Introduction to Operating Systems
CS 1550

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Some slides are from **Silberschatz, Galvin and Gagne ©2013)**

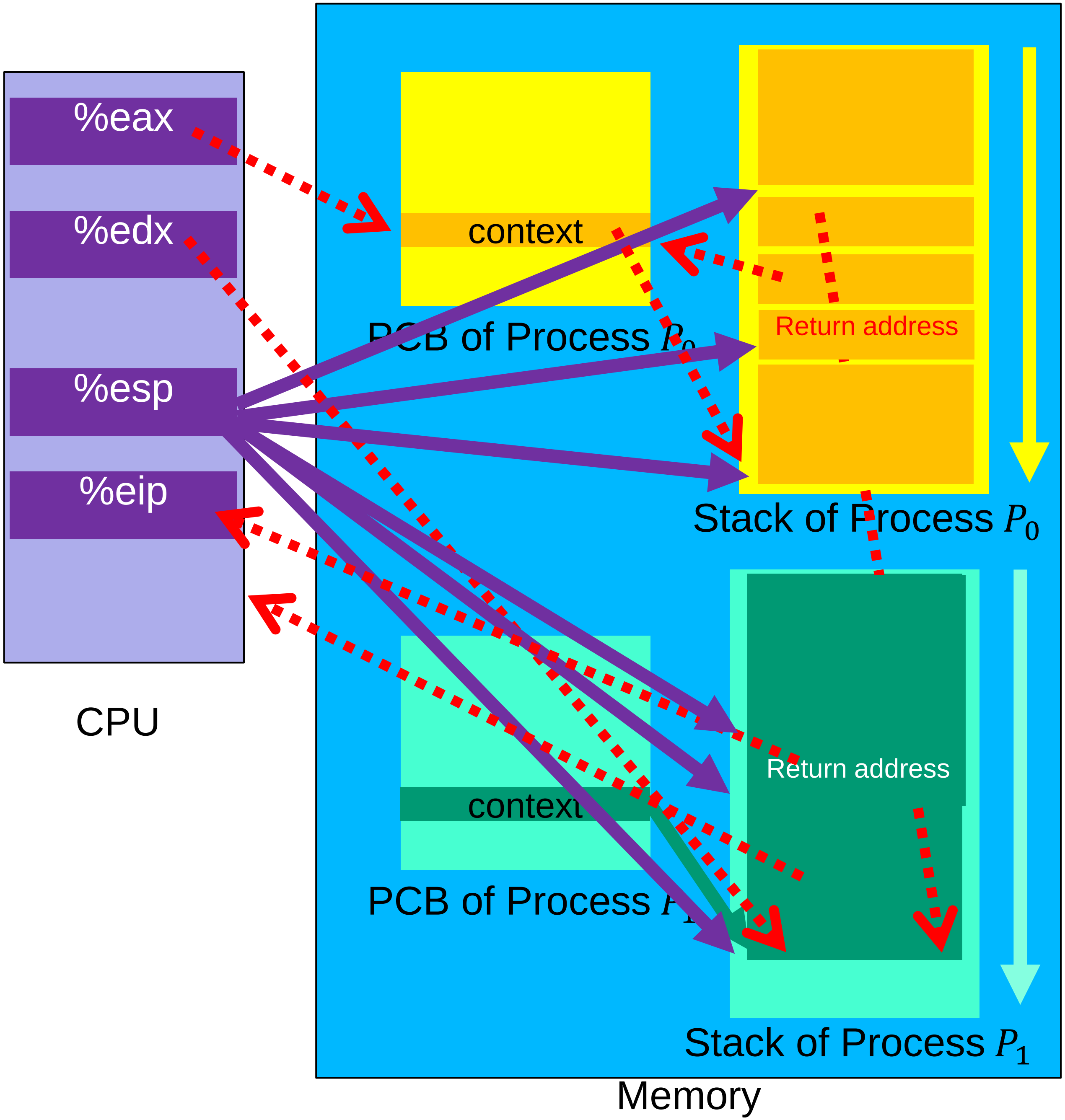
Announcements

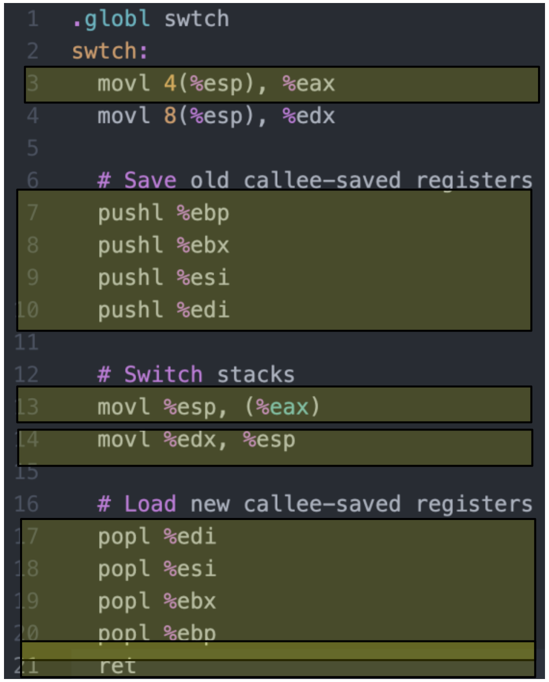
- Homework 1 is due tomorrow at 11:59 pm
- Recitations start this week
- Project 1 will be posted on Canvas this Friday
