CS 1550

Week 6

Lab 3 and Project 2

Teaching Assistant

Henrique Potter

CS 1550 – Lab 3

- Modify xv6 scheduler from round robin to a priority based.

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

xv6

Scheduling of processes

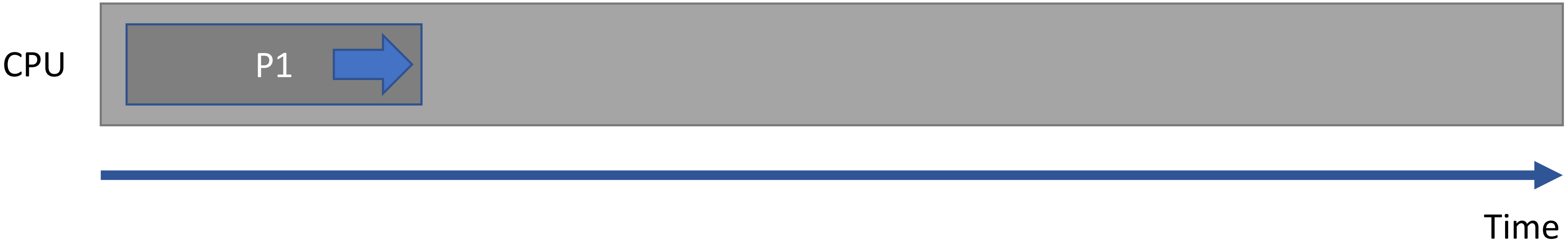
- Important feature of OS’s is allowing concurrent execution of processes
- Better utilization of resources
  - While a process waits for I/O another one can execute
- In **xv6**, processes are scheduled in a round-robin fashion

xv6

However, how does the scheduler work in xv6?

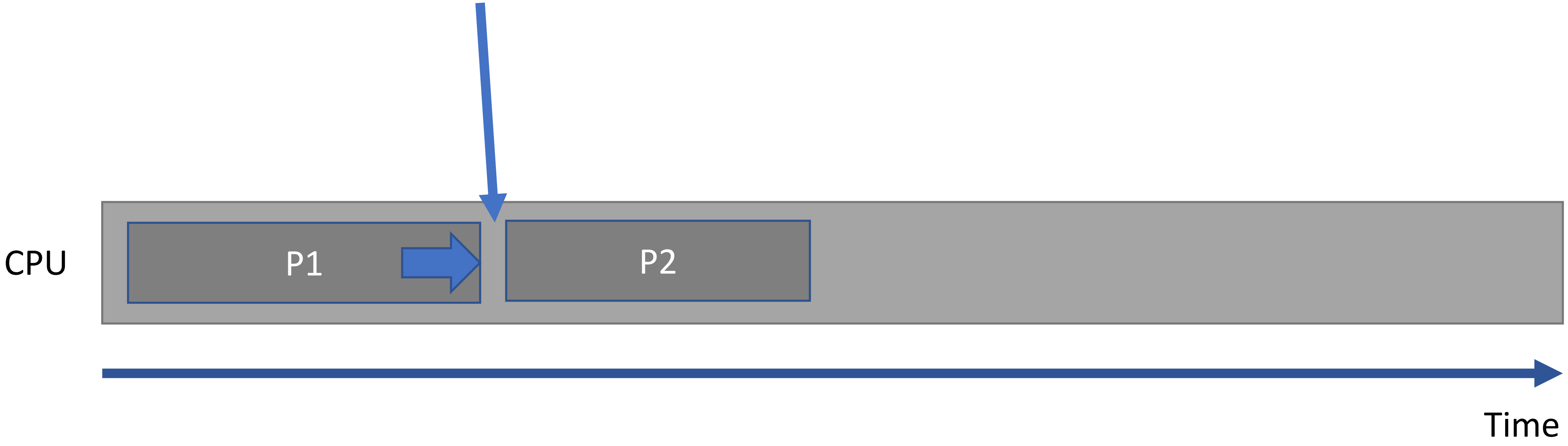
Scheduling of processes

- xv6 scheduler interrupts



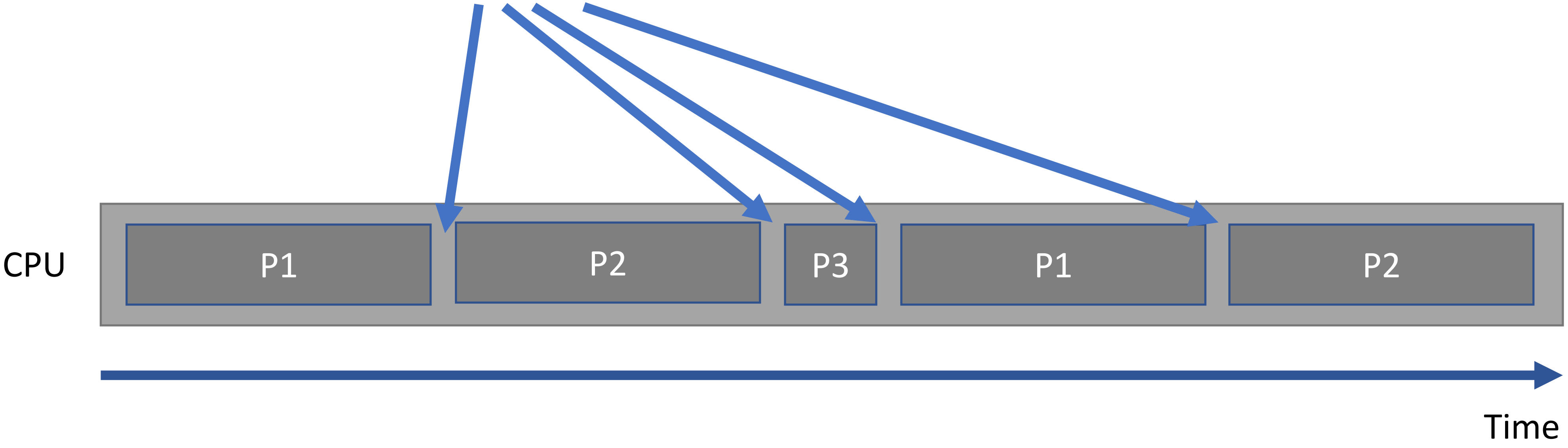
Scheduling of processes

- xv6 scheduler interrupts
- The scheduler is called, and a new process is selected



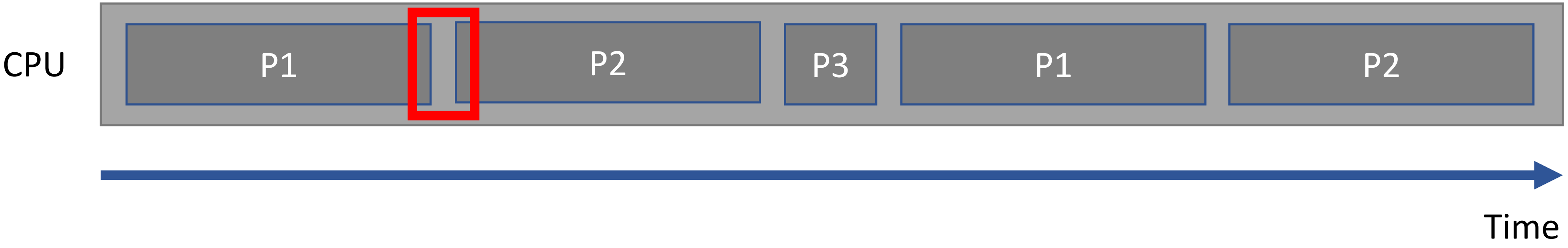
Scheduling of processes

- xv6 scheduler interrupts
- The scheduler is called, and a new process is selected



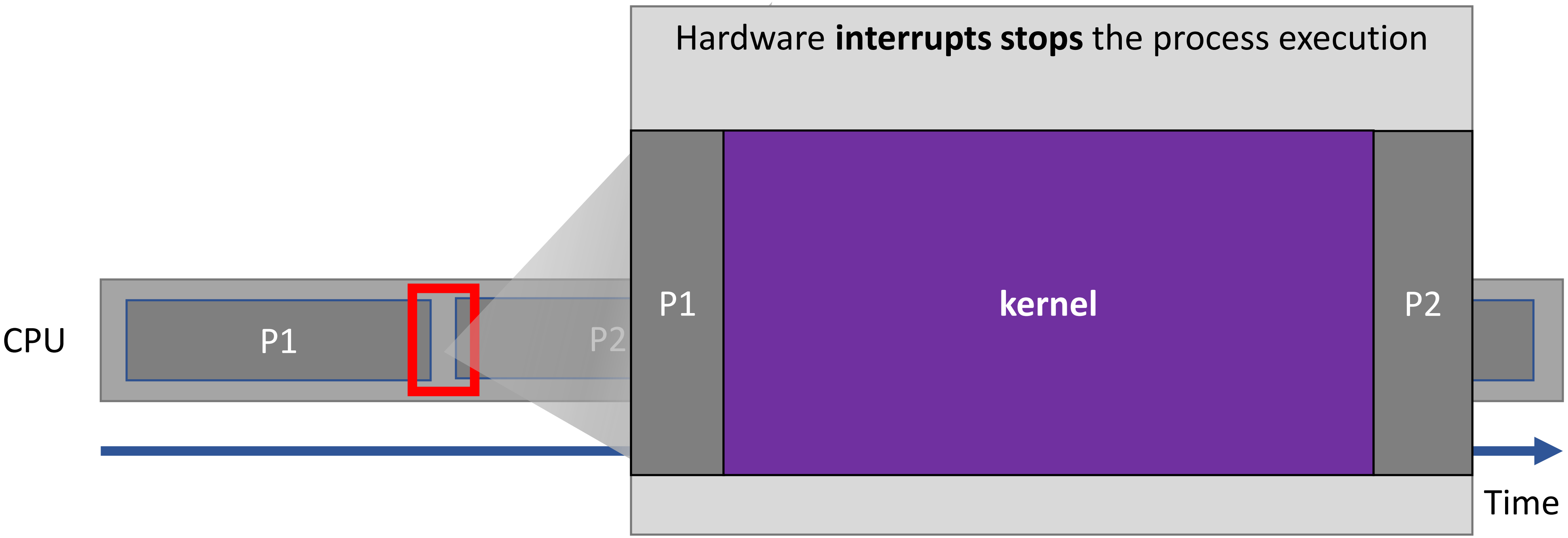
Scheduling of processes

- How processes are switched during their execution?



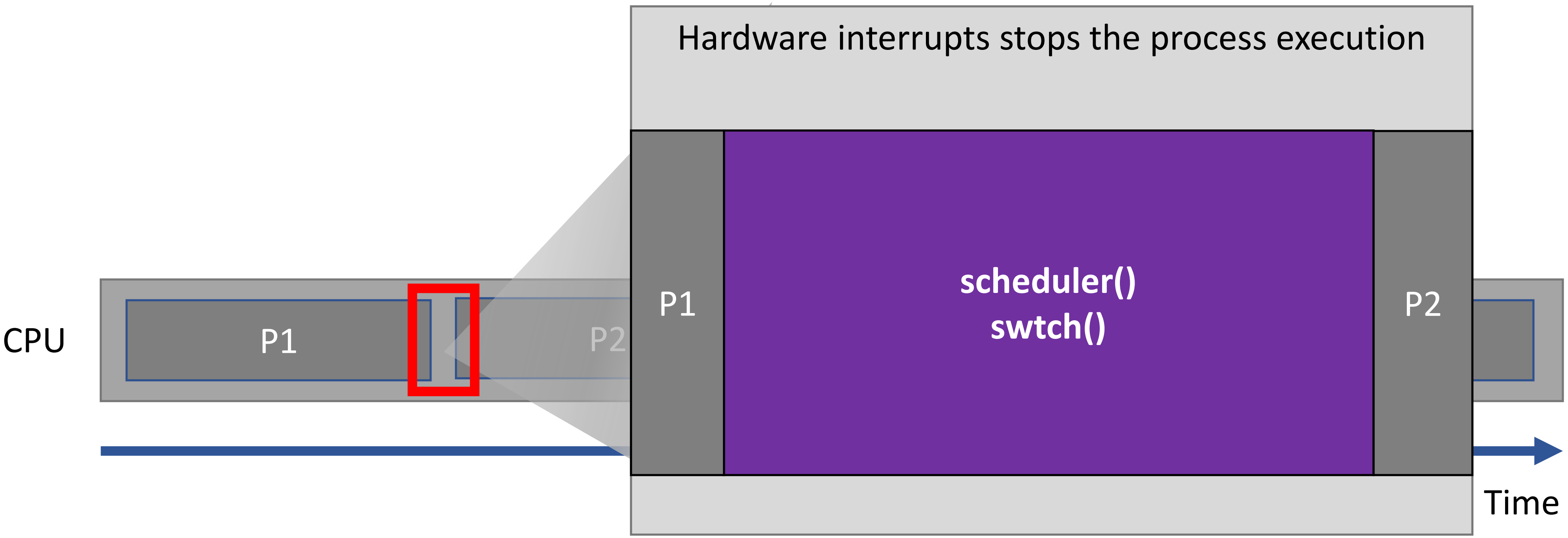
Scheduling of processes

- How processes are switched during their execution?



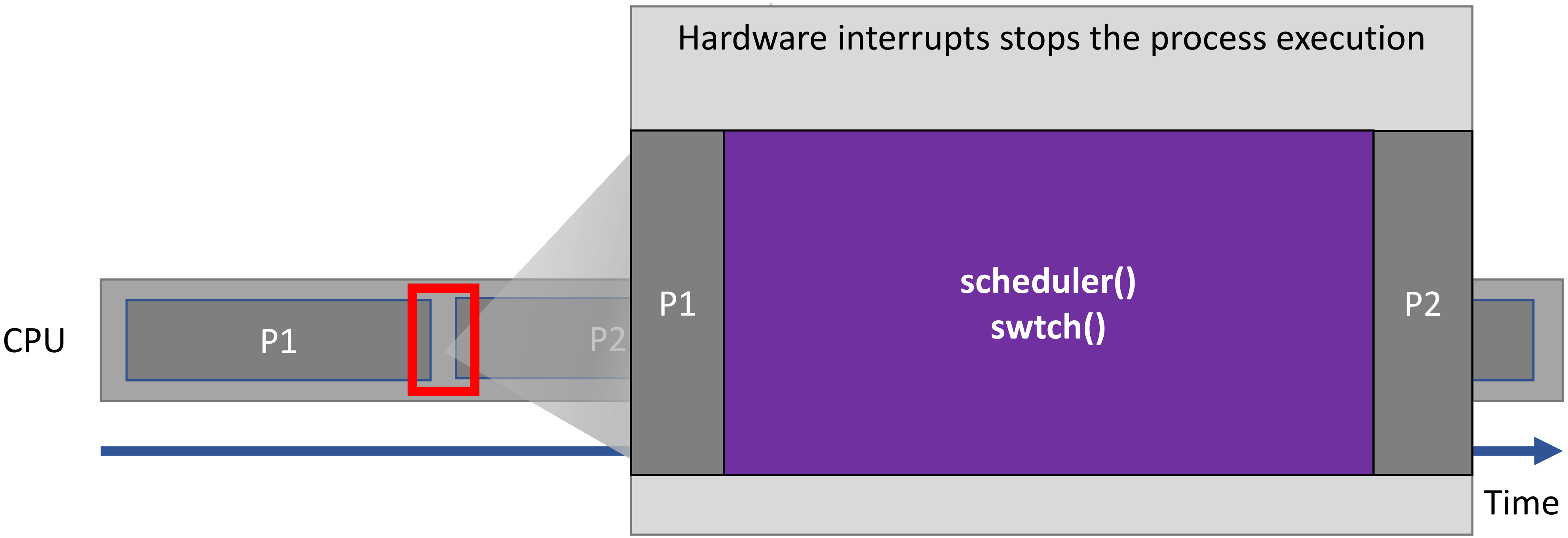
Scheduling of processes

- How processes are switched during their execution?



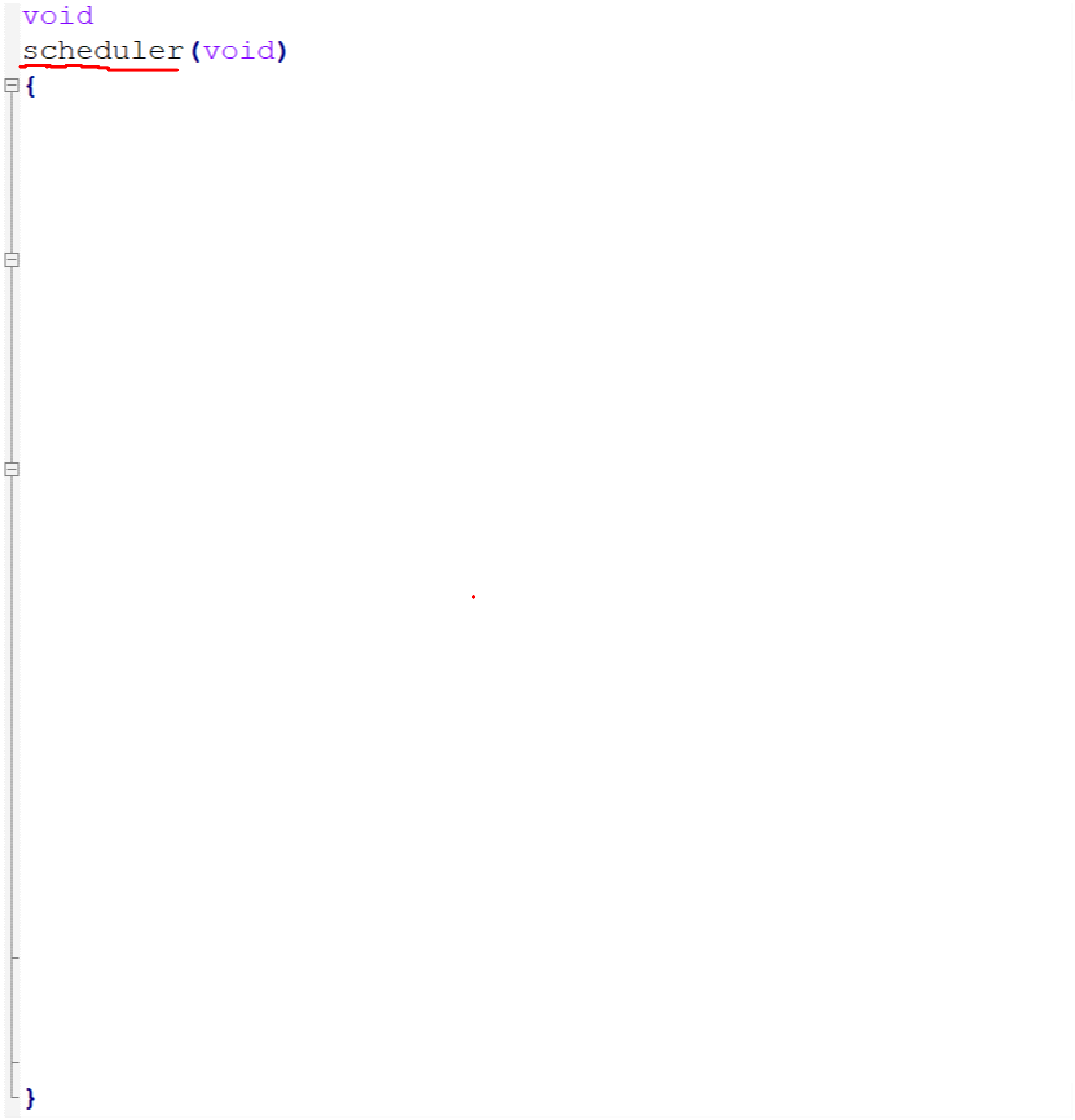
Scheduling of processes

- How processes are switched during their execution?



