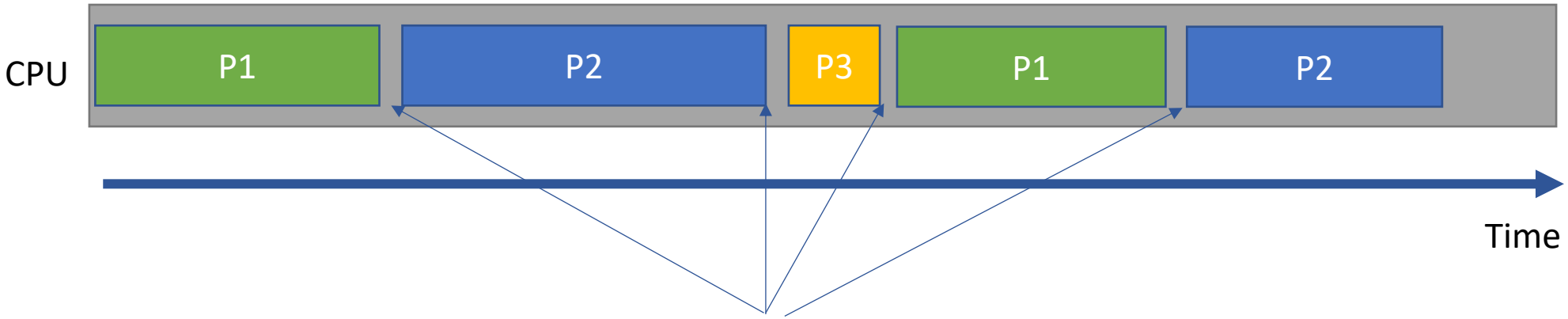
- Switch processes during their execution.
- Currently, processes run in Round Robin inside your XV6



What is the little gap?

Scheduling of processes

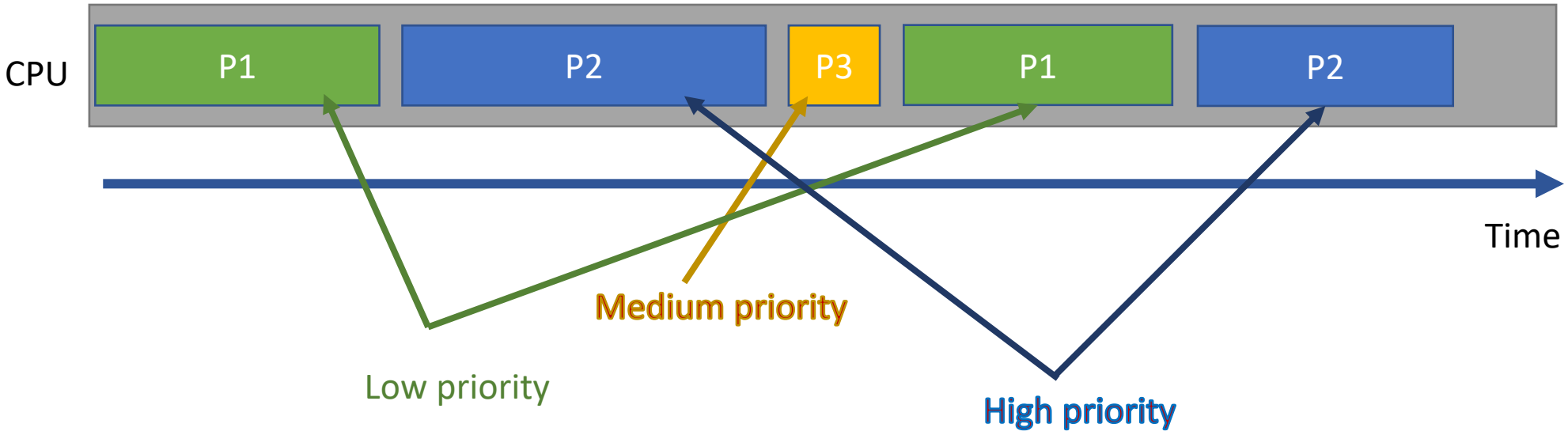
- Switch processes during their execution.
- Currently, processes run in Round Robin inside your XV6



What is the little gap?
The OS **Scheduler** is called when a timer interrupt occurs, and the kernel calls yield