- Docker images for labs and projects are available on Canvas
  - As an alternative to running the labs and projects on the Thoth server

Agenda

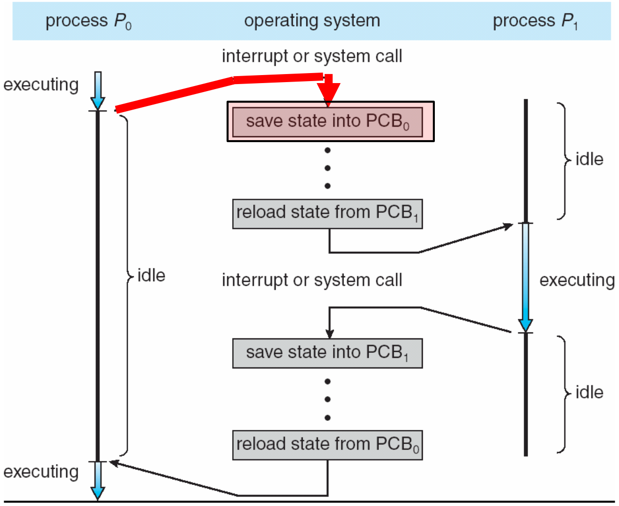
- Context Switching
- Critical Region as a solution to the Race Condition problem
- Spinlocks to implement Critical Region
- Busy Waiting Problem
- Why does it happen?
- What are its implications?