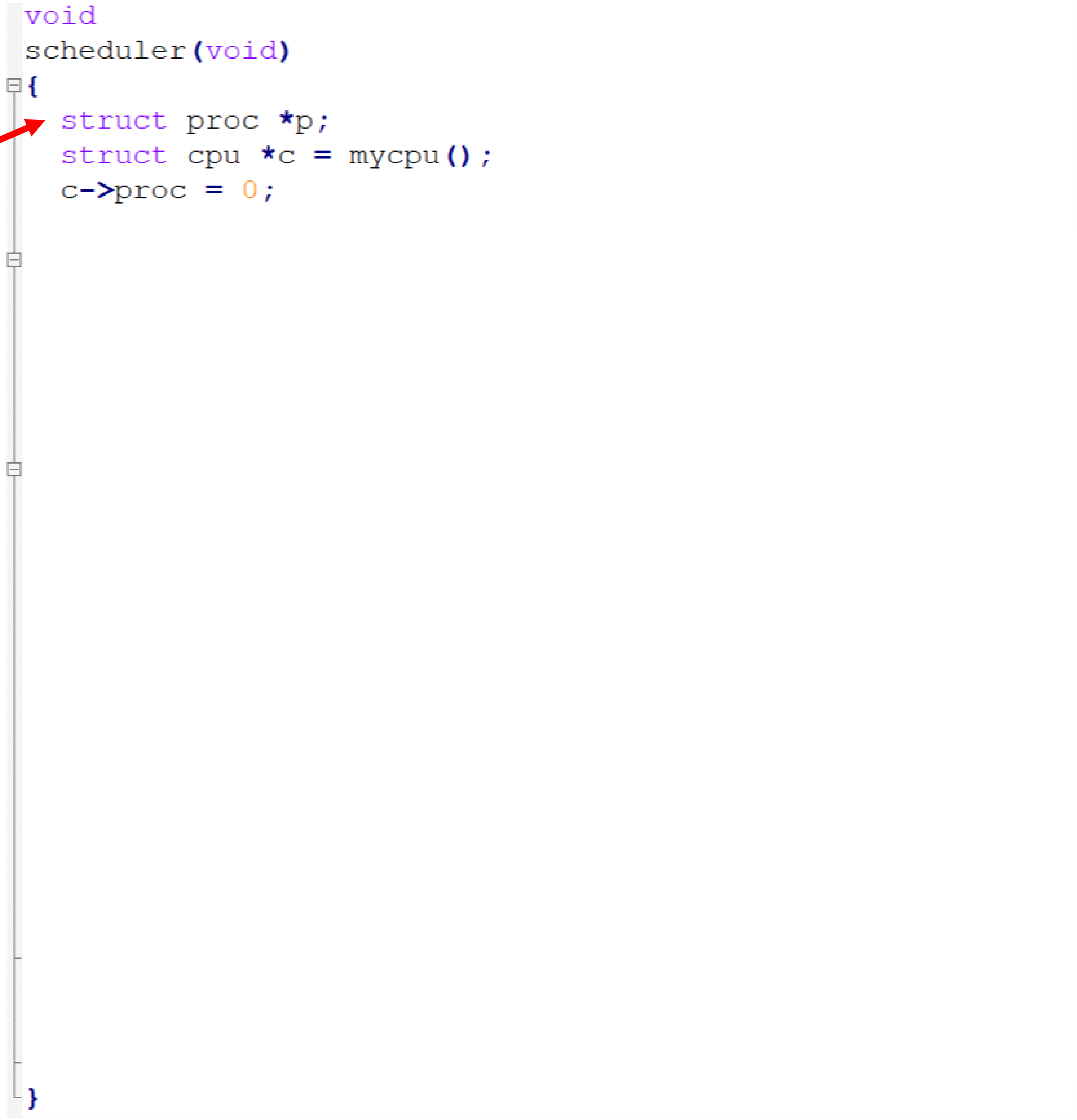
proc.c implements the scheduler function

- **proc.c** file



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

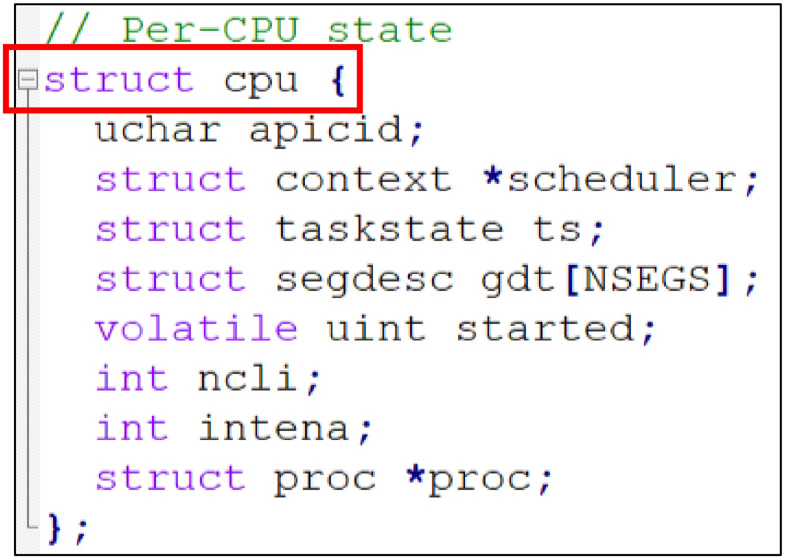
- **proc.c** file



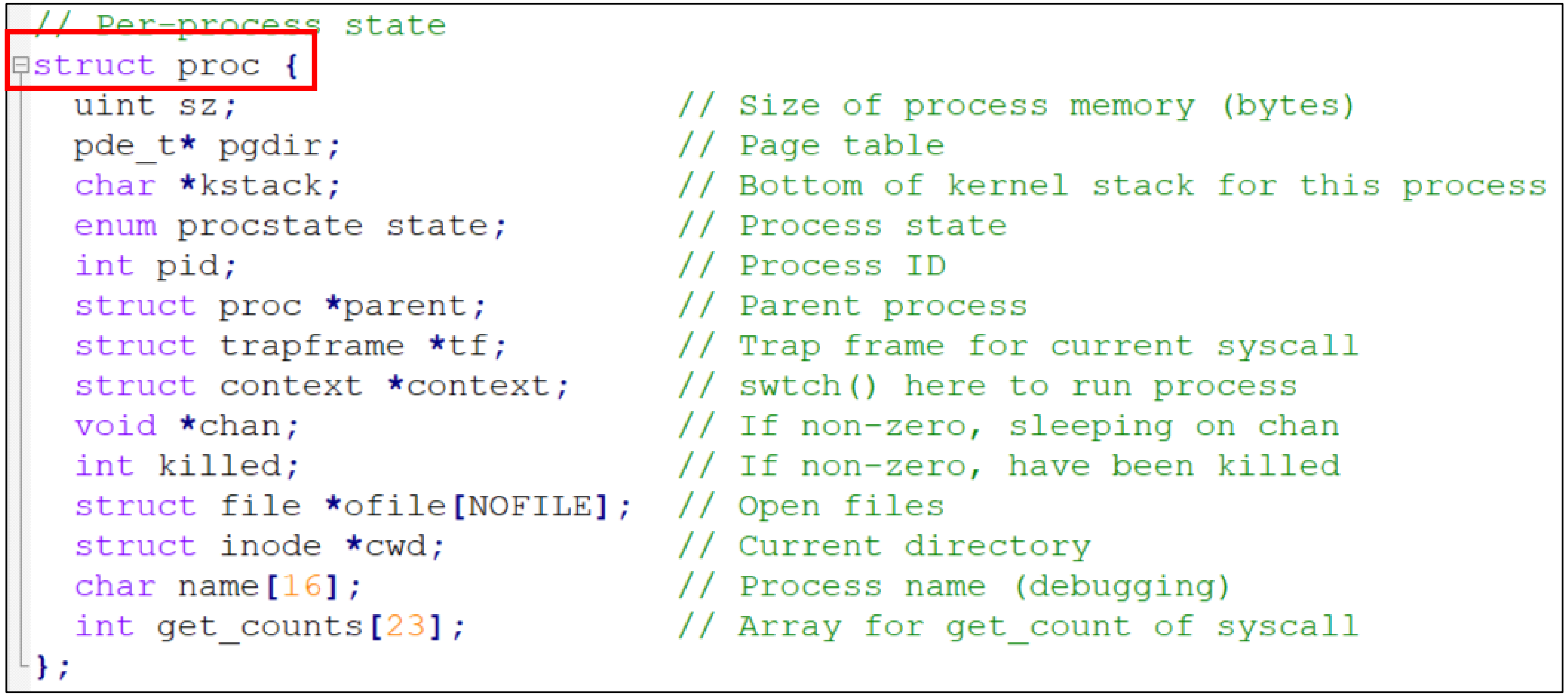
```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;








}
```

The process information

- **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;
};
```

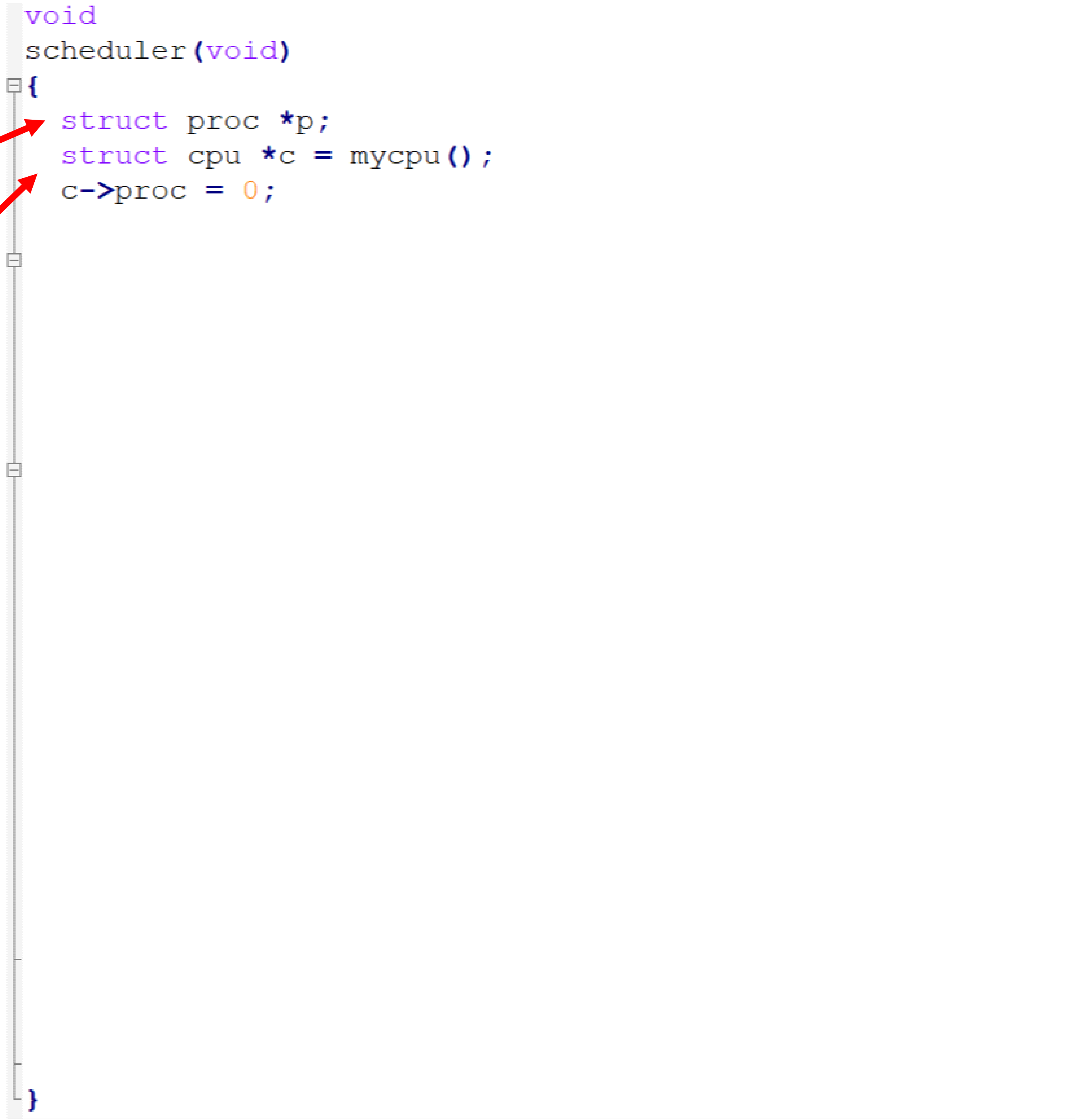


```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // 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
};
```

- **proc.c** file

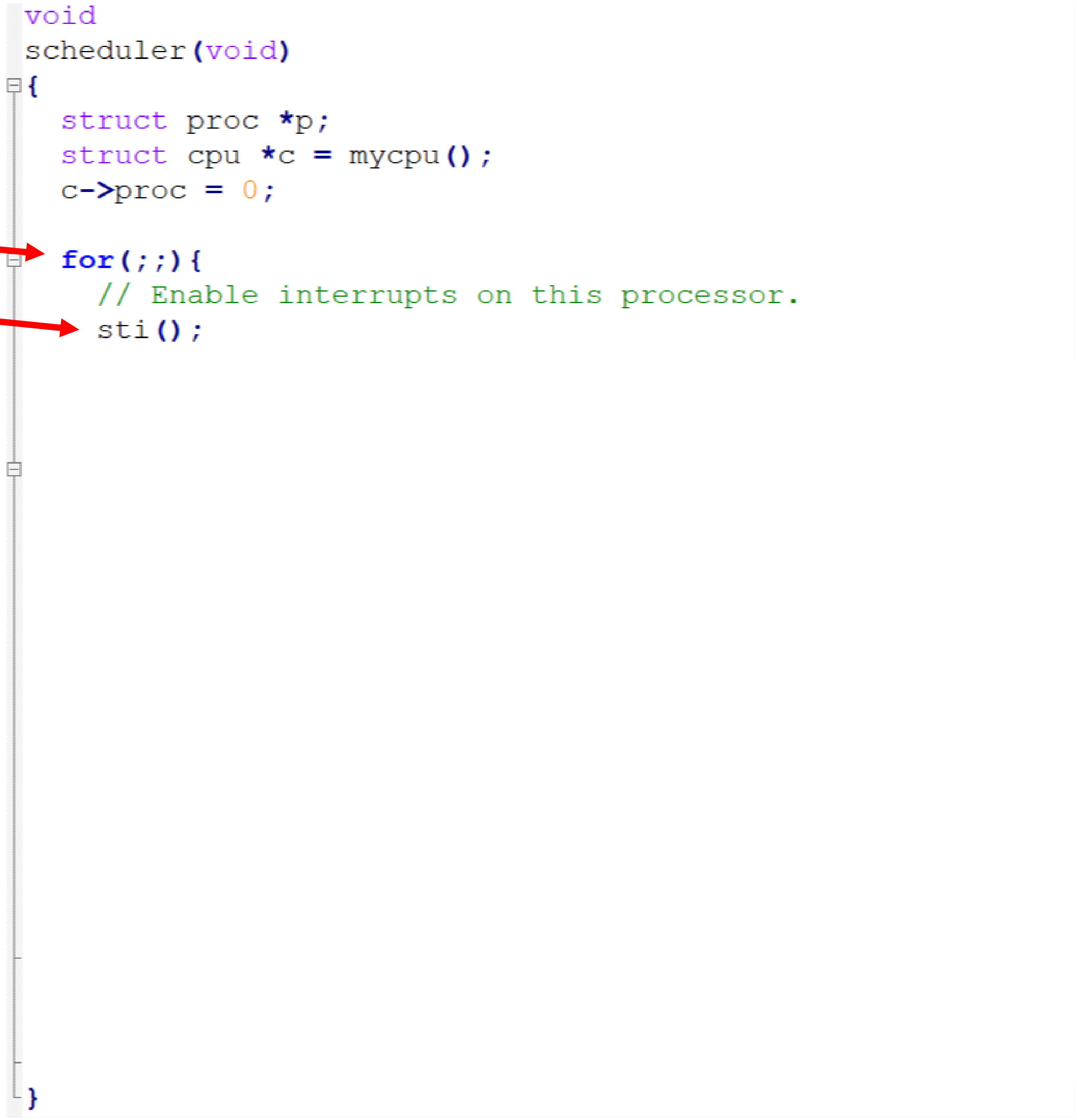
The process state information

The cpu state information



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









}
```

- **proc.c** file

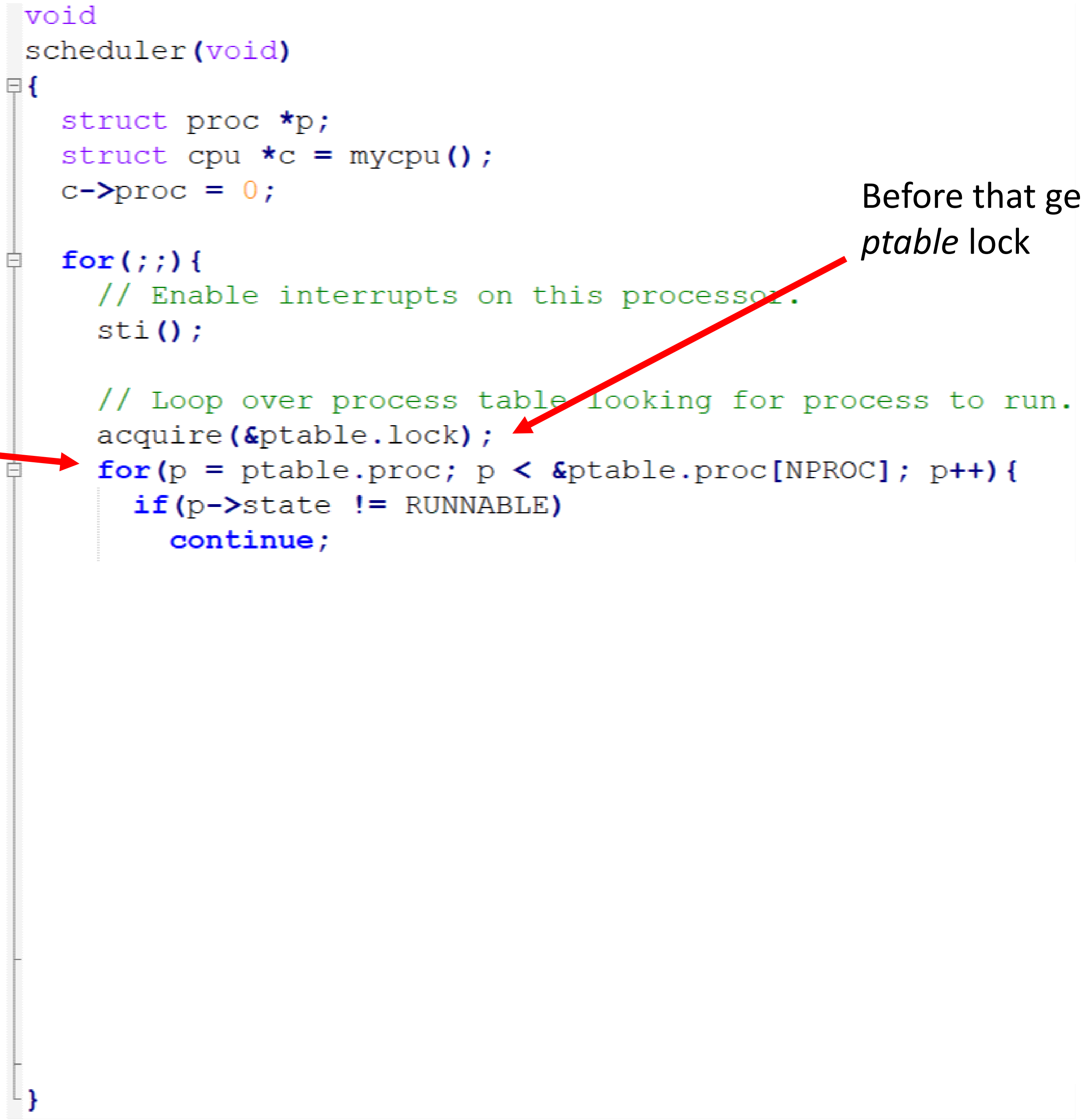


```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

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









}
```