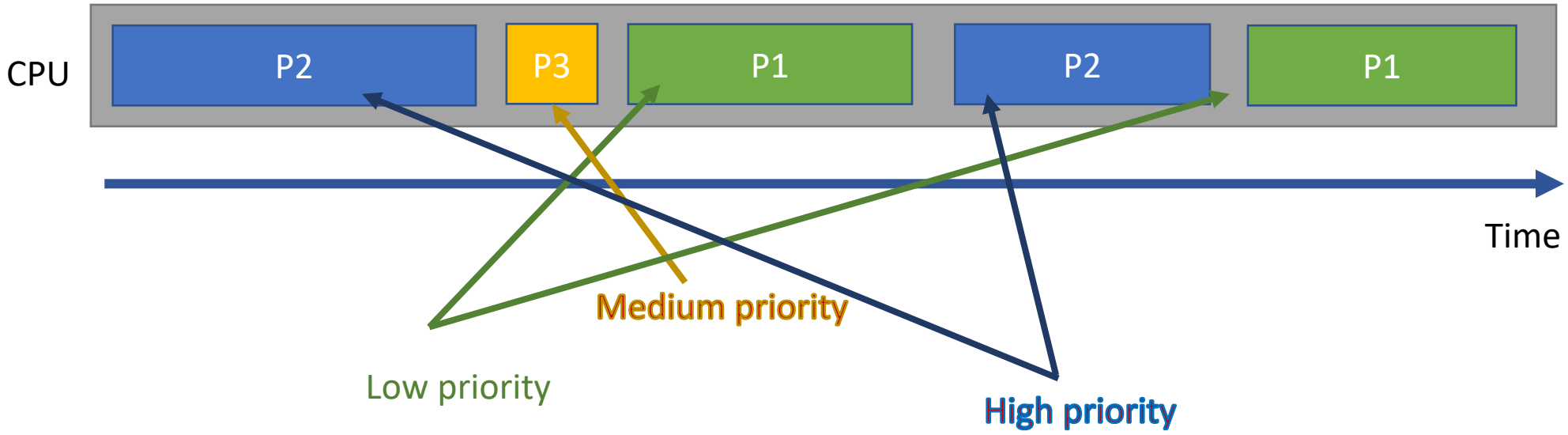
Priority scheduling of processes

- What if processes have different priorities?



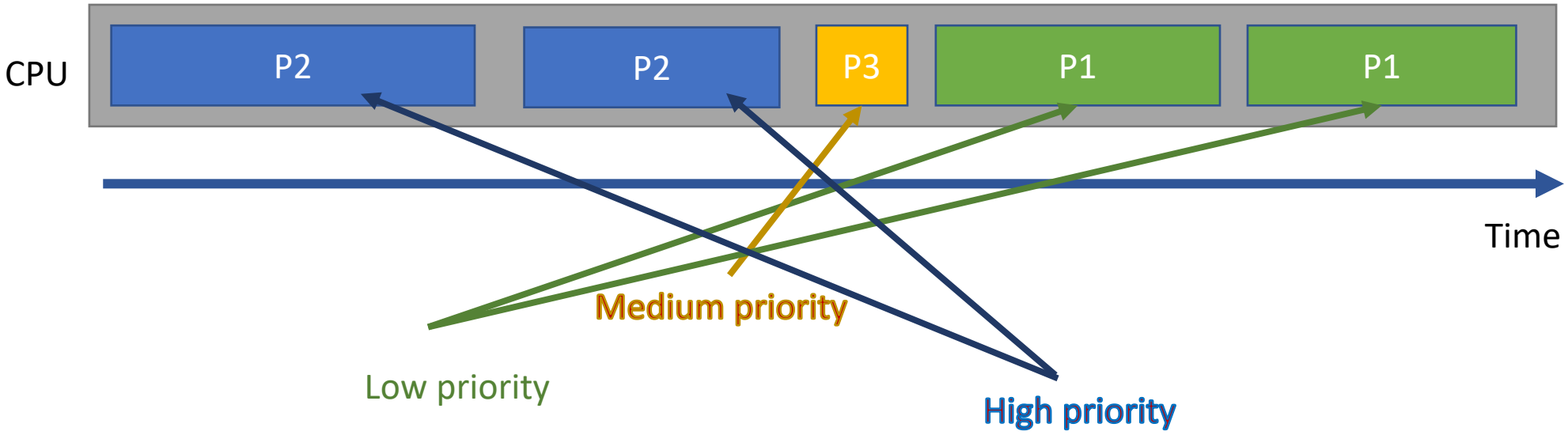
Priority scheduling of processes

- Is this a better arrangement of processes? Can we do better?