Context Switching in Xv6



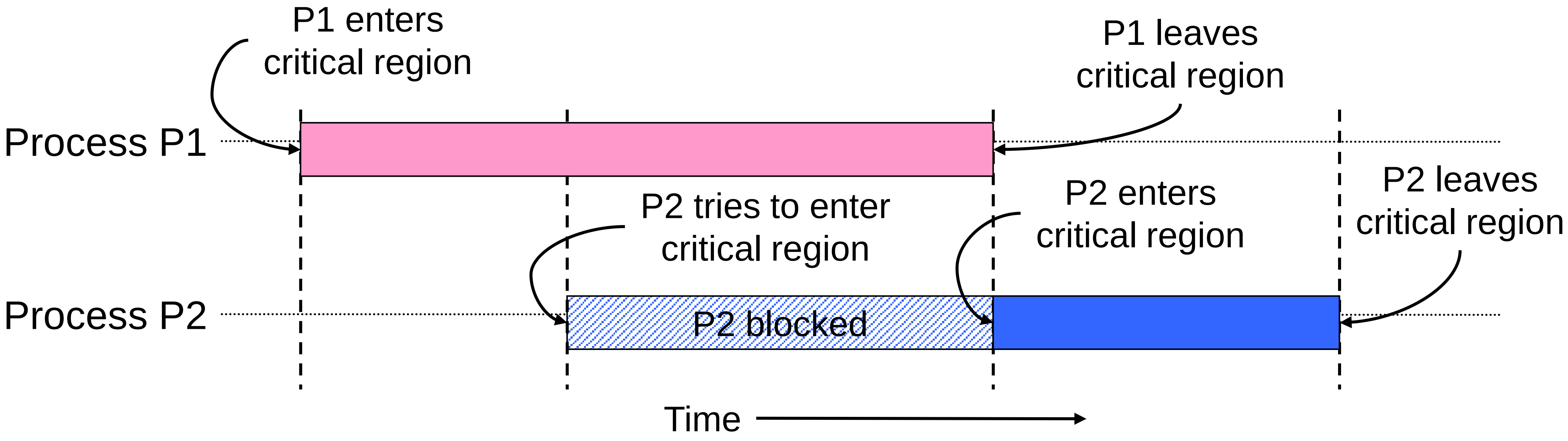


```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```



Critical regions

- Back to the race conditions problem
- Use critical regions to provide *mutual exclusion* and help fix race conditions
- Let’s put the statement x++ in a critical region



How to implement critical regions?

- Turn-based solutions
- Spinlocks
- Semaphores
- Monitors

Using Spinlocks