Infinite loop

Enable interrupts

- **proc.c** file

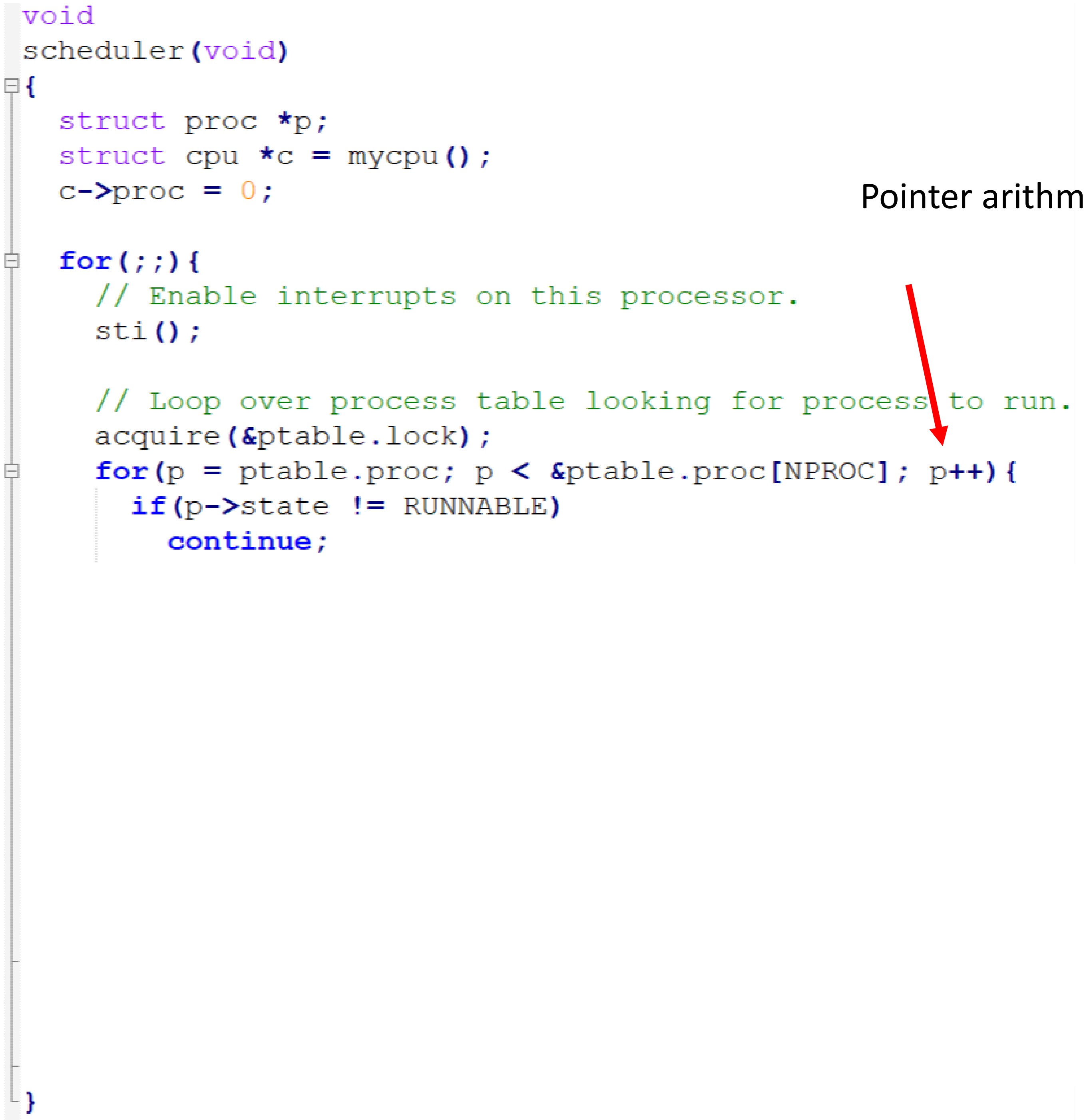


```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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

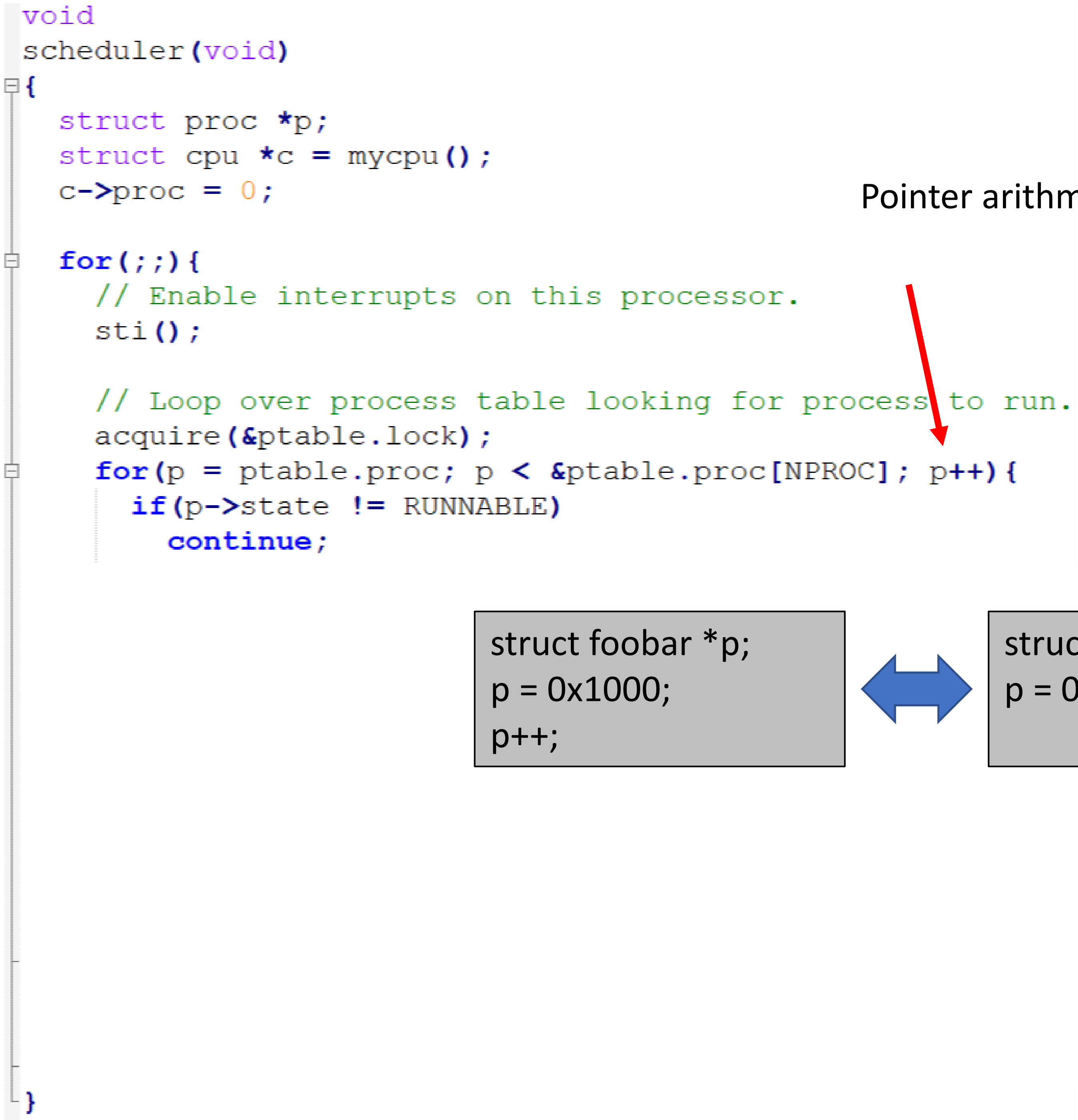
- **proc.c** file



```
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



```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;







}
```

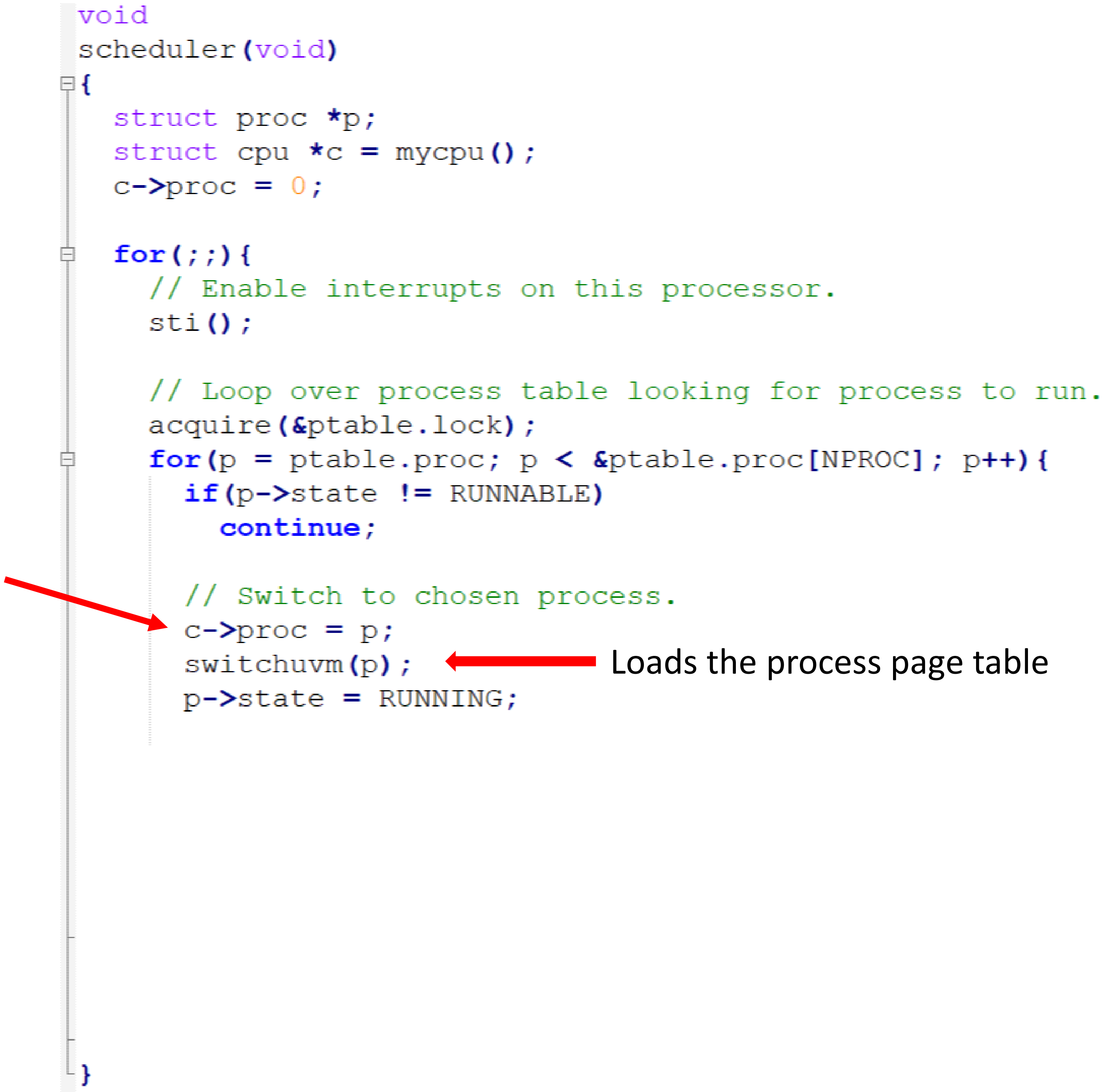
Pointer arithmetic!

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

⇔

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

- **proc.c** file



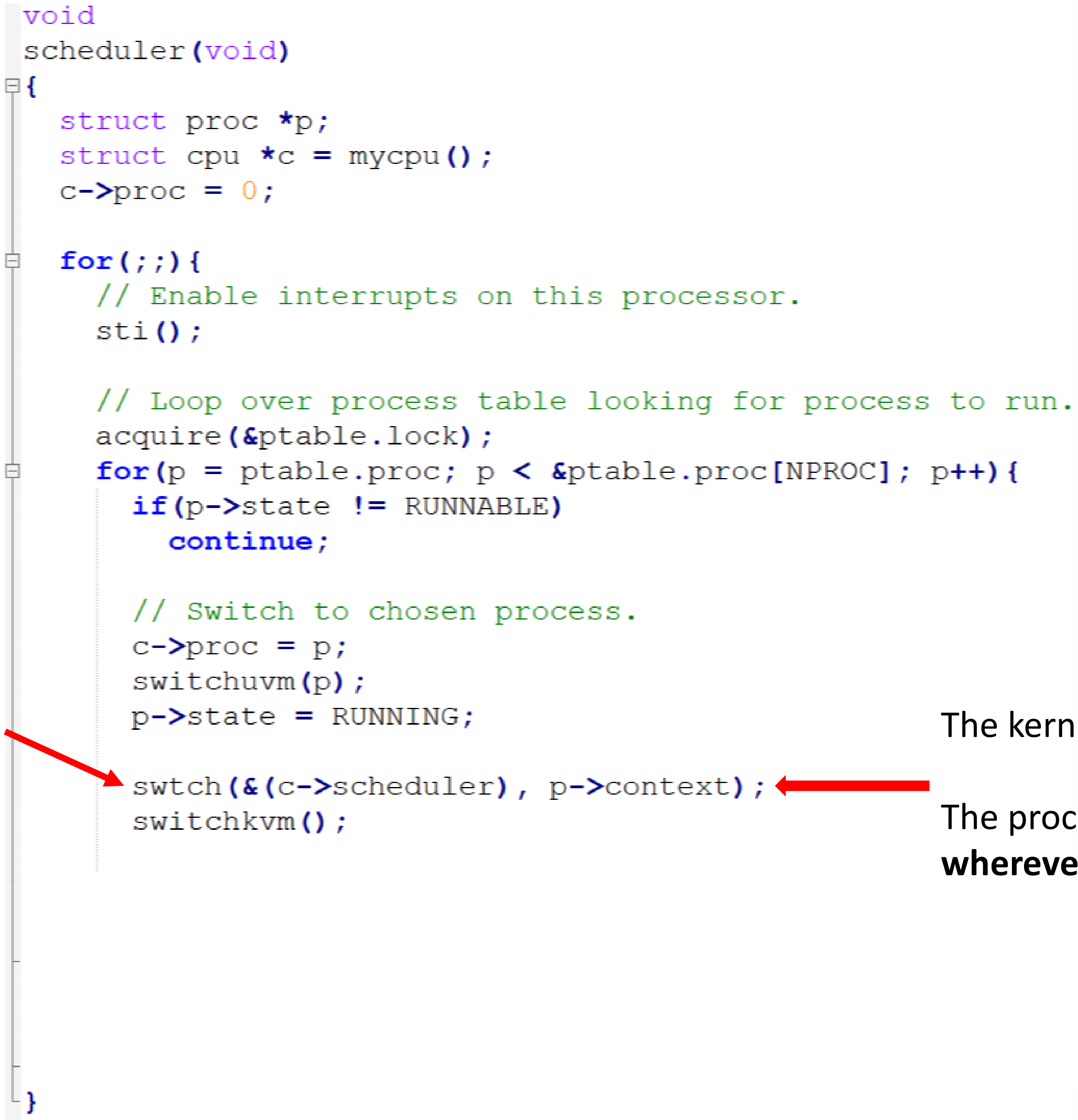
```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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;
      switchuvm(p);
      p->state = RUNNING;









}
```

cpu process is set

This is what myproc() returns

Loads the process page table

- **proc.c** file



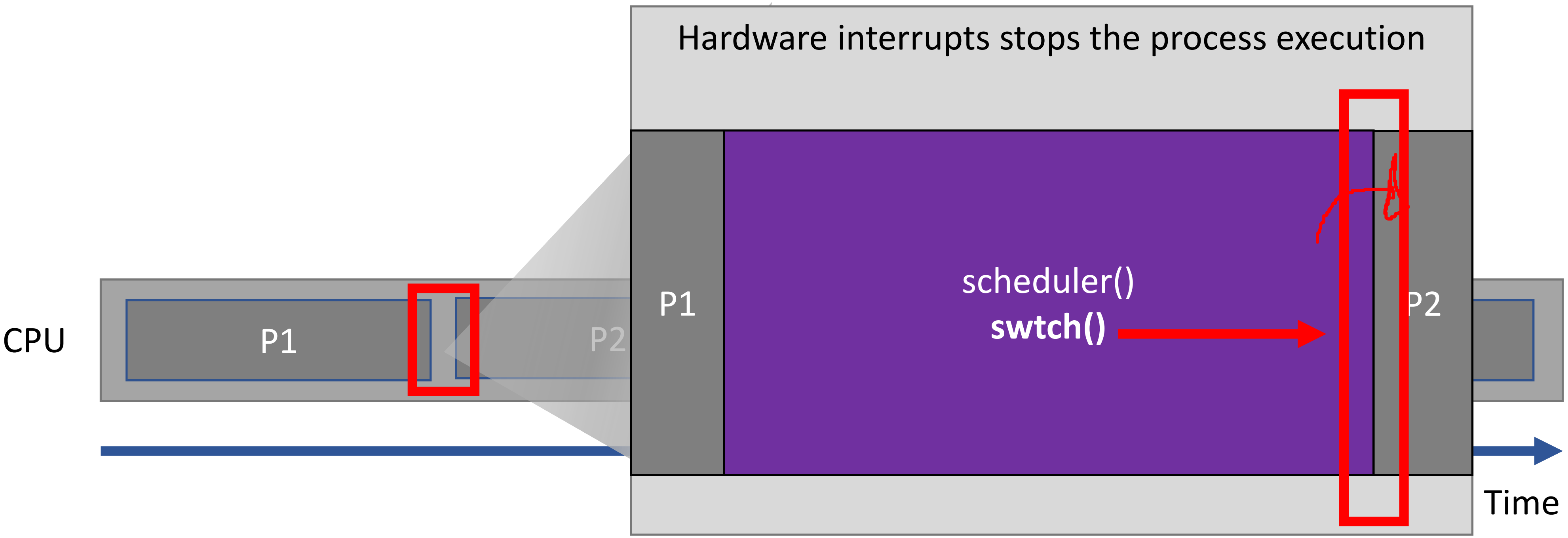
```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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;
      switchuvm(p);
      p->state = RUNNING;

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






}
```

Here the process is **switched** to execute

The kernel execution will **stop here**

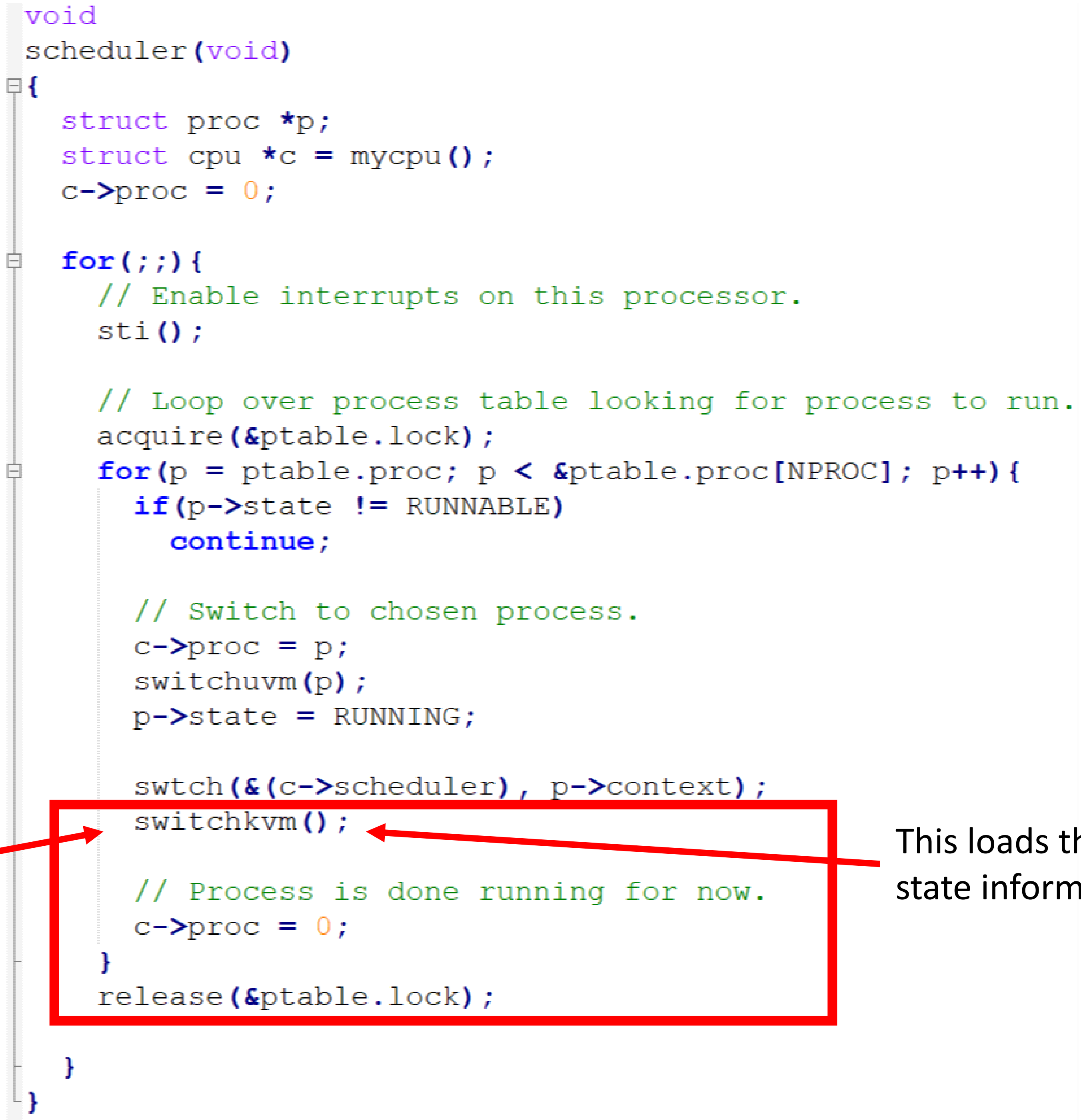
The process will **continue** from **wherever** is stopped