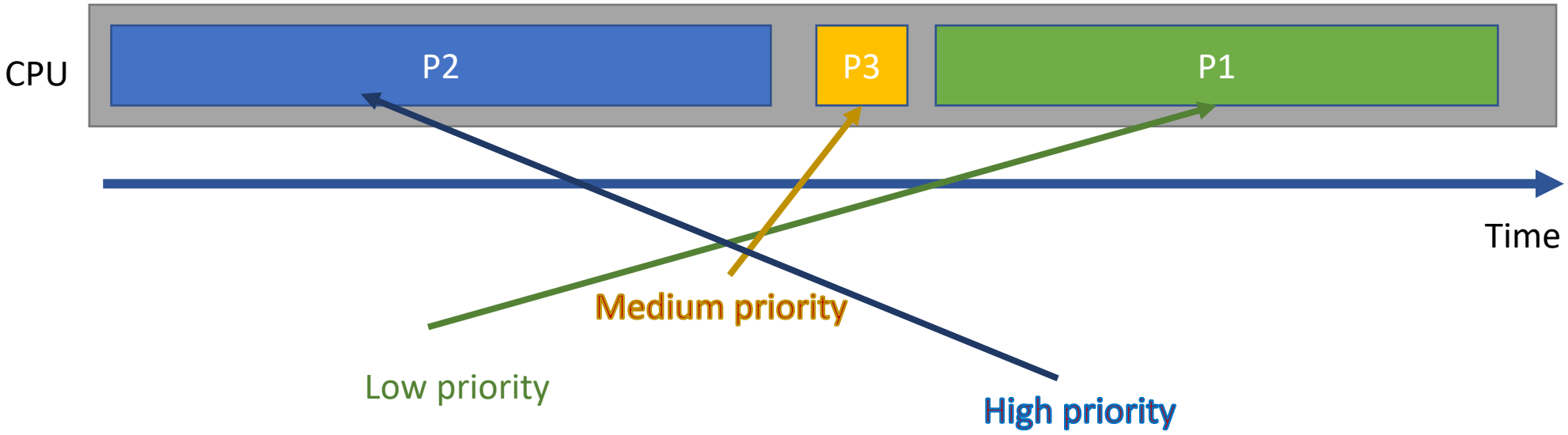
Priority scheduling of processes

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



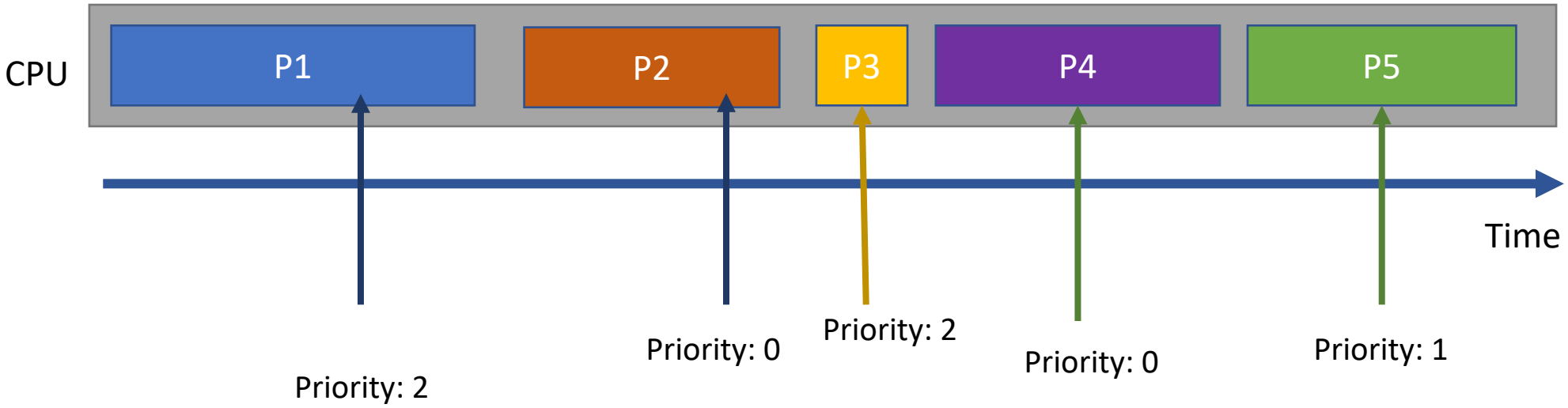
Priority scheduling of processes

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



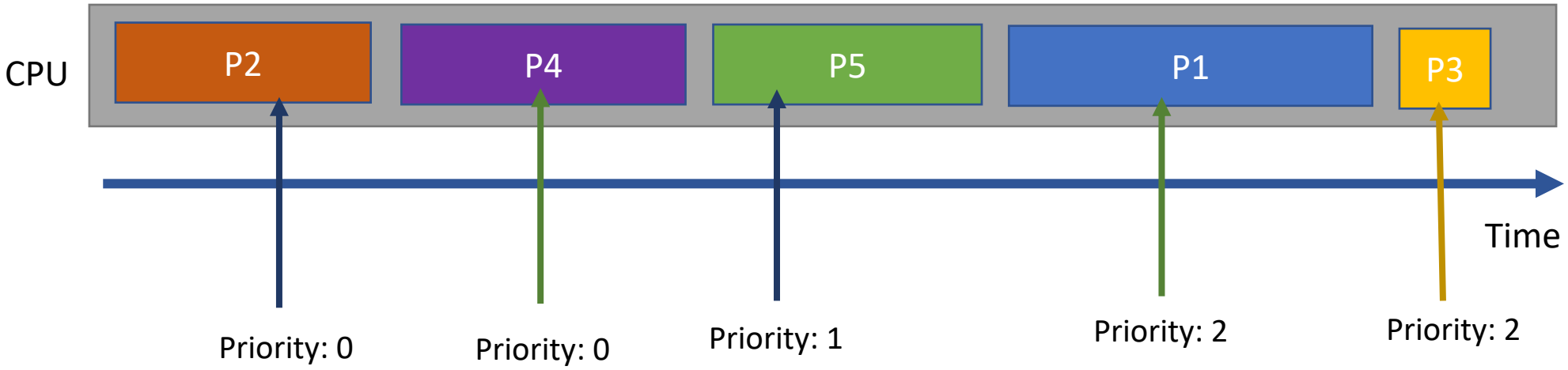
Processes with same priorities

- What if different processes have the same priorities?