```
Spinlock lock;
```

Code for process P_i

```
While(1){
  Lock(lock)
  // critical section
  Unlock(lock);
  // remainder of code
  }
}
```

Spinlock implementation (1/2)

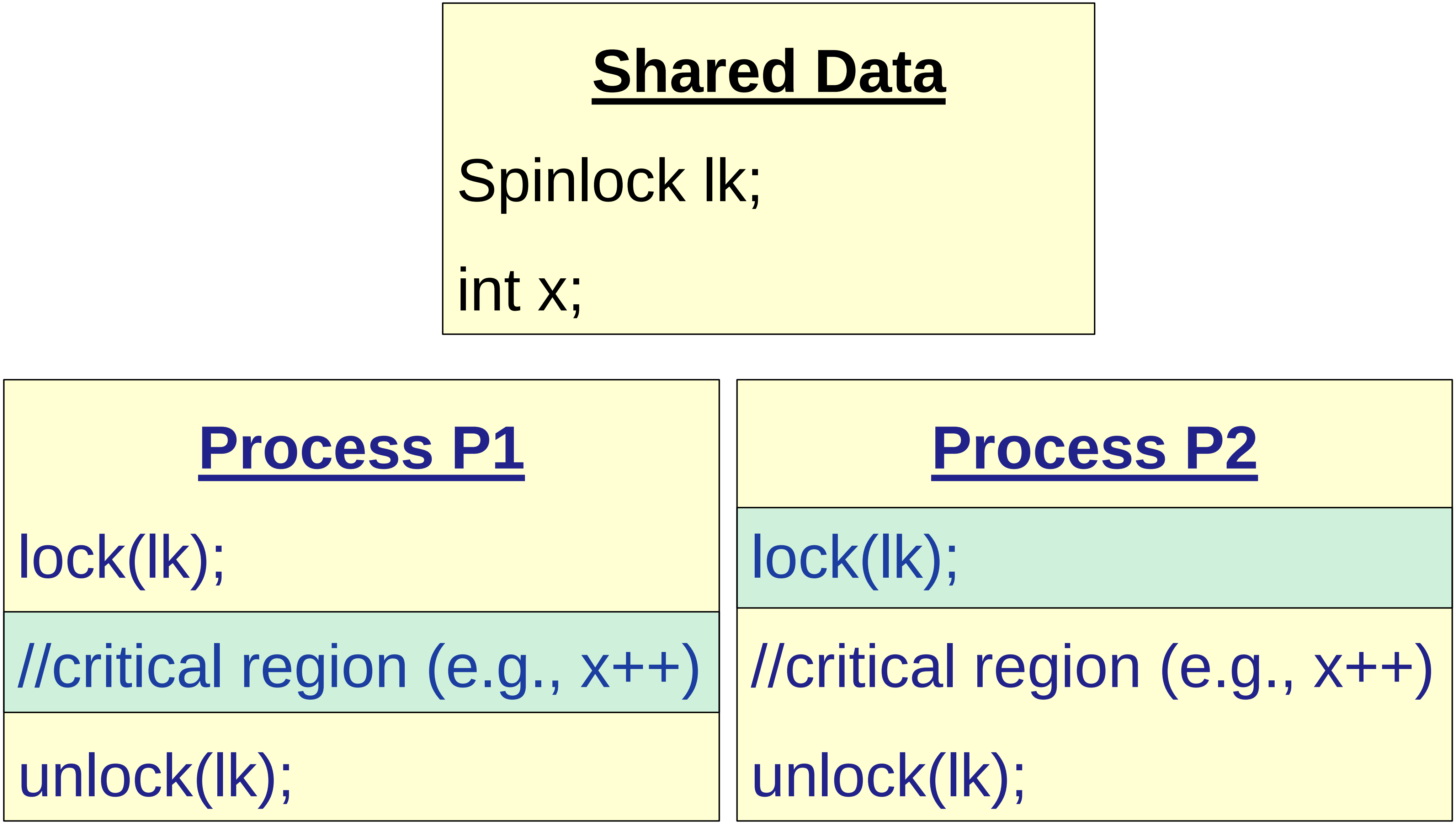
- Solution: use hardware
- Several hardware methods
  - Test & set: test a variable and set it in one instruction
  - Atomic swap: switch register & memory in one instruction
  - Turn off interrupts: process won’t be switched out unless it asks to be suspended
- The first two methods can be implemented in user land
  - Why can’t we implement the third method in user land?

Busy Waiting

- A process that is trying to acquire a locked spinlock is running!
  - It continuously checks:
    - can I get the lock? No, lock is held by another process
    - can I get the lock? No, lock is held by another process
    - …
  - This continuous check is called **spinning** or **busy waiting**
  - But what is wrong with that?
    - Busy waiting wastes CPU cycles
    - on a single-core system it delays the process that is holding the lock from releasing it

Today's problem: Busy Waiting

While P1 is in the critical region, P2 is busy waiting



Shared Data

```
Spinlock lk;
int x;
```

Process P1

```
lock(lk);
//critical region (e.g., x++)
unlock(lk);
```

Process P2

```
lock(lk);
//critical region (e.g., x++)
unlock(lk);
```

But why?

Why does busy waiting happen with spinlocks?

Atomic TestAndSet

- TestAndSet is an atomic instruction
- Works for single-core and multi-core Symmetric Multi-Processing (SMP)

```
int TestAndSet(int &x){
  lock memory access to x
  int temp = *x;
  *x = 1;
  unlock memory access to x
  return temp;
}
```

Spinlock implementation using TestAndSet

- Single **shared** variable: lock
- Works for any number of processes

```
int lock = 0;

Lock(){
   while (TestAndSet(&lock))
     ;
}

Unlock(){
   lock = 0;
}
```

Atomic Swap

- Swap is an atomic instruction
- Works for single-core and multi-core Symmetric Multi-Processing (SMP)