Scheduling of processes



proc.c implements the scheduler function

- **proc.c** file

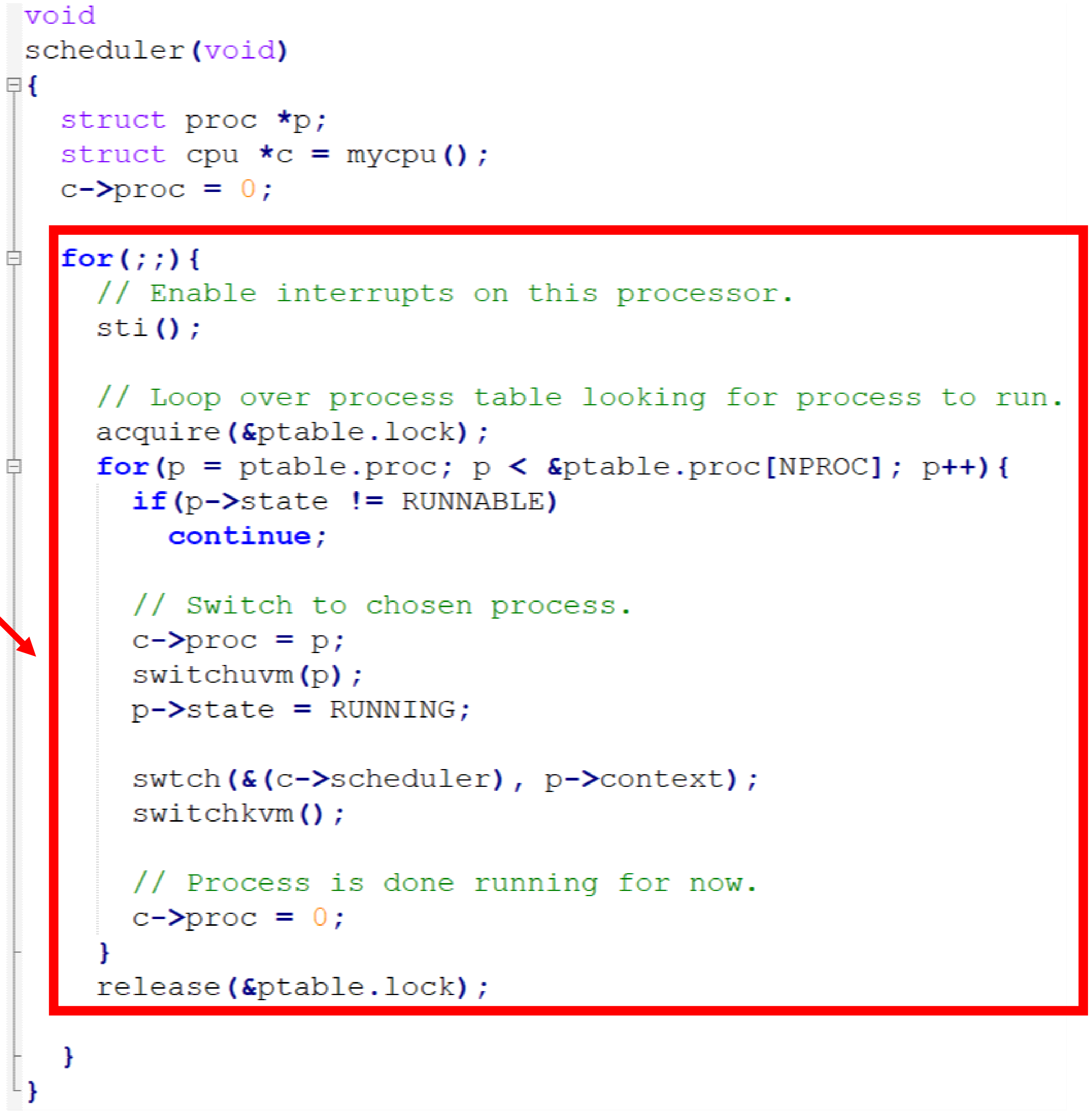


```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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;
      switchuvm(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

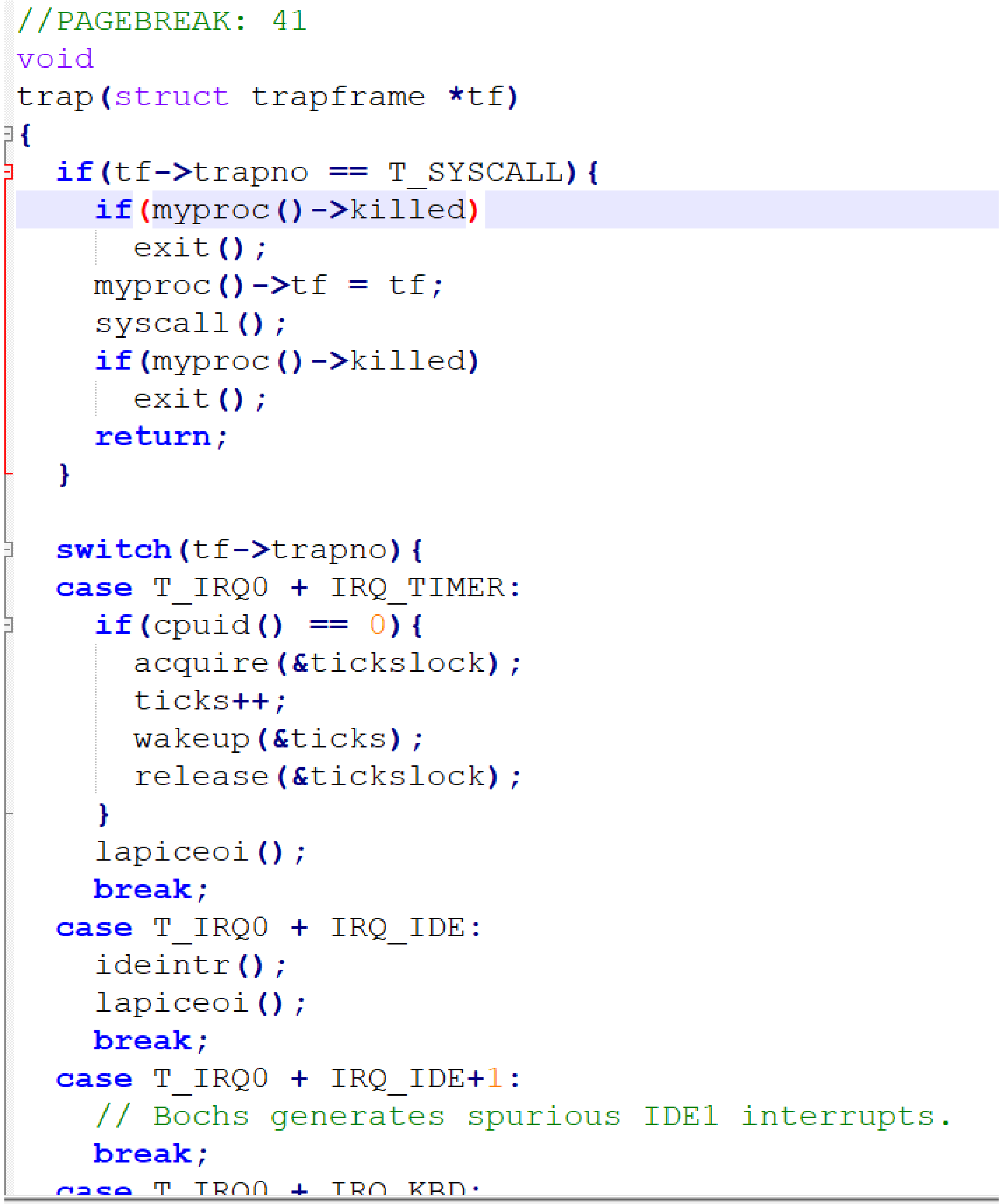


```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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);

  }
}
```

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

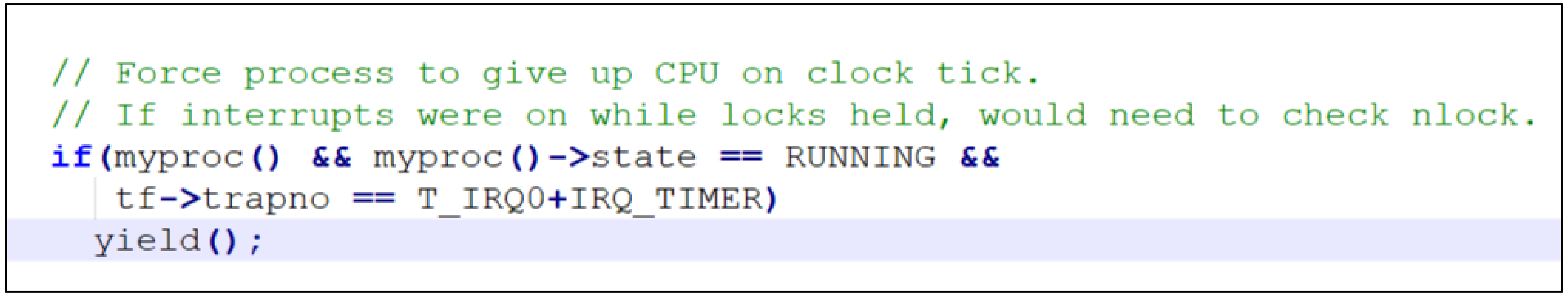


```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  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:

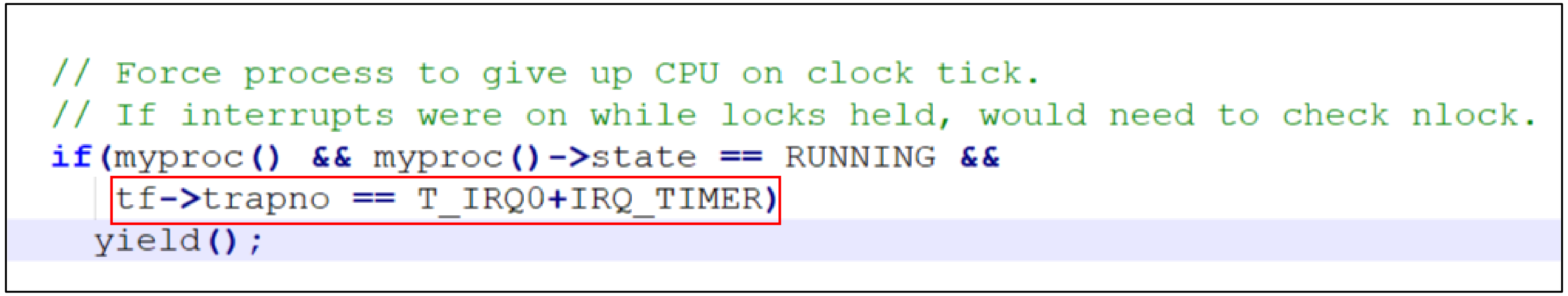


```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

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:

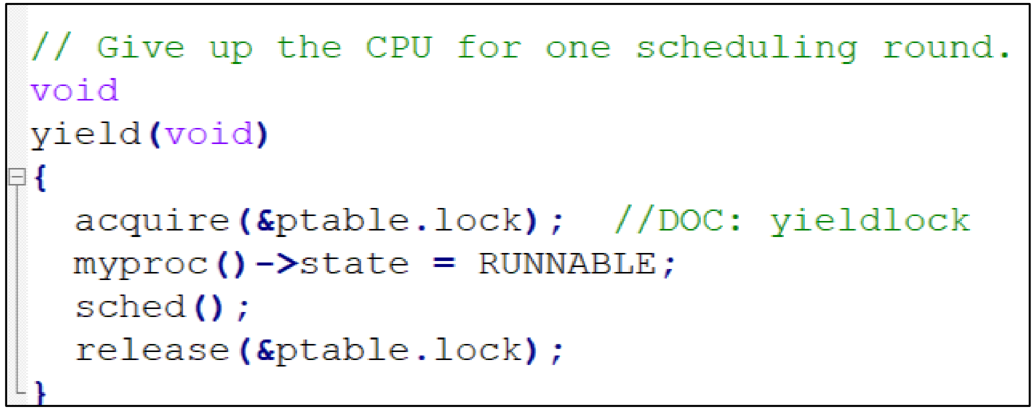


```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

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

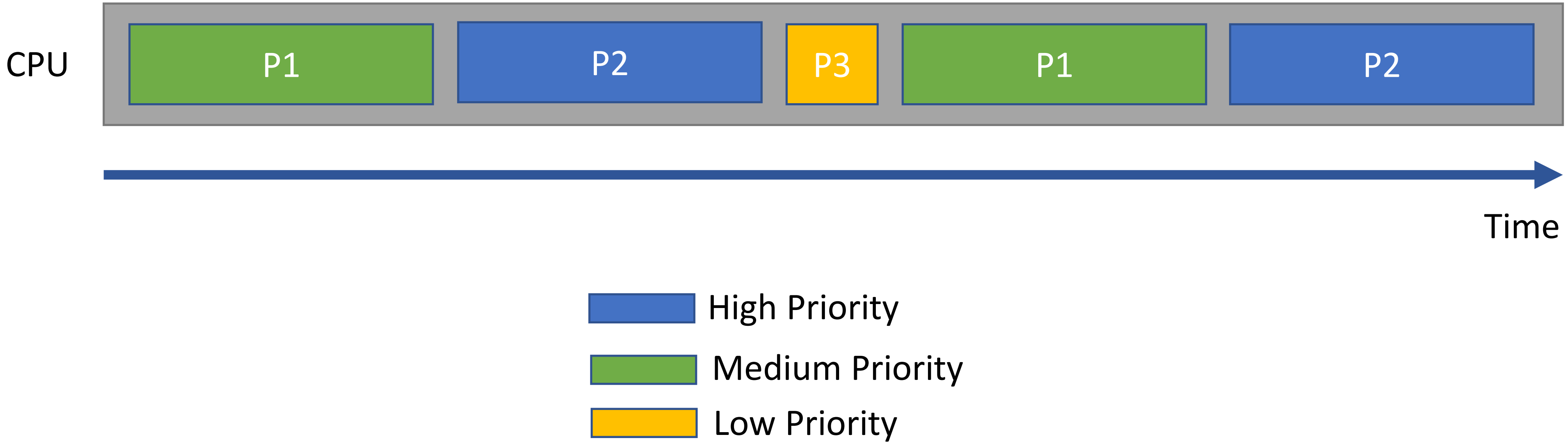
In proc.c:



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

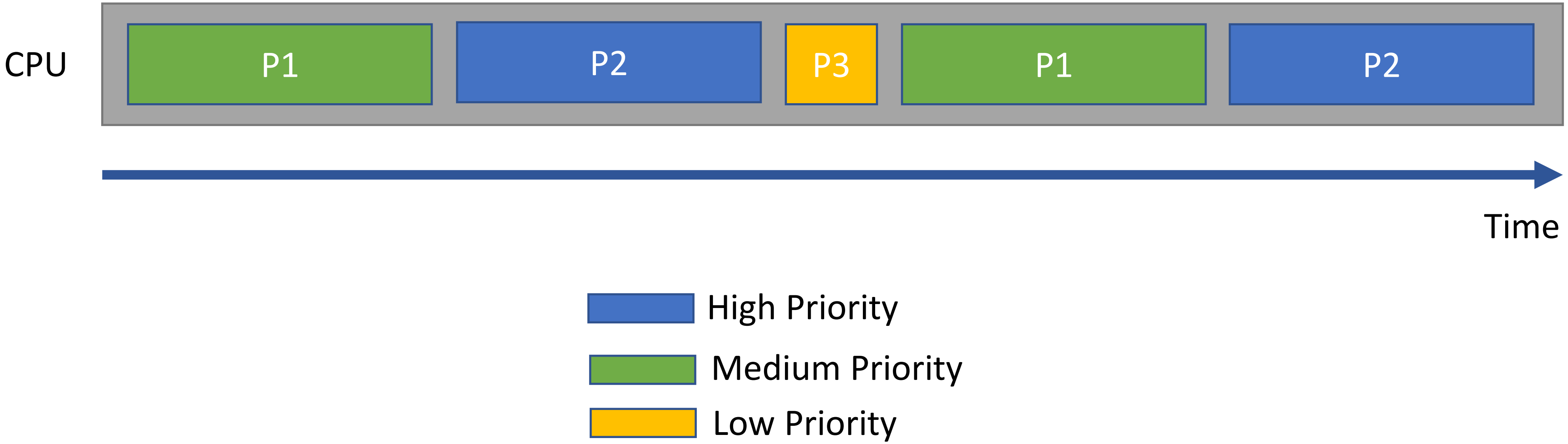
Priority scheduling of processes

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



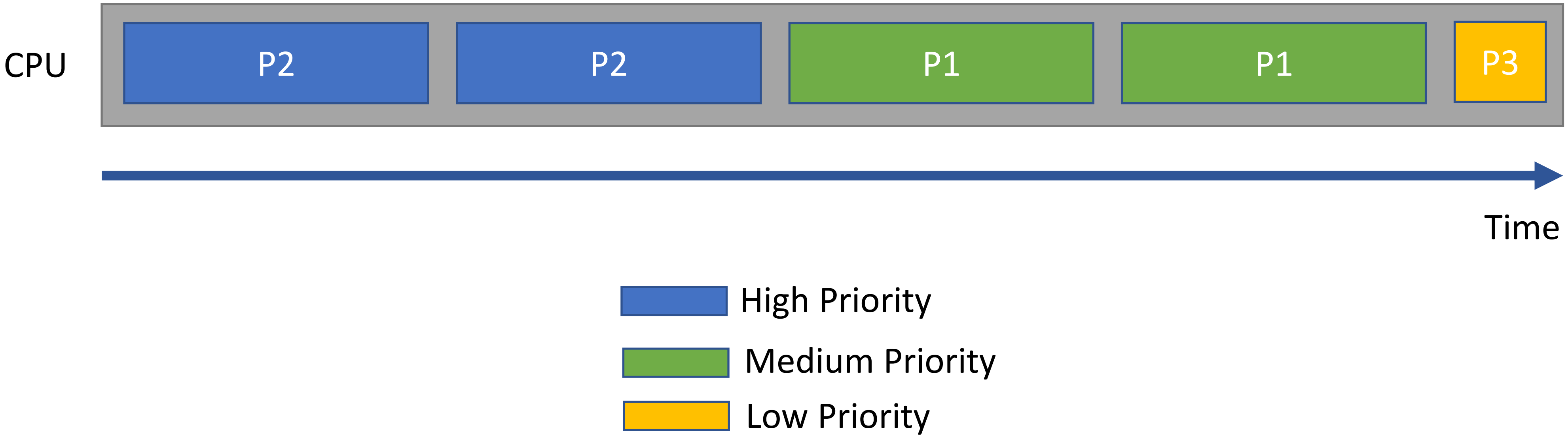
Priority scheduling of processes

- What if processes have different priorities?