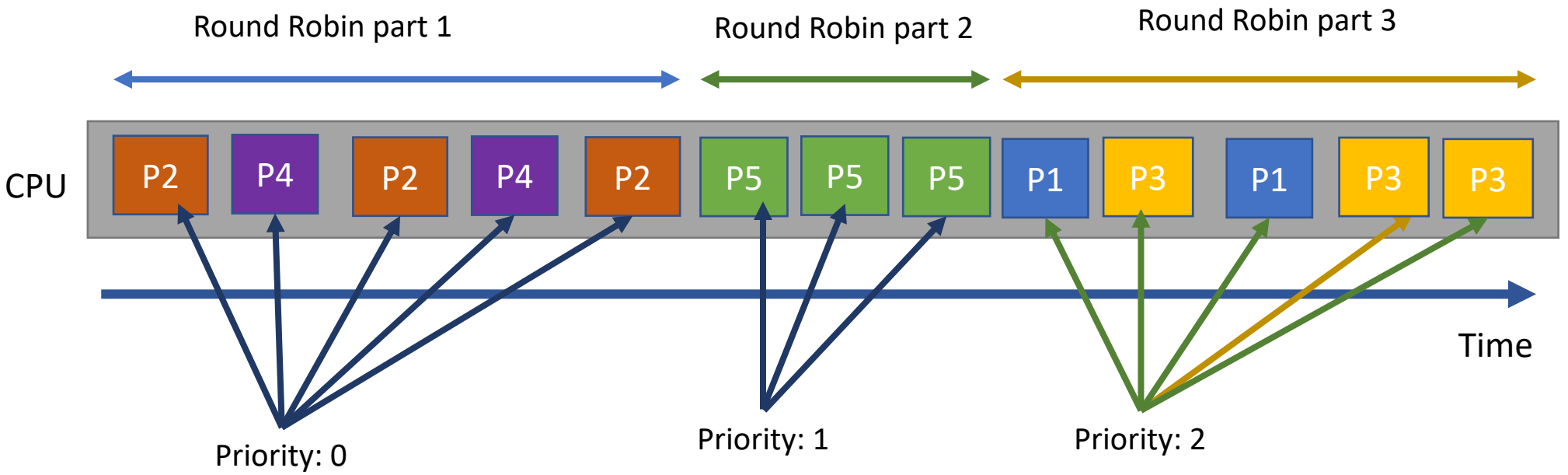
Processes with same priorities

- Group processes with the same priorities together!



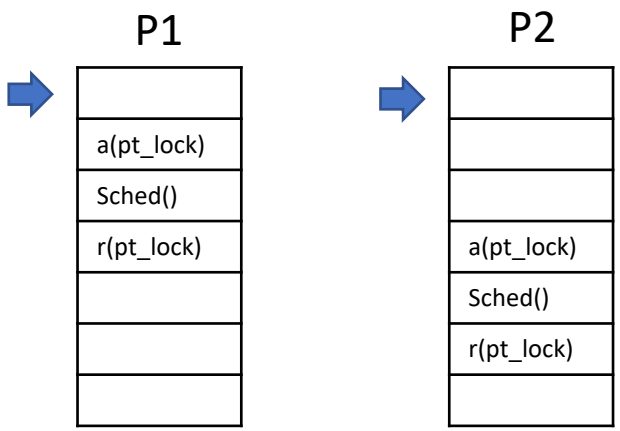
Processes with same priorities

- Processes with same priority should run in Round Robin!