Priority scheduling of processes

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



CPU

P2 P2 P1 P1 P3

Time

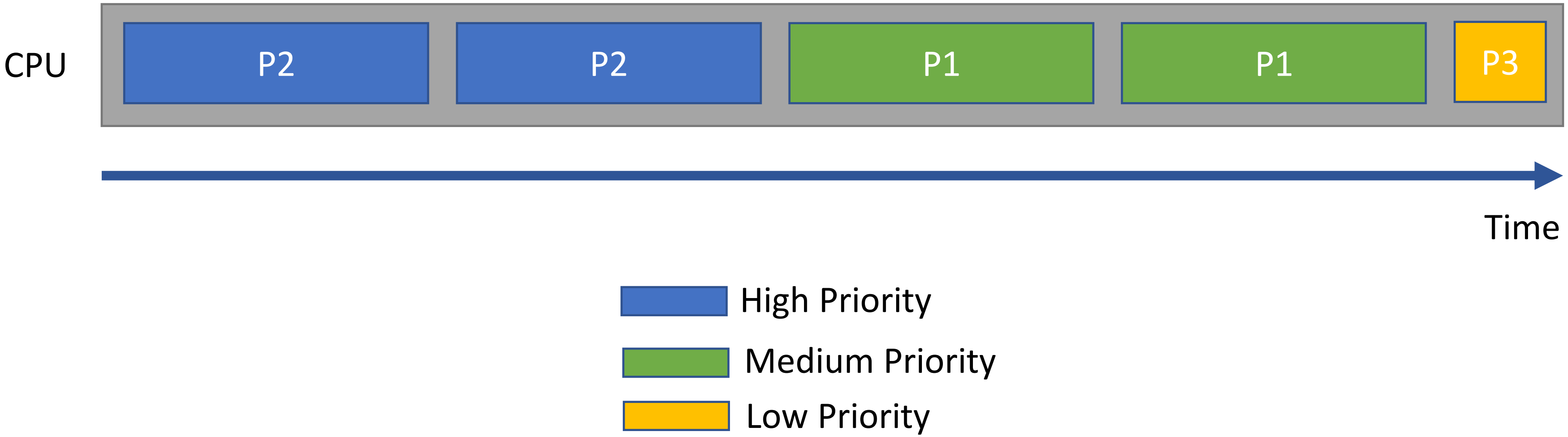
🟦 High Priority

🟩 Medium Priority

🟨 Low Priority

Priority scheduling of processes

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



CPU

P2 P2 P1 P1 P3

Time

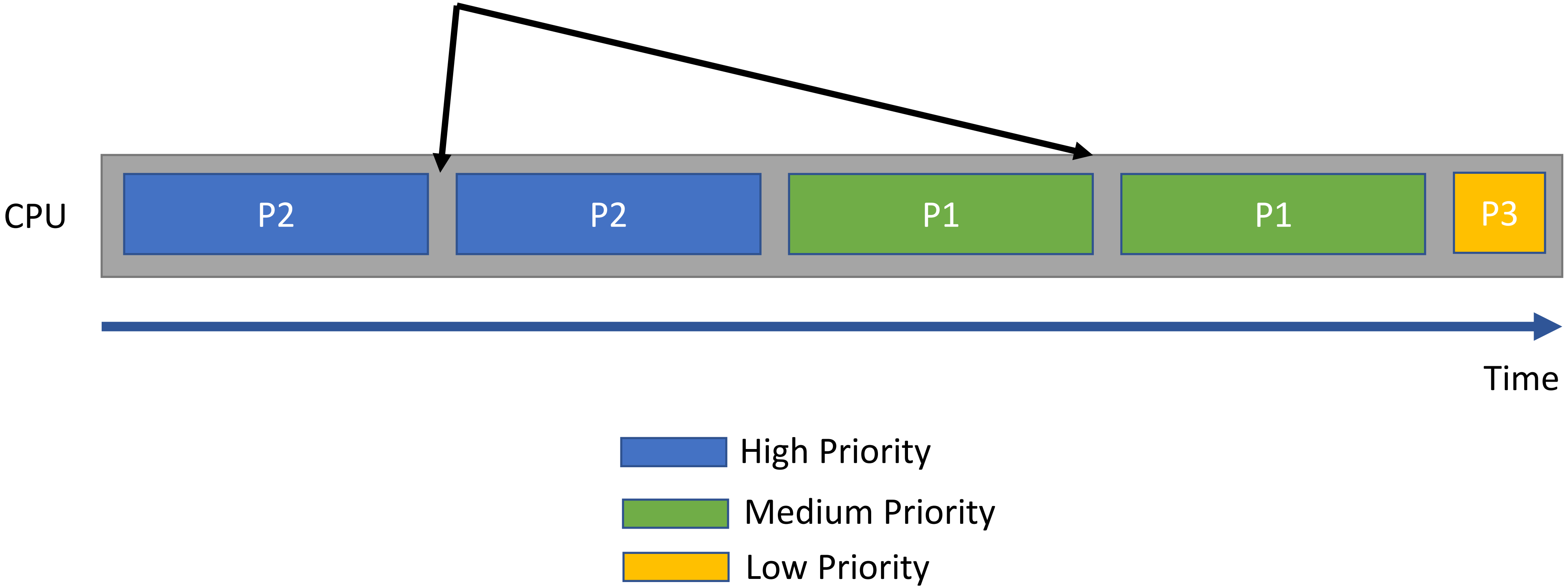
🟦 High Priority

🟩 Medium Priority

🟨 Low Priority

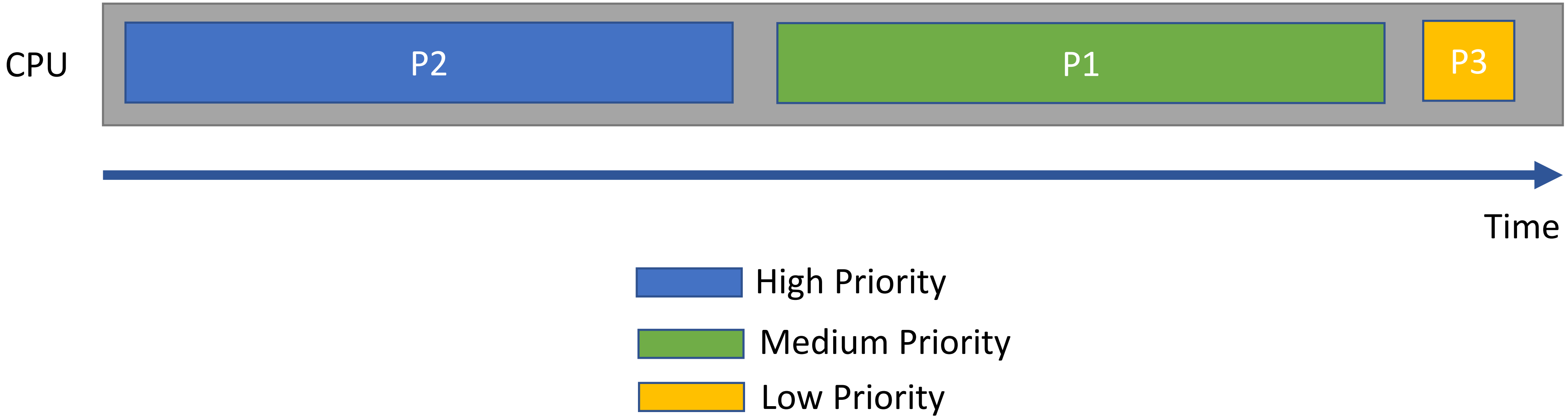
Priority scheduling of processes

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



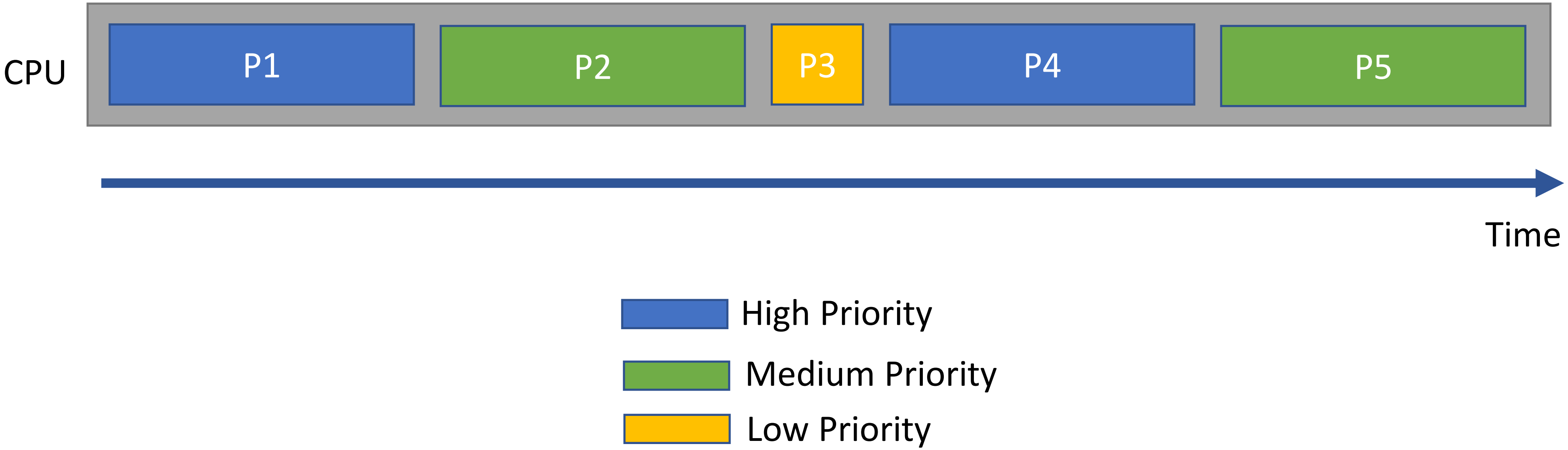
Priority scheduling of processes

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



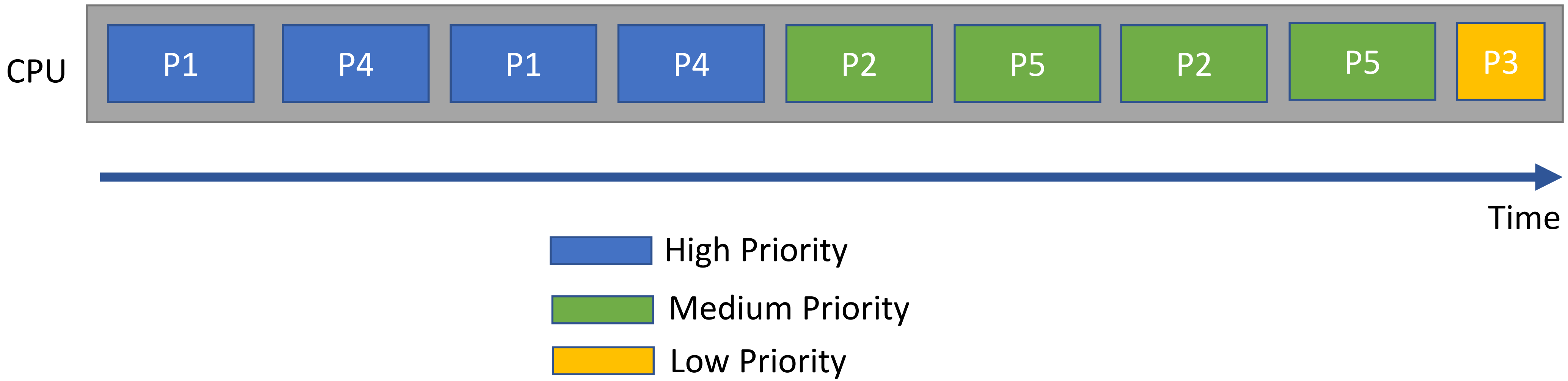
Processes with same priorities

- What if different processes have the same priorities?



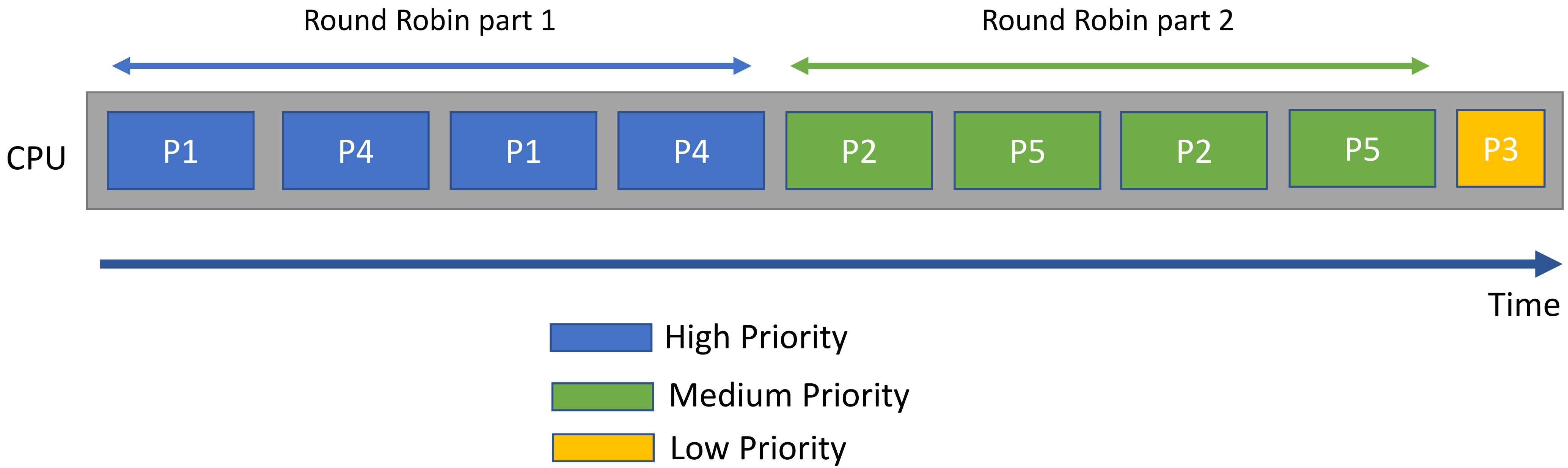
Processes with same priorities

- 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.

Synchronization Barrier

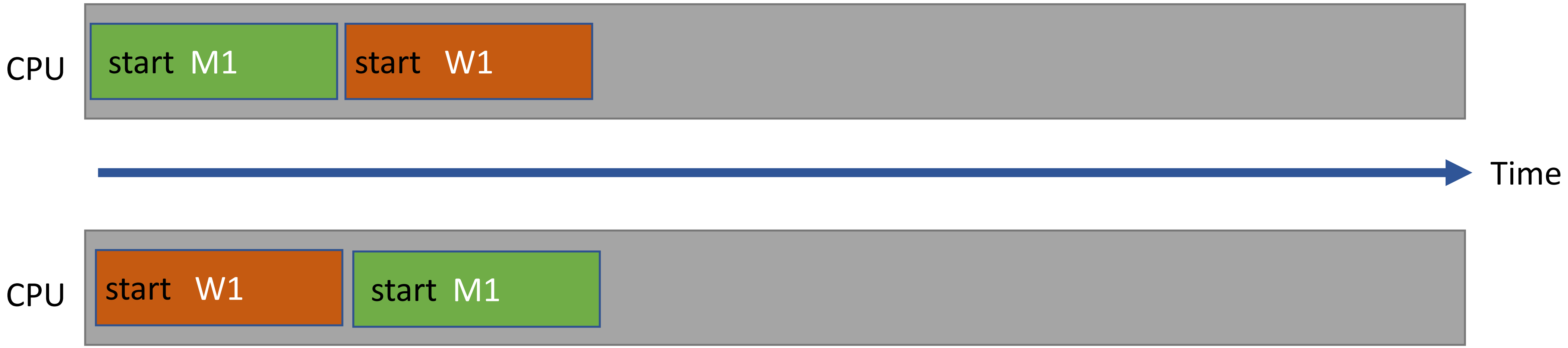
- **Question 6**
    - Pair up **men** and **women** as they enter a Friday night mixer.

Synchronization Barrier

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer
  - Each **man** and each **woman** will be represented by **one thread(Process)**

Synchronization Barrier

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer
  - Each man and each woman will be represented by one thread



Synchronization Barrier

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer.
  - Each man and each woman will be represented by one thread
  - When the **man** or **woman** enters the **mixer**, its thread will call **one** of two procedures, ***man*** or ***woman***, depending on the **thread gender**.

Synchronization Barrier

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer.
  - Each man and each woman will be represented by one thread
  - When the **man** or **woman** enters the **mixer**, its thread will call **one** of two procedures, ***man*** or ***woman***, depending on the **thread gender**.

```
Man () {

}
```

```
Woman () {

}
```

Synchronization Barrier

- **Question 6**
  - Pair up men and women as they enter a Friday night mixer.
  - Each man and each woman will be represented by one thread
  - When the man or woman enters the mixer, its thread will call one of two procedures, *man* or *woman*, depending on the thread gender.
  - Each procedure takes a single parameter, ***name***, which is just an integer name for the **thread**.

```
Man (name) {

}
```

```
Woman (name) {

}
```

Synchronization Barrier

- **Question 6**
  - The procedure **must wait** until there is an **available thread** of the opposite **gender** and must then **exchange names** with this **thread**.

```
Man (name) {

}
```

```
Woman (name) {

}
```

Synchronization Barrier

- **Question 6**
  - The procedure **must wait** until there is an **available thread** of the opposite **gender** and must then **exchange names** with this **thread**

```
Semaphore: sem = 0;
String: nameM, nameW;
```

```
Man (name) {
   nameM = name;
   }
```

```
Woman (name) {
   nameW = name;
   }
```

Synchronization Barrier

- **Question 6**
  - The procedure must wait until there is an available thread of the opposite gender and must then exchange names with this thread.
  - Each procedure must **return** the integer **name** of the thread it paired up with

```
Semaphore: sem = 0;
String: nameM, nameW;
```

```
Man (name) {
   nameM = name;
   return nameW;
}
```

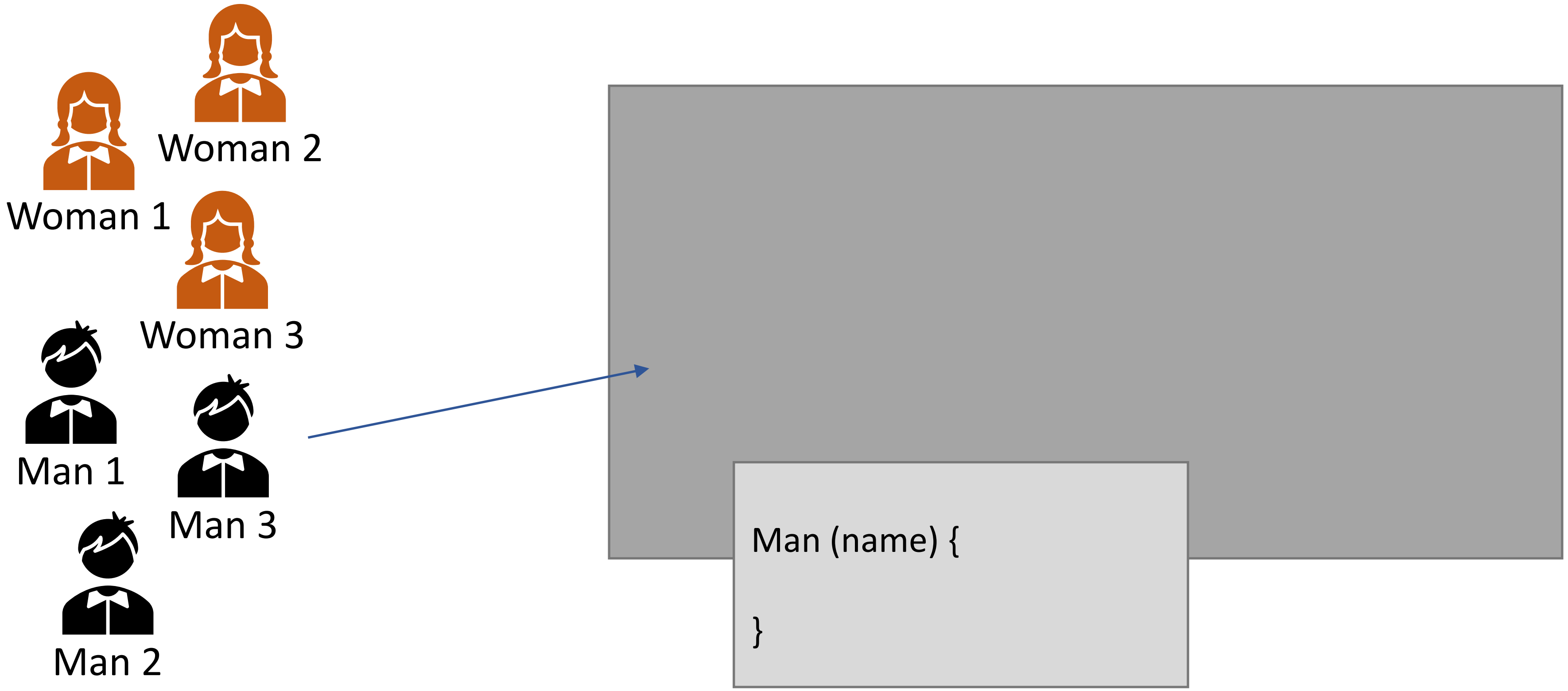
```
Woman (name) {
   nameW = name;
   return nameM;
}
```

Synchronization Barrier

- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with

Synchronization Barrier

- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with

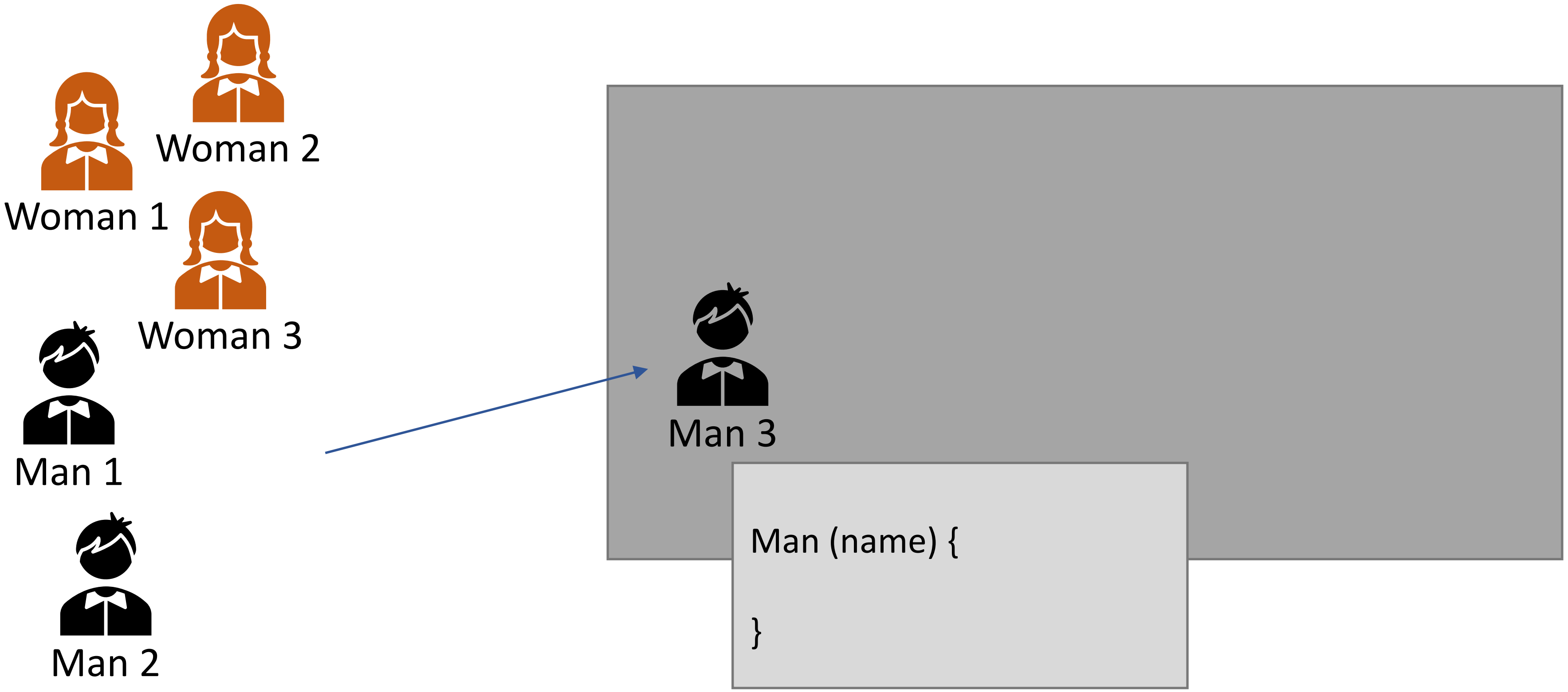


```
Man (name) {

}
```

When a Man attempts to enter a call to the **Man function** is done.