The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

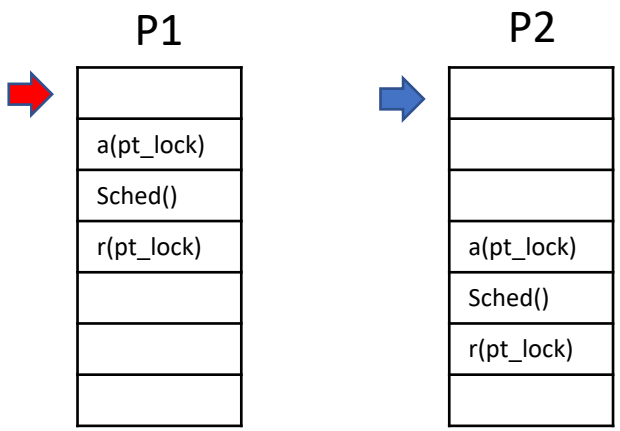


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

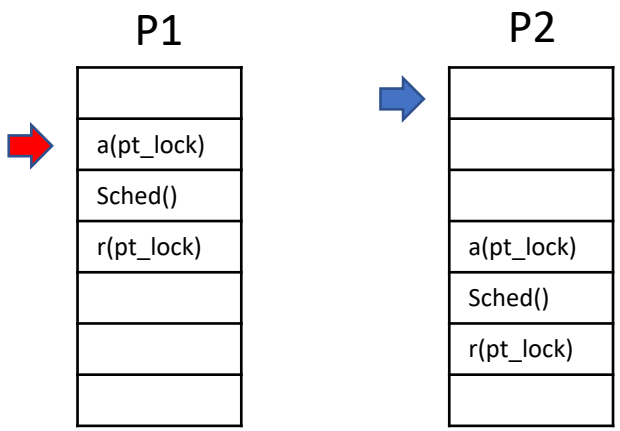


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

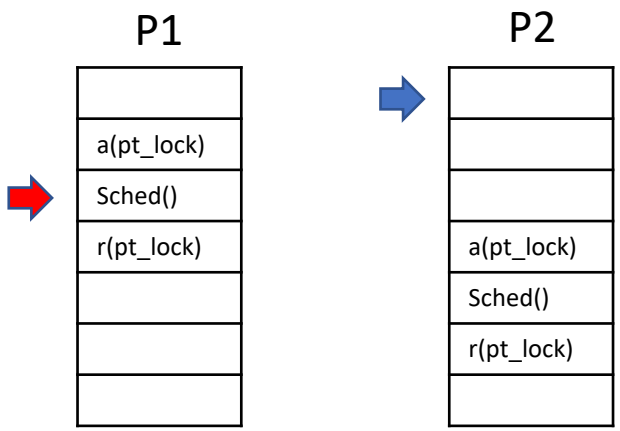


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

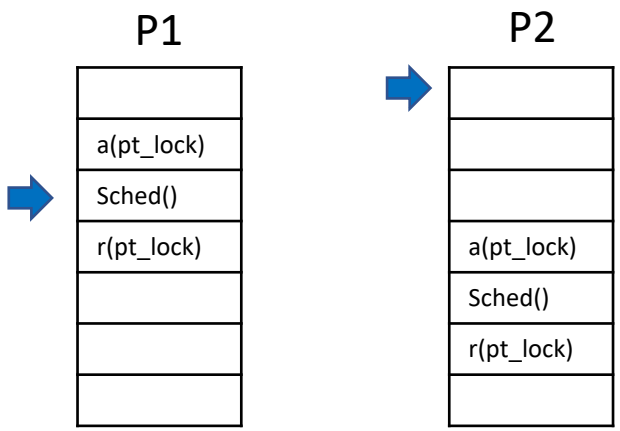


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

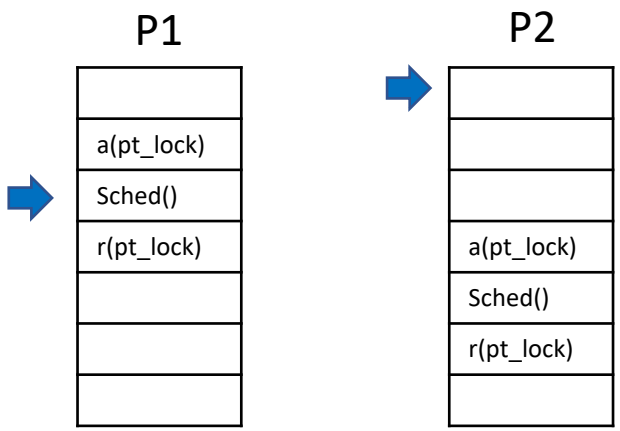


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

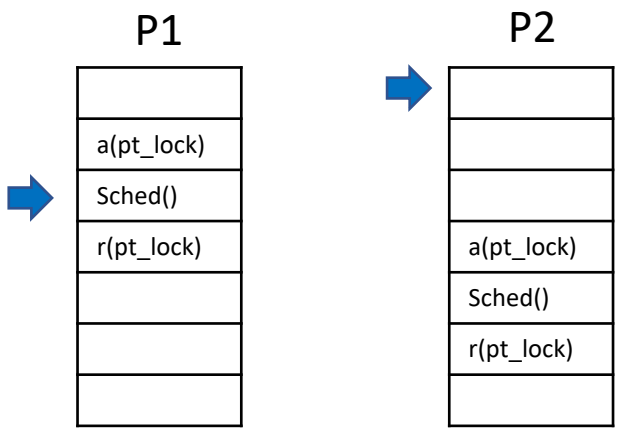


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

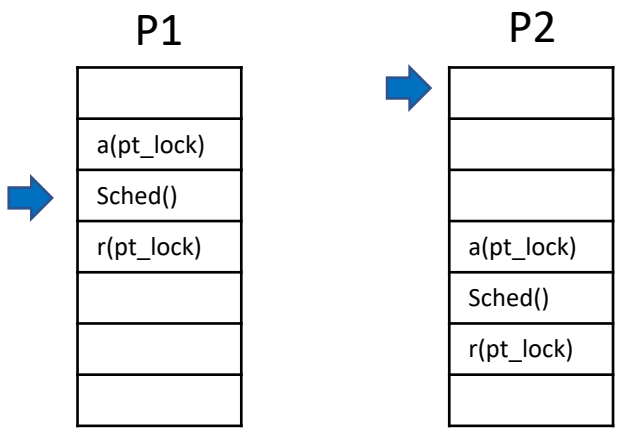


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

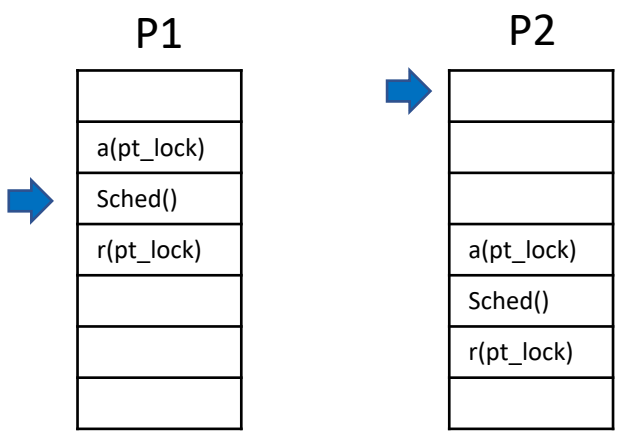


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

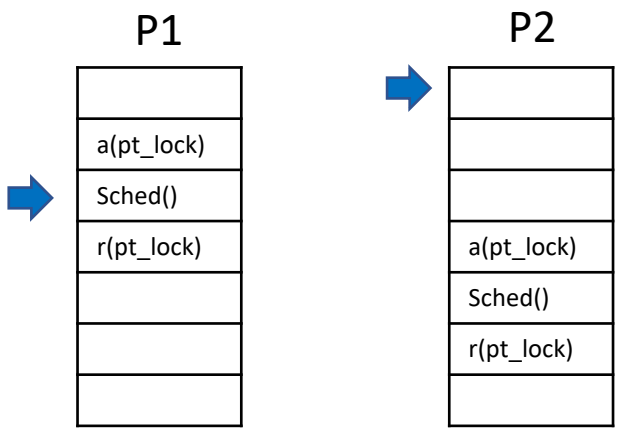


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

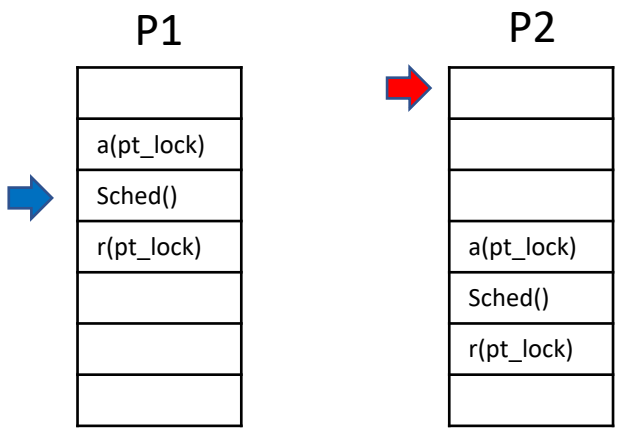


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

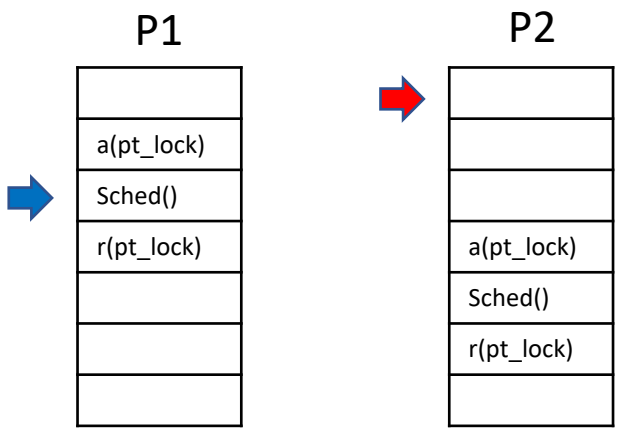


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

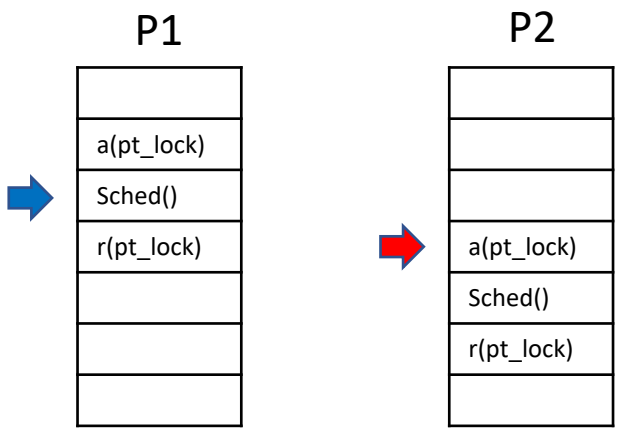


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

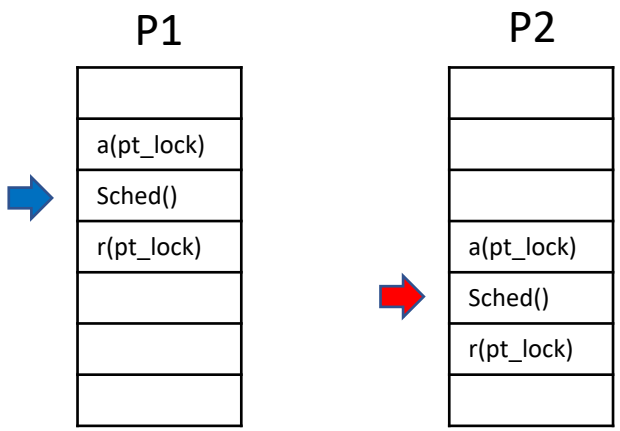


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