Synchronization Barrier

- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with

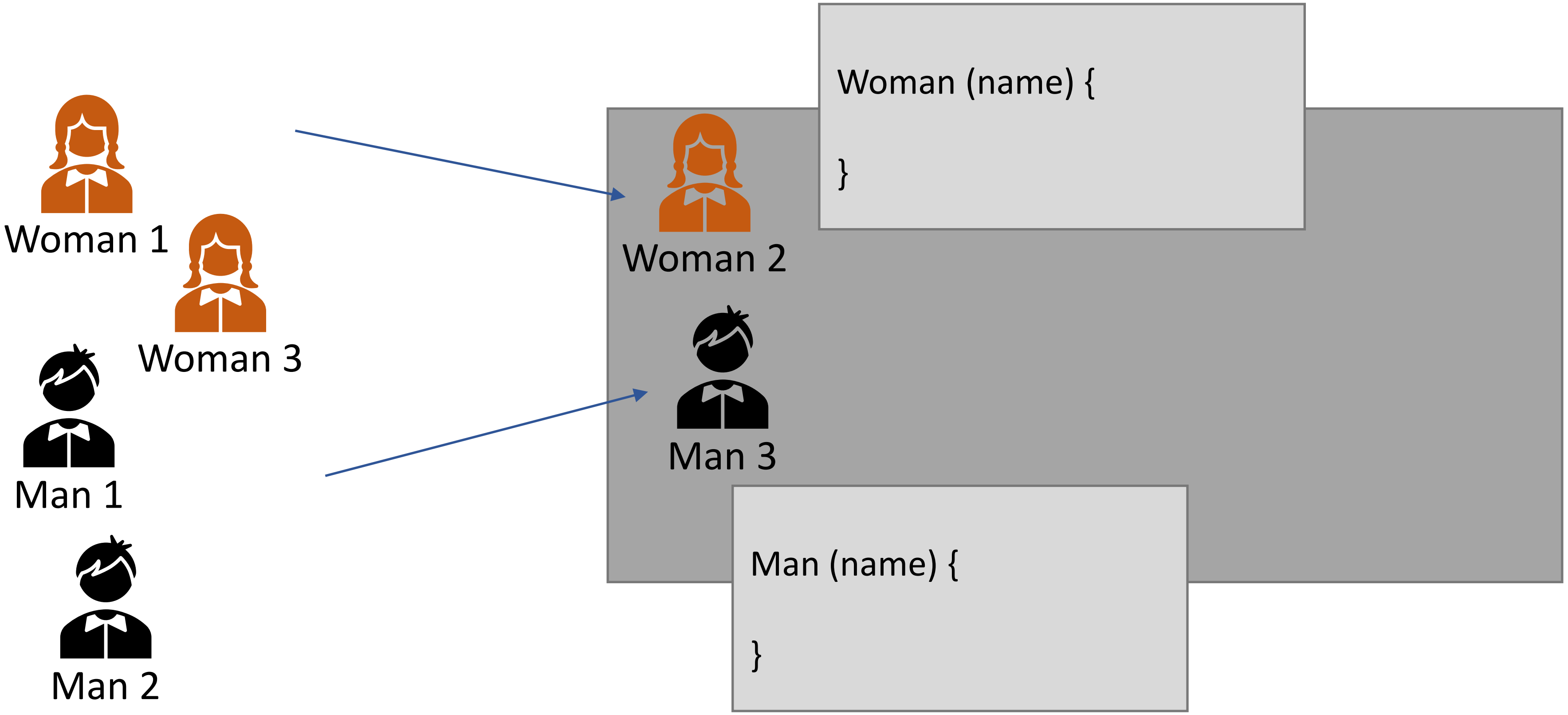


```
Man (name) {

}
```

He must **wait** to be paired with a Woman’s name.

Synchronization Barrier

- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with

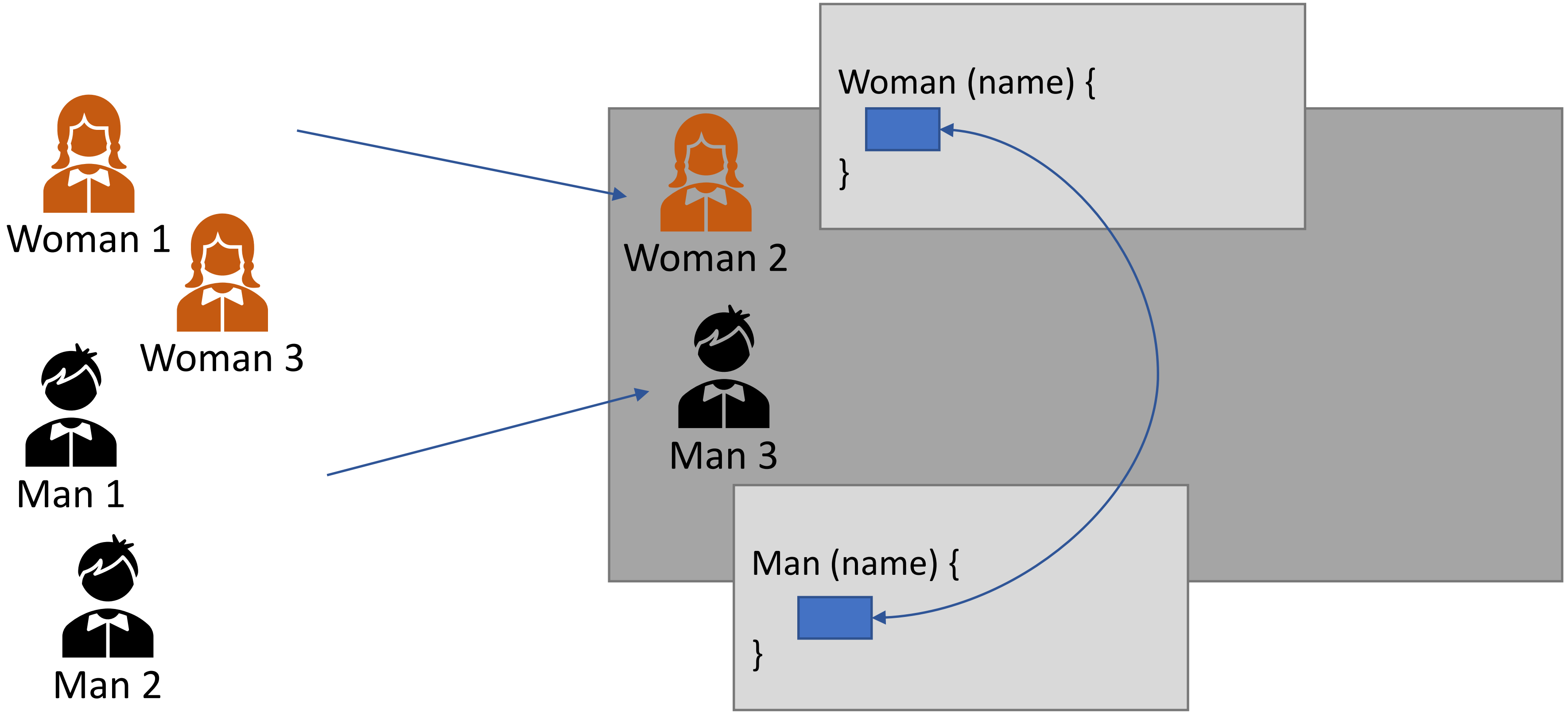


```
Woman (name) {

}
```

```
Man (name) {

}
```

Synchronization Barrier

- **Question 6**
  - Each procedure must **return** the integer **name** of the thread it paired up with



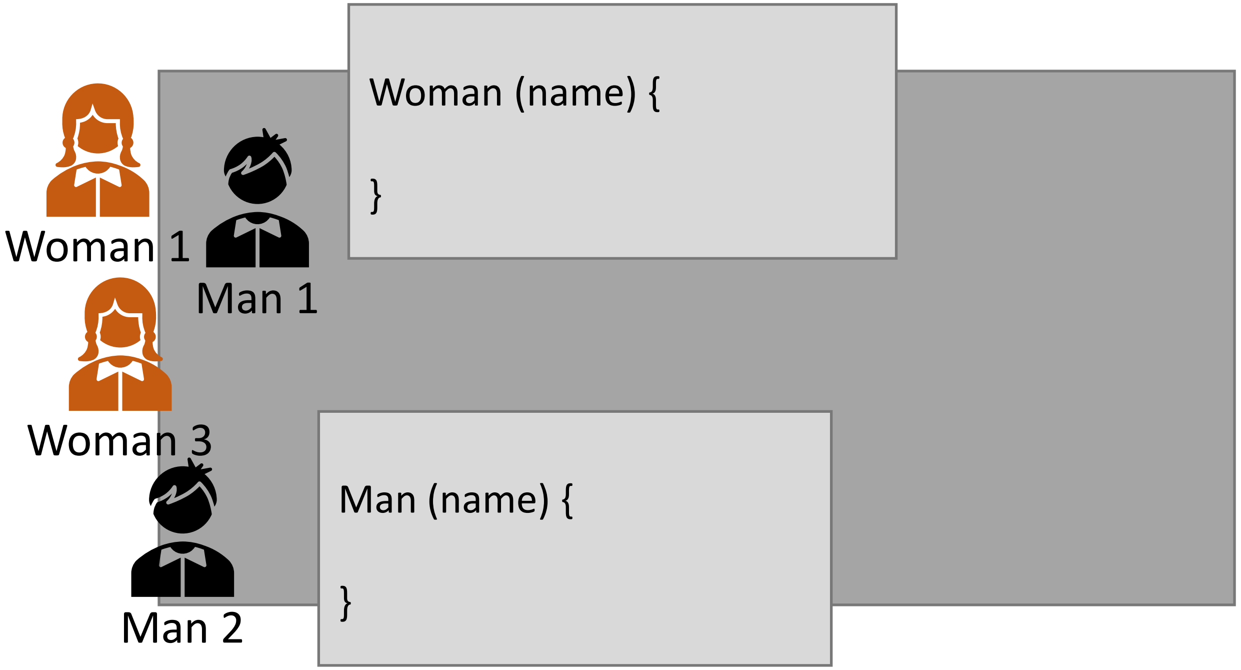
We need a **signaling mechanism** that would hold both processes/threads(Man and Woman) and only allow them to go when they are **paired**

Synchronization Barrier

- **Question 6**
  - Men and women may enter the fraternity **in any order**, and many threads may **call** the ***man*** and ***woman*** **procedures simultaneously**.

Synchronization Barrier

- **Question 6**
  - Men and women may enter the fraternity in any order, and many threads may call the *man* and *woman* procedures simultaneously.

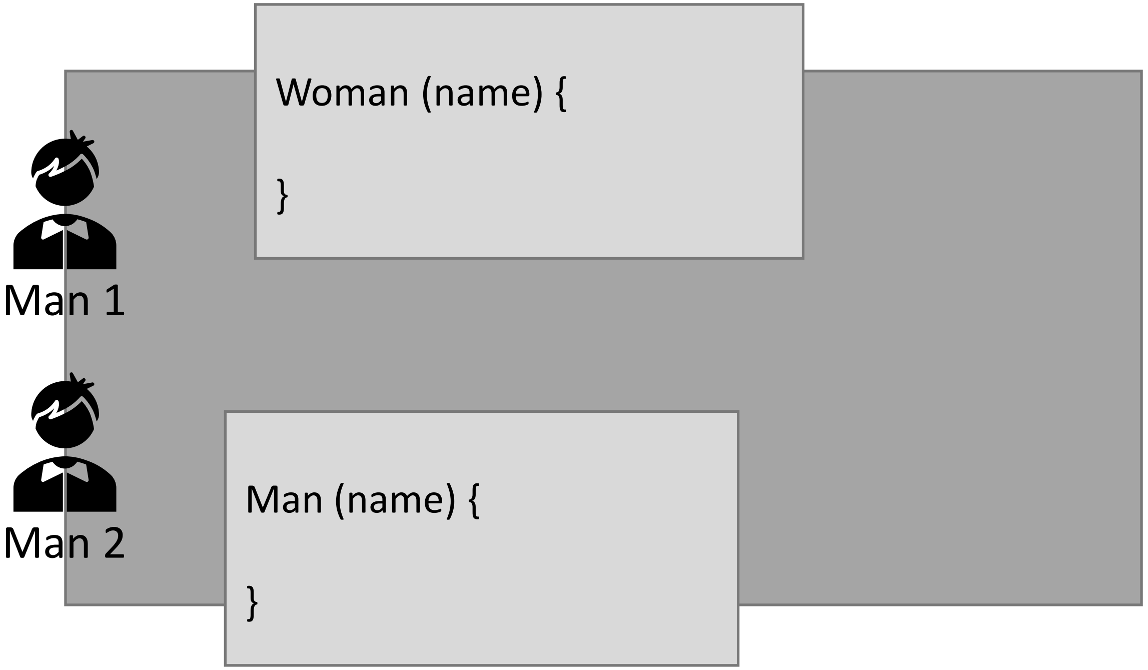


```
Woman (name) {

}
```

```
Man (name) {

}
```

Synchronization Barrier

- **Question 6**
  - Men and women may enter the fraternity in any order, and many threads may call the *man* and *woman* procedures simultaneously.

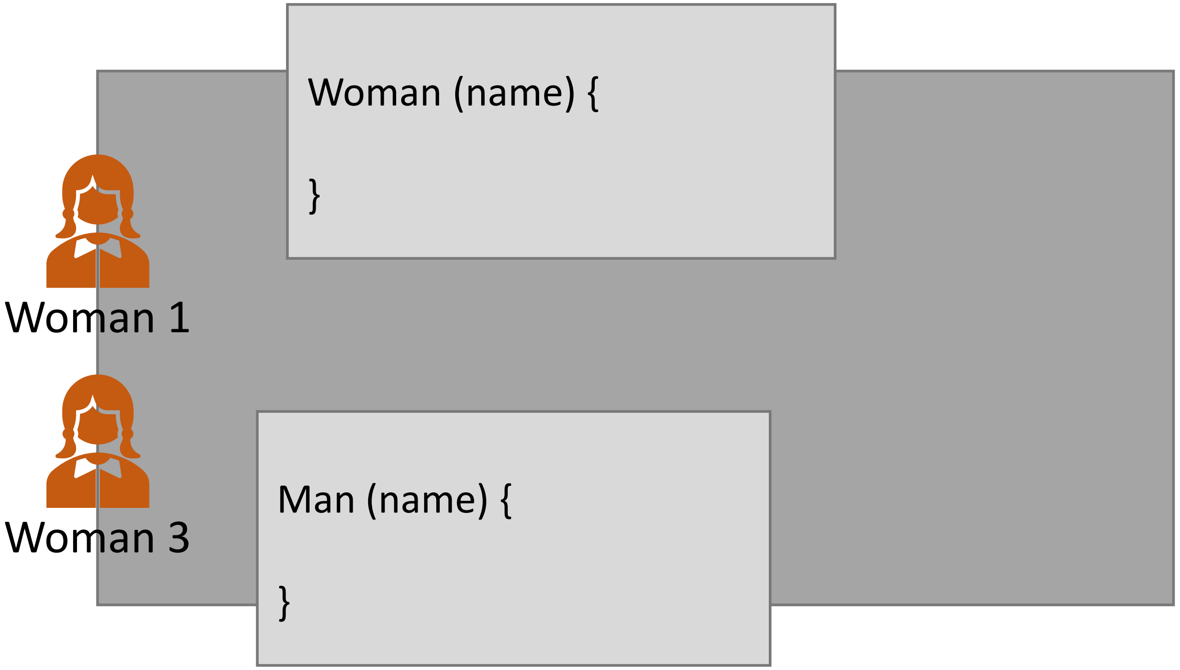


```
Woman (name) {

}
```

```
Man (name) {

}
```

Synchronization Barrier

- **Question 6**
  - Men and women may enter the fraternity in any order, and many threads may call the *man* and *woman* procedures simultaneously.



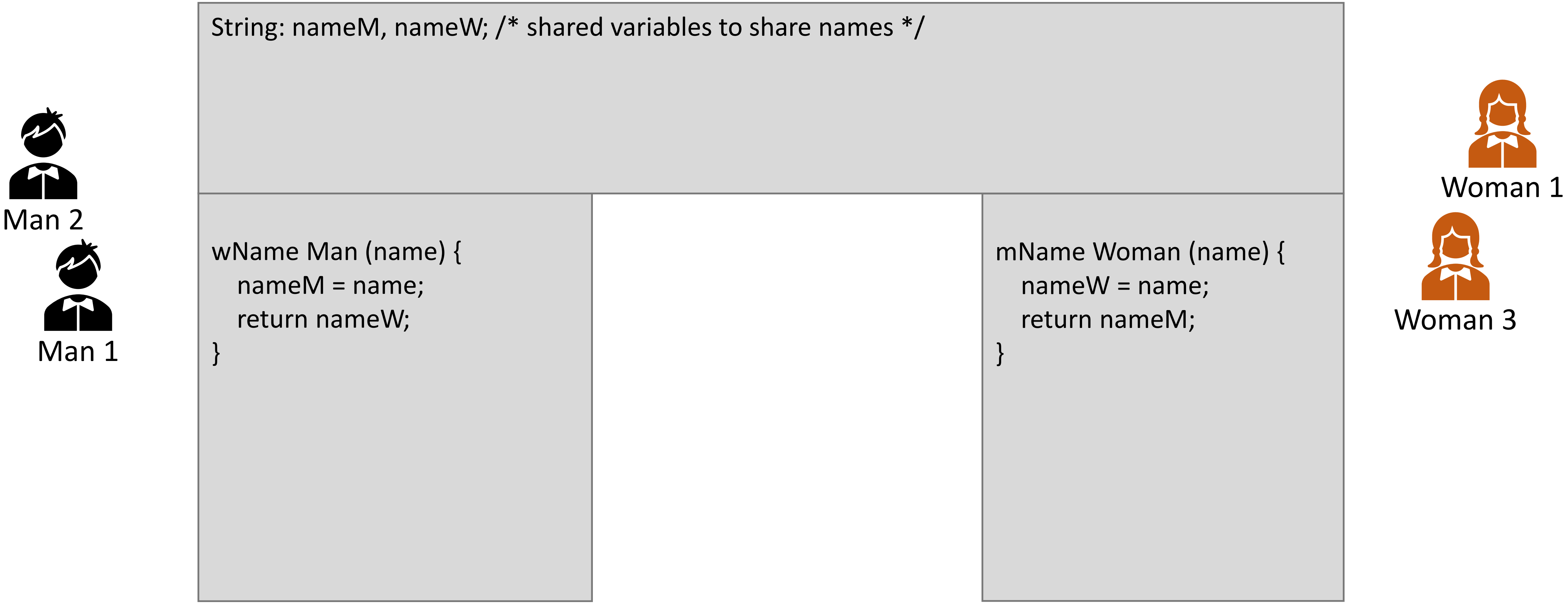
```
Woman (name) {

}
```

```
Man (name) {

}
```

Synchronization Barrier

- **Question 6**
  - Men and women may enter the fraternity in any order, and many threads may call the *man* and *woman* procedures simultaneously.
  - It doesn't **matter which man** is paired up with **which woman** (Pitt frats aren't very choosy in this exercise), as long as each pair contains one man and one woman, and each gets the other's name.
  - Use semaphores and shared variables to implement the **two procedures**.

Synchronization Barrier

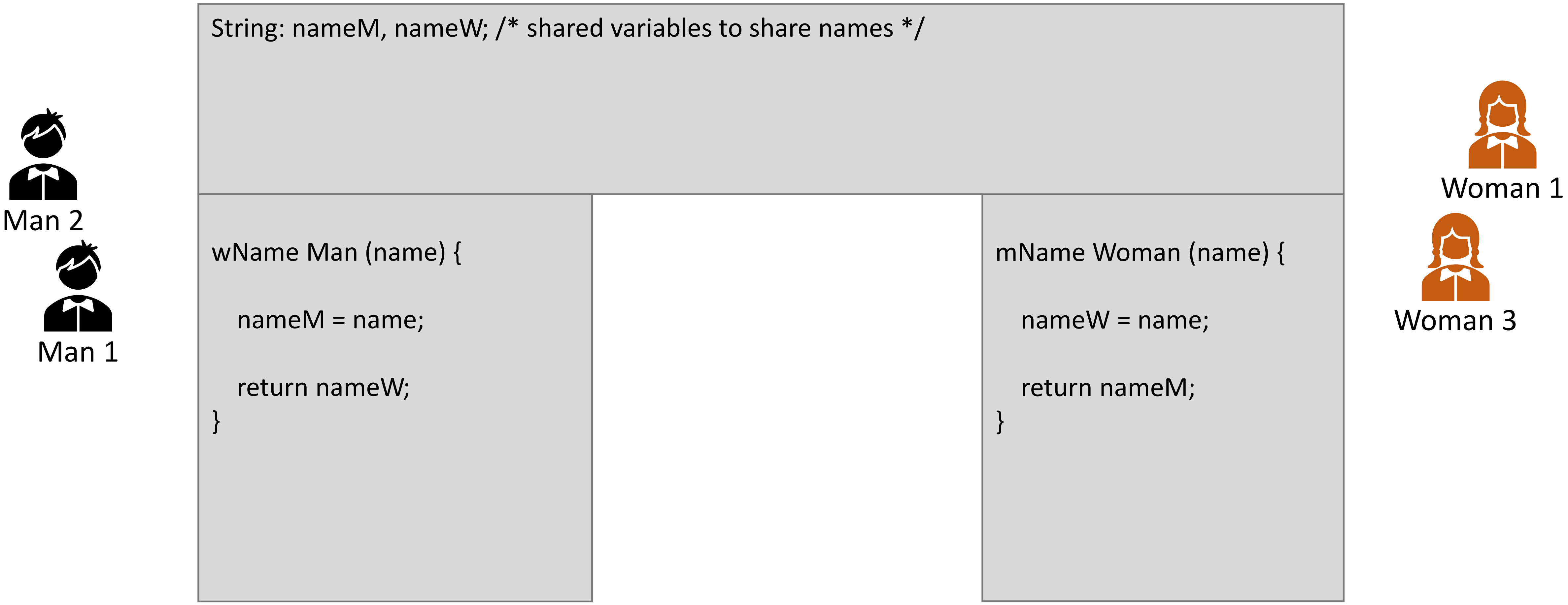


```
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   nameM = name;
   return nameW;
}
```

```
mName Woman (name) {
   nameW = name;
   return nameM;
}
```

Synchronization Barrier

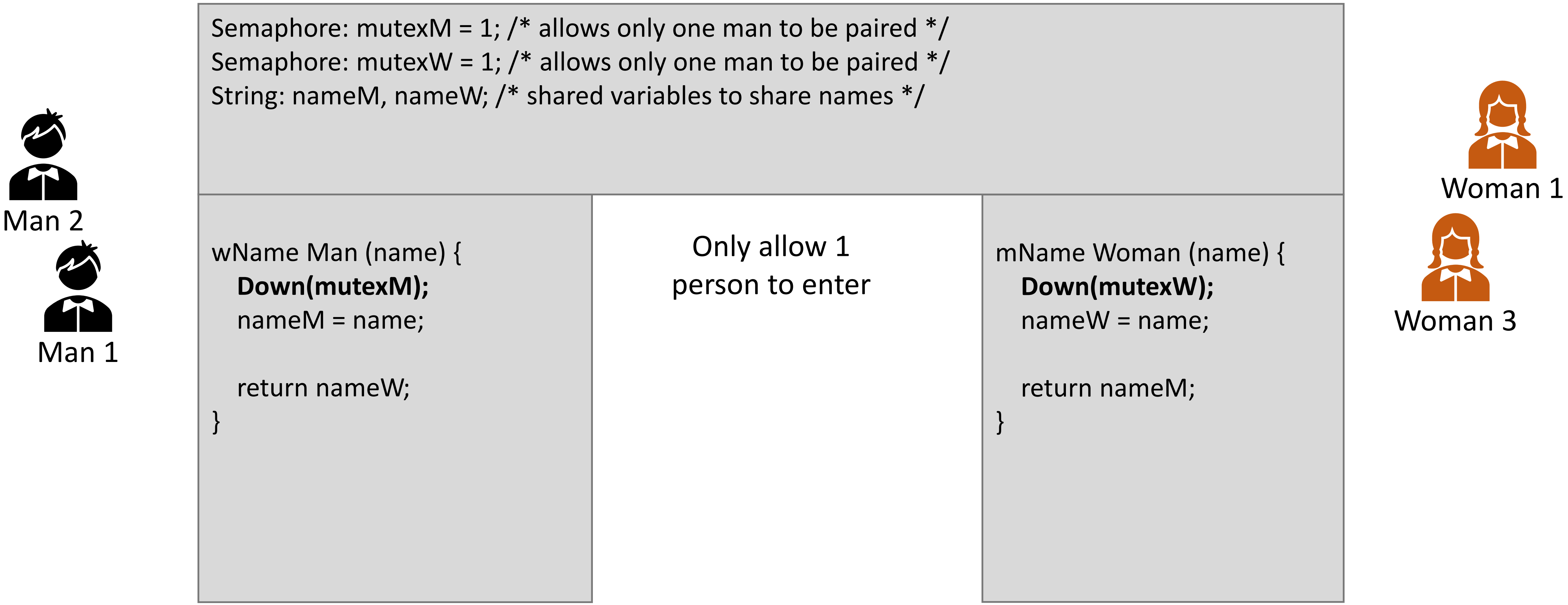


```
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {

   nameM = name;

   return nameW;
}
```

```
mName Woman (name) {

   nameW = name;

   return nameM;
}
```

Synchronization Barrier



```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;

   return nameW;
}
```

Only allow 1 person to enter

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;

   return nameM;
}
```

Synchronization Barrier

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

Man 2

Man 1

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexM);
   return nameW;
}
```

Only allow 1 person to enter

Should we allow each process to signal back to the same gender?

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;

   return nameM;
}
```

Woman 1

Woman 3

Synchronization Barrier

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

Man 2

Man 1

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexM);
   return nameW;
}
```

Only allow 1 person to enter

Should we allow each process to signal back to the same gender?

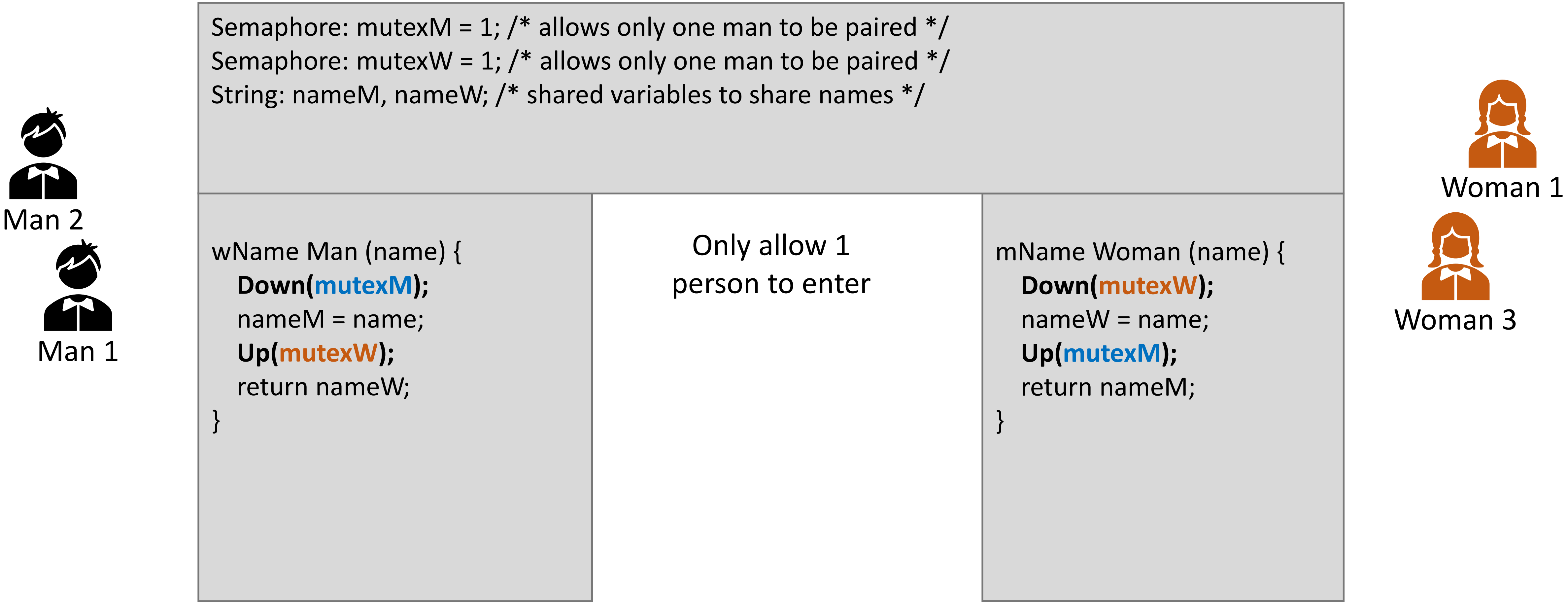
No, multiple Mans would overwrite each others name.

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;

   return nameM;
}
```

Woman 1

Woman 3

Synchronization Barrier



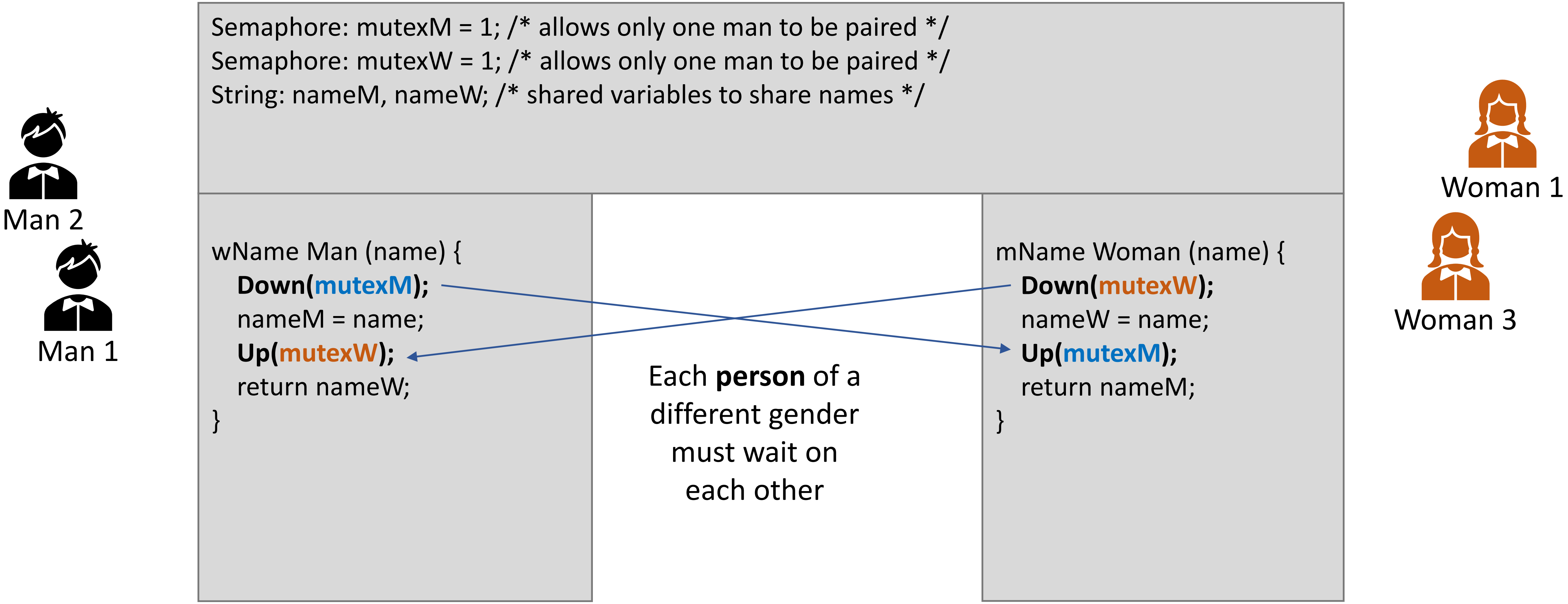
```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

Only allow 1 person to enter

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Synchronization Barrier



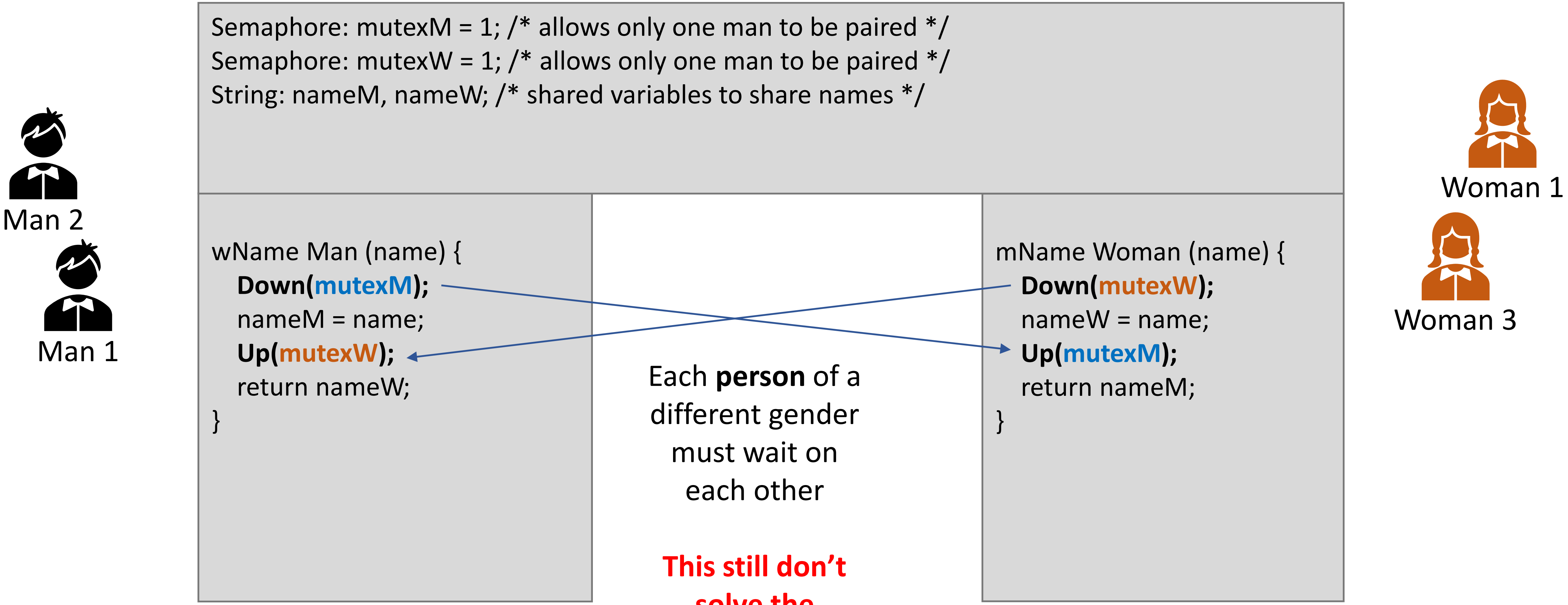
```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

Each **person** of a different gender must wait on each other

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Synchronization Barrier