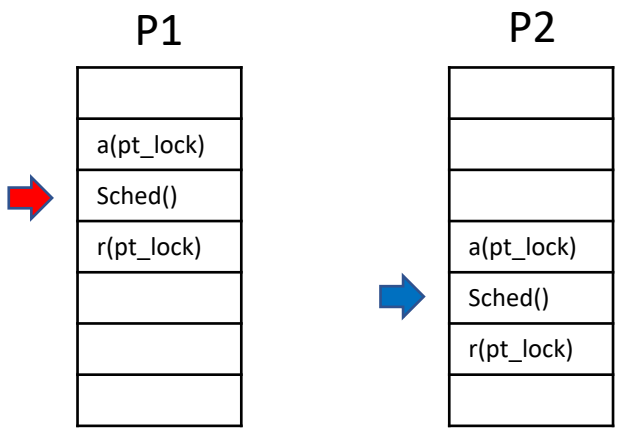


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch back to the scheduler.

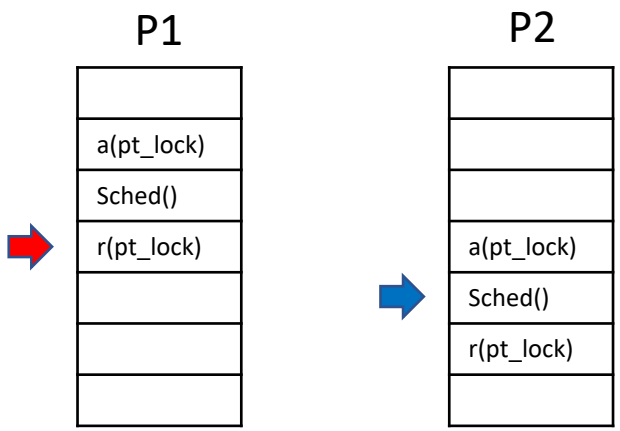


In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

The scheduler function

- Per-CPU process scheduler.
- Each CPU calls scheduler() after setting itself up.
- Scheduler never returns. It loops, doing:
  - choose a process to run
  - call swtch() to start running that process
  - eventually that process transfers control
  - via swtch() back to the scheduler.



In proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;          Frans Kaashoek (2 years ago) · Eliminate code

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

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

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

  }
}
```

Yield in trap

- Force process to give up CPU on clock tick.
- IRQ stands for Interrupt Requests
- The timer interrupt occurs after the current process's quantum is over
- The kernel takes over and calls yield() in the trap.c
- Yield() acquires ptable lock and then calls sched()

In trap.c:

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

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);
}
```

Lab 3 is out!

- Due: 11:59 PM Friday (March 8th)

CS 1550

Week 7 – Priority Scheduling with xv6

Teaching Assistant

Maher Khan