```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

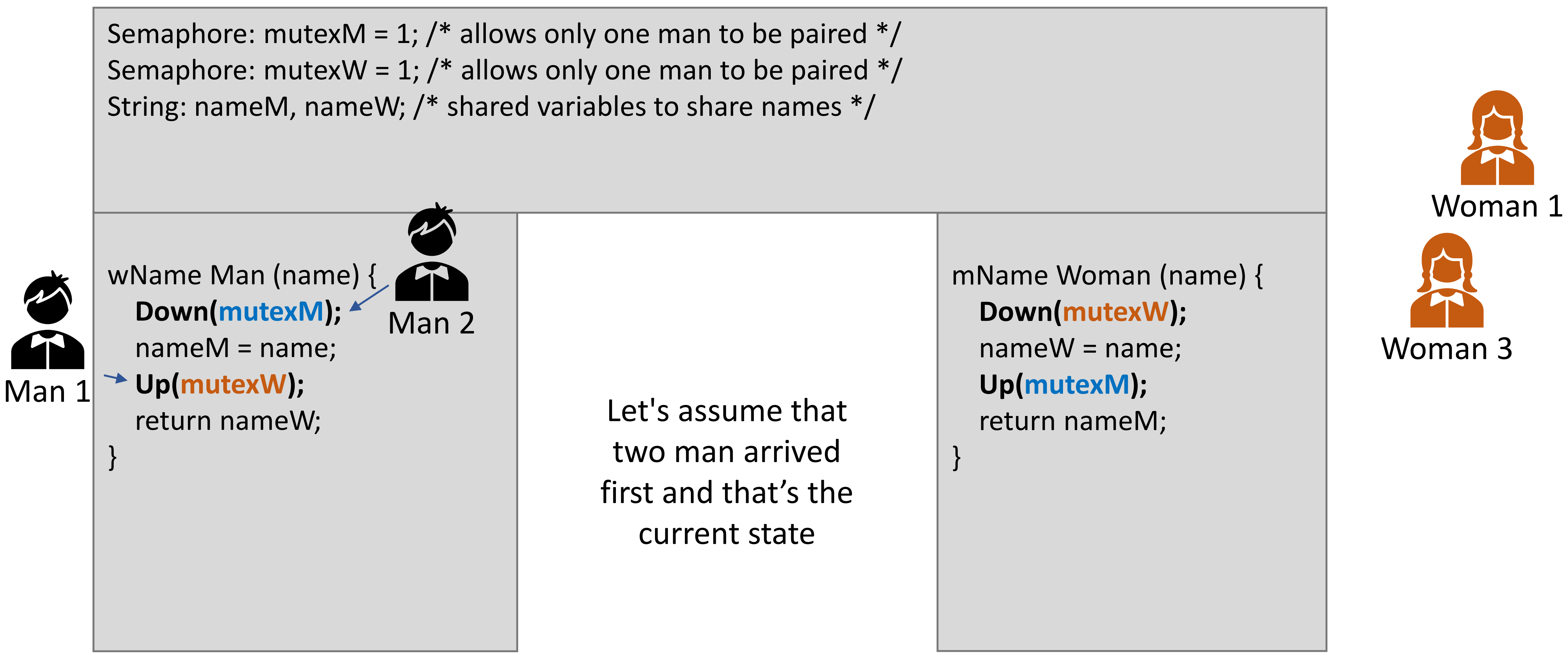
```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Each **person** of a different gender must wait on each other

This still don’t solve the problem

Synchronization Barrier



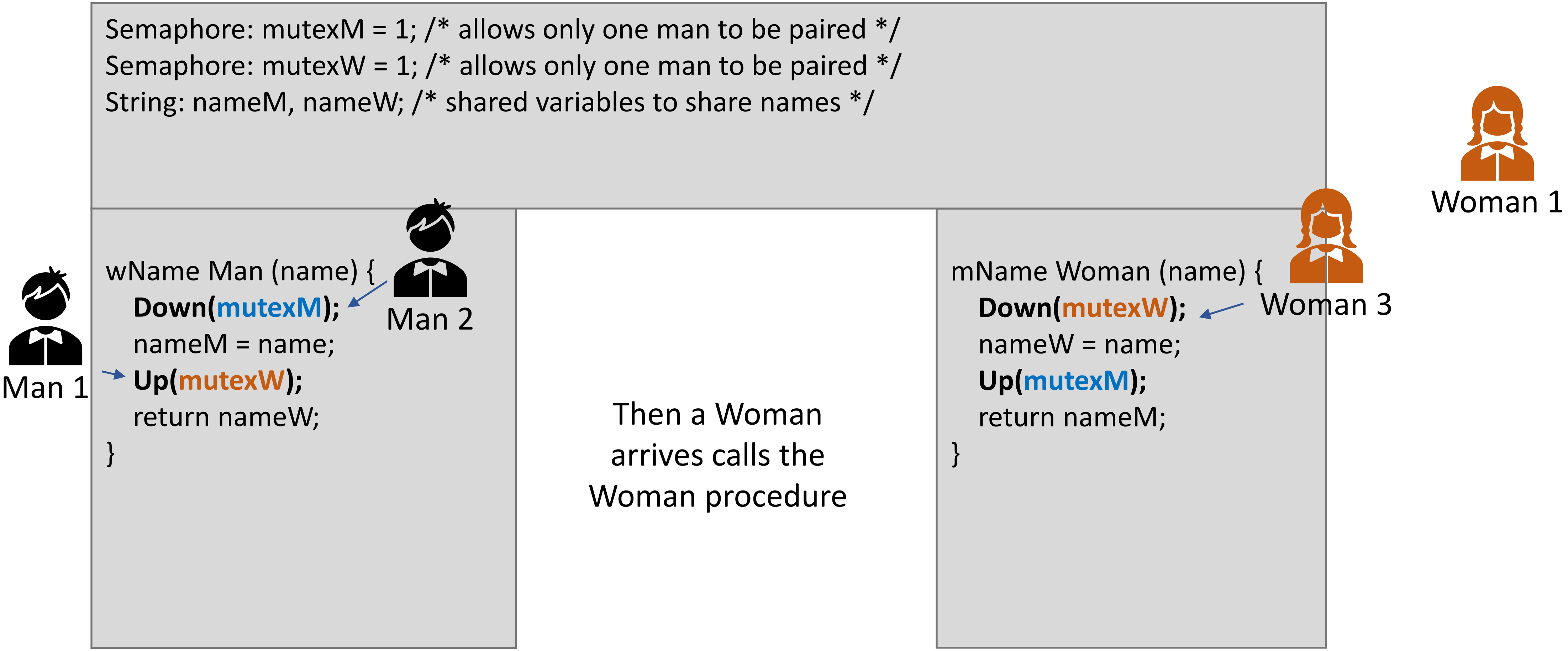
```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Let's assume that two man arrived first and that’s the current state

Synchronization Barrier



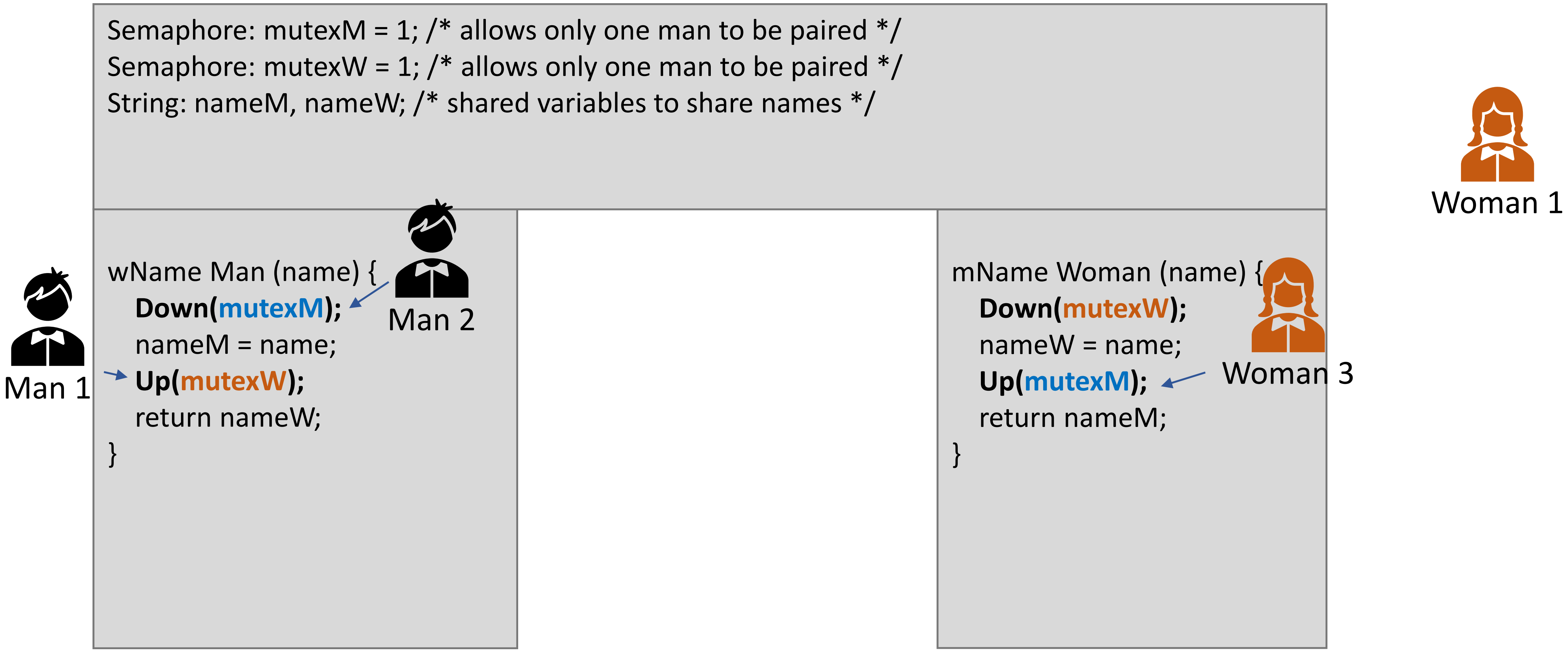
```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

Then a Woman arrives calls the Woman procedure

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Synchronization Barrier

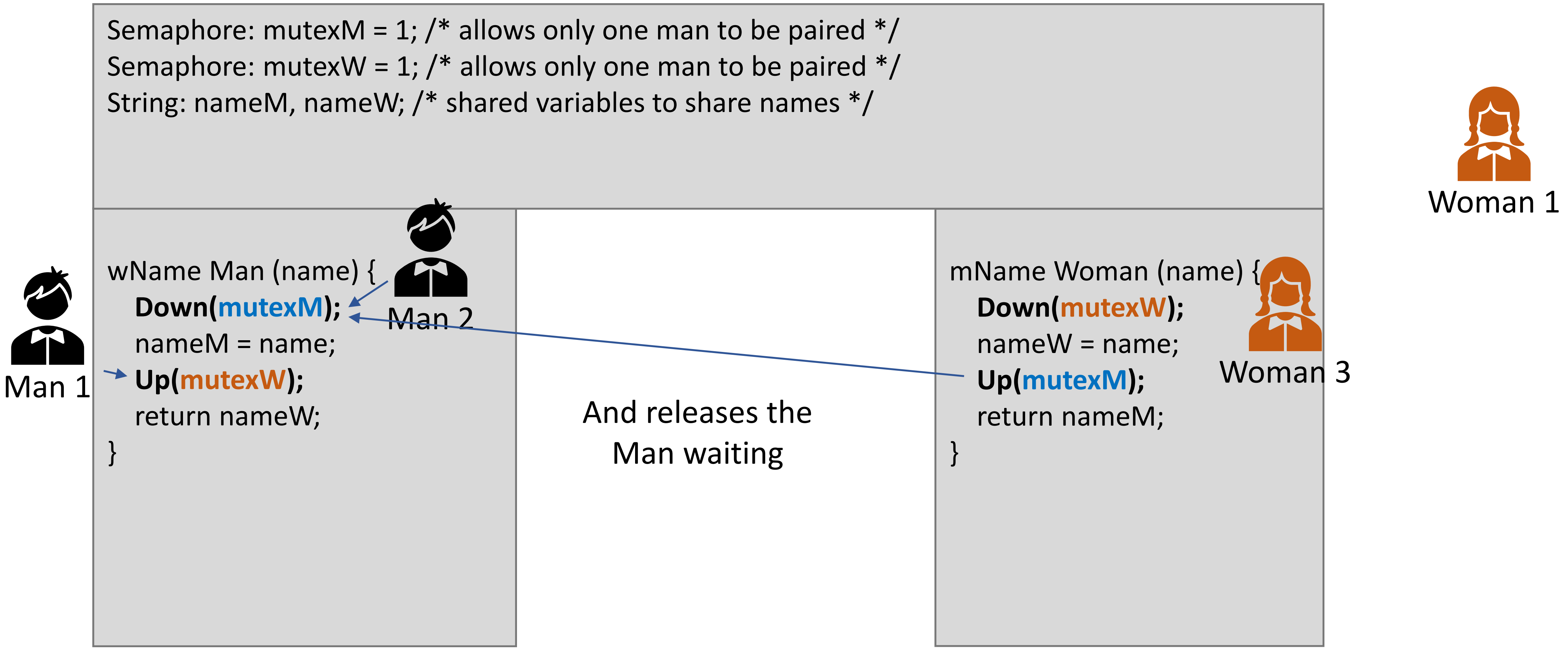


```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Synchronization Barrier



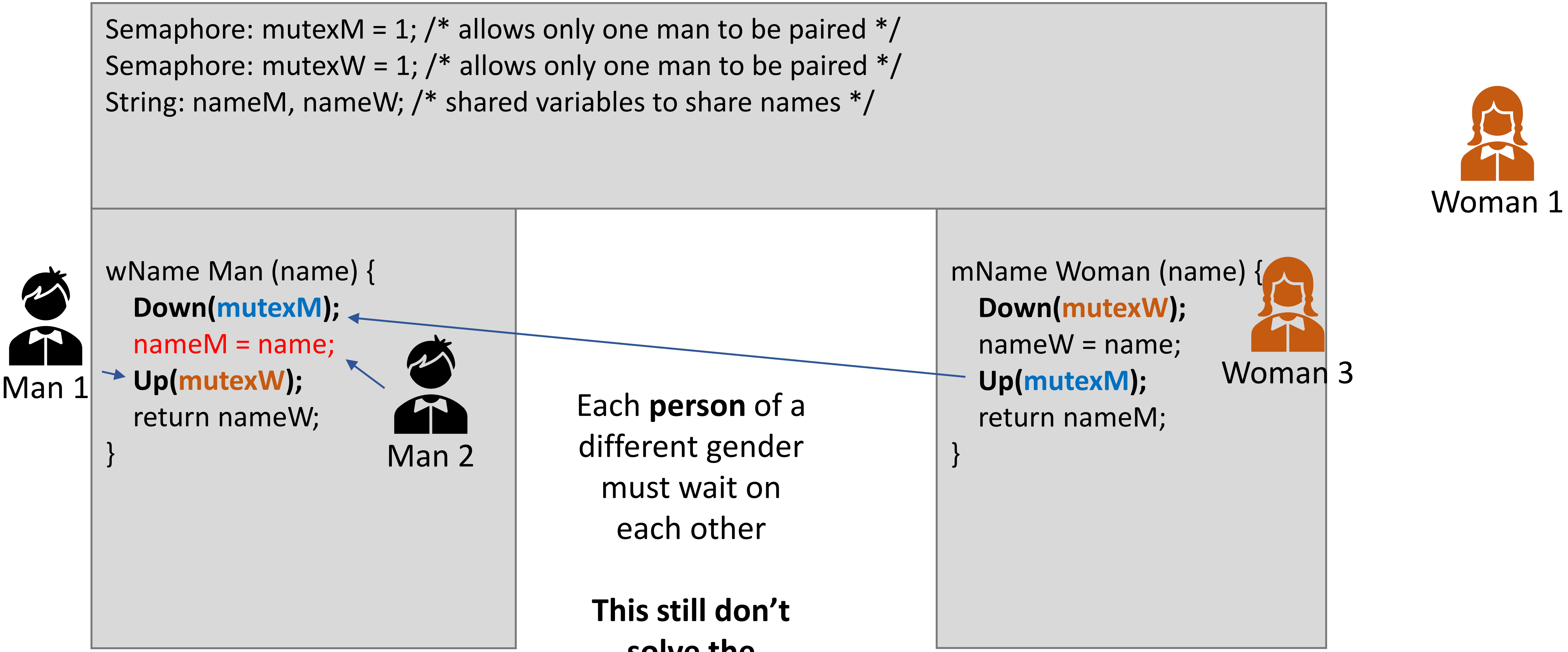
```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
    Down(mutexM);
    nameM = name;
    Up(mutexW);
    return nameW;
}
```

And releases the Man waiting

```
mName Woman (name) {
    Down(mutexW);
    nameW = name;
    Up(mutexM);
    return nameM;
}
```

Synchronization Barrier



```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Each **person** of a different gender must wait on each other

This still don’t solve the problem

Synchronization Barrier

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
String: nameM, nameW; /* shared variables to share names */
```

Man 2

Man 1

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(mutexW);
   return nameW;
}
```

We need to also that a woman can only return the name of a single man

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(mutexM);
   return nameM;
}
```

Woman 1

Woman 3

Synchronization Barrier

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Down(waitW);
   Up(mutexW);
   return nameW;
}
```

We need to also that a woman can only return the name of a single man

We needs processes to signal each other

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Down(waitM);
   Up(mutexM);
   return nameM;
}
```

Woman 1

Woman 3

Synchronization Barrier

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Down(waitW);
   Up(mutexW);
   return nameW;
}
```

We need to also that a woman can only return the name of a single man

We needs processes to signal each other

Now each is waiting on each other **on deadlock**

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Down(waitM);
   Up(mutexM);
   return nameM;
}
```

Woman 1

Woman 3

Synchronization Barrier

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);

   Up(mutexW);
   return nameW;
}
```

We need to also that a woman can only return the name of a single man

We needs processes to signal each other

Now each is waiting on each other **on deadlock**

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);

   Up(mutexM);
   return nameM;
}
```

Woman 1

Woman 3

Synchronization Barrier

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0; /*  allows woman to wait for man */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);

   Up(mutexW);
   return nameW;
}
```

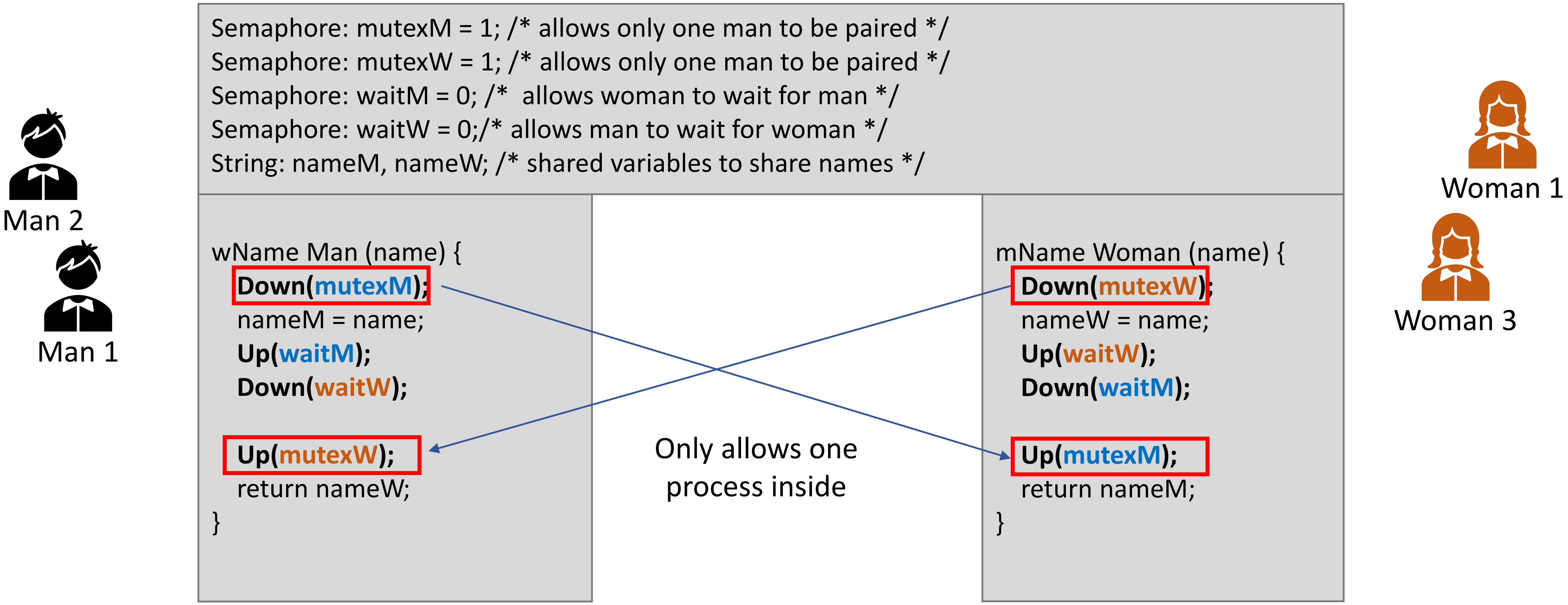
Makes processes wait for each other

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);

   Up(mutexM);
   return nameM;
}
```

Woman 1

Woman 3

Synchronization Barrier



```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);

   Up(mutexW);
   return nameW;
}
```

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);

   Up(mutexM);
   return nameM;
}
```

Only allows one process inside

Synchronization Barrier

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);

   Up(mutexW);
   return nameW;
}
```

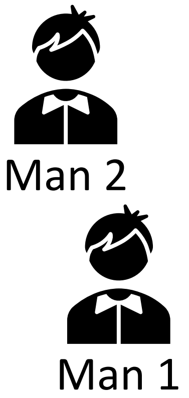
We still have a problem. We cannot return directly the shared **global** variable value. It mays **still be changed**.

```
mName Woman (name) {
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);

   Up(mutexM);
   return nameM;
}
```

Woman 1

Woman 3

Synchronization Barrier



```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   String temp;
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);
   temp = nameW;
   Up(mutexW);
   return temp;
}
```

We still have a problem. We cannot return directly the shared **global** variable value. It mays **still be changed**.

It must be a local variable.

```
mName Woman (name) {
   String temp;
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);
   temp = nameM;
   Up(mutexM);
   return temp;
}
```

Synchronization Barrier

Man 2

Man 1

```
Semaphore: mutexM = 1; /* allows only one man to be paired */
Semaphore: mutexW = 1; /* allows only one man to be paired */
Semaphore: waitM = 0; /*  allows woman to wait for man */
Semaphore: waitW = 0;/* allows man to wait for woman */
String: nameM, nameW; /* shared variables to share names */
```

```
wName Man (name) {
   String temp;
   Down(mutexM);
   nameM = name;
   Up(waitM);
   Down(waitW);
   temp = nameW;
   Up(mutexW);
   return temp;
}
```

Finally we have the solution!

```
mName Woman (name) {
   String temp;
   Down(mutexW);
   nameW = name;
   Up(waitW);
   Down(waitM);
   temp = nameM;
   Up(mutexM);
   return temp;
}
```

Woman 1

Woman 3

CS 1550

Week 6

Lab 3 and Project 2

Teaching Assistant

Henrique Potter