```
int Swap(int &x, int y){
  lock memory access to x
  int temp = *x;
  *x = y;
  unlock memory access to x
  return temp;
}
```

Spinlock implementation using Swap

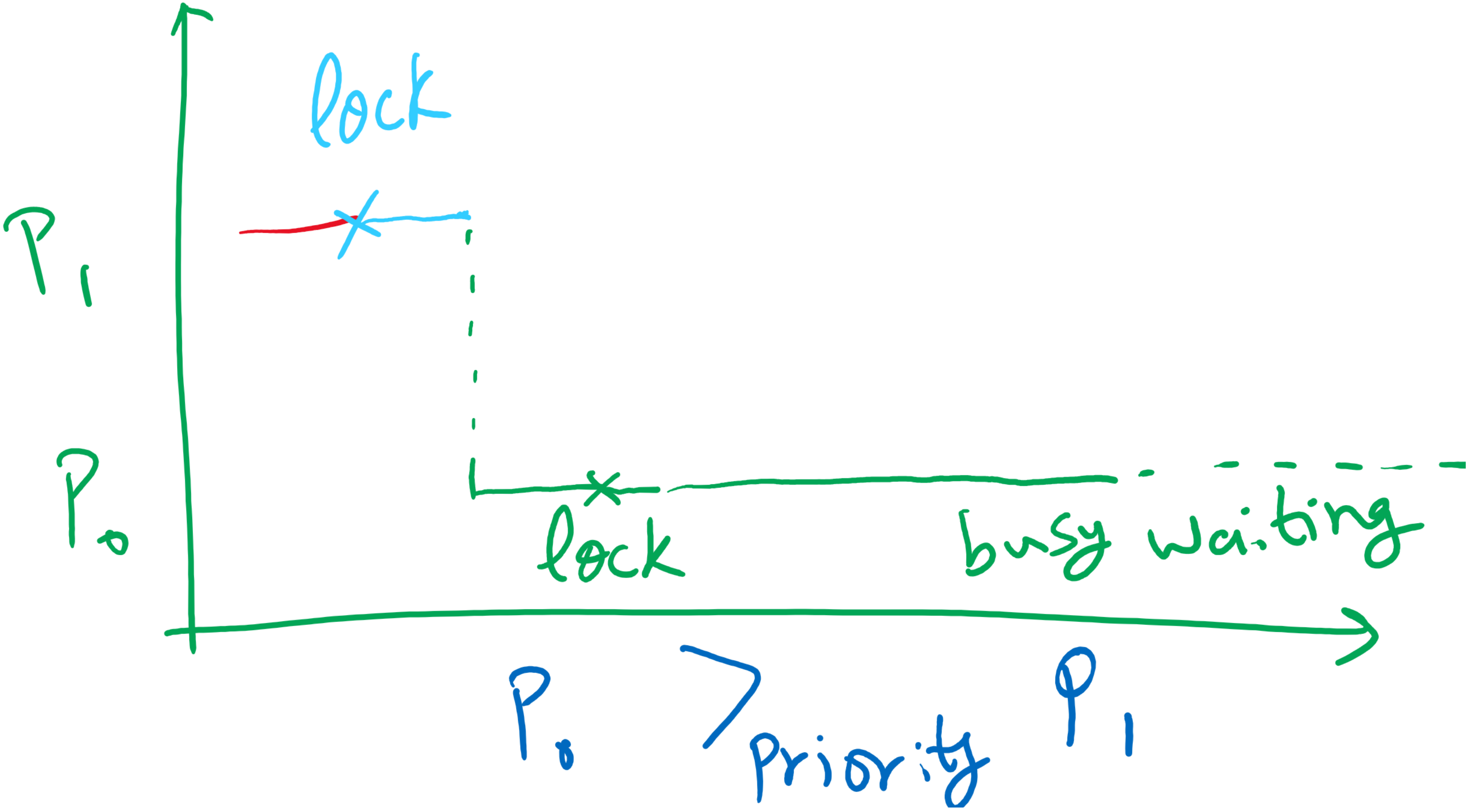
- Single **shared** variable: lock
- Works for any number of processes

```
int lock = 0;
```

```
Lock(){
   while (Swap(&lock, 1))
     ;
}

Unlock(){
   lock = 0;
}
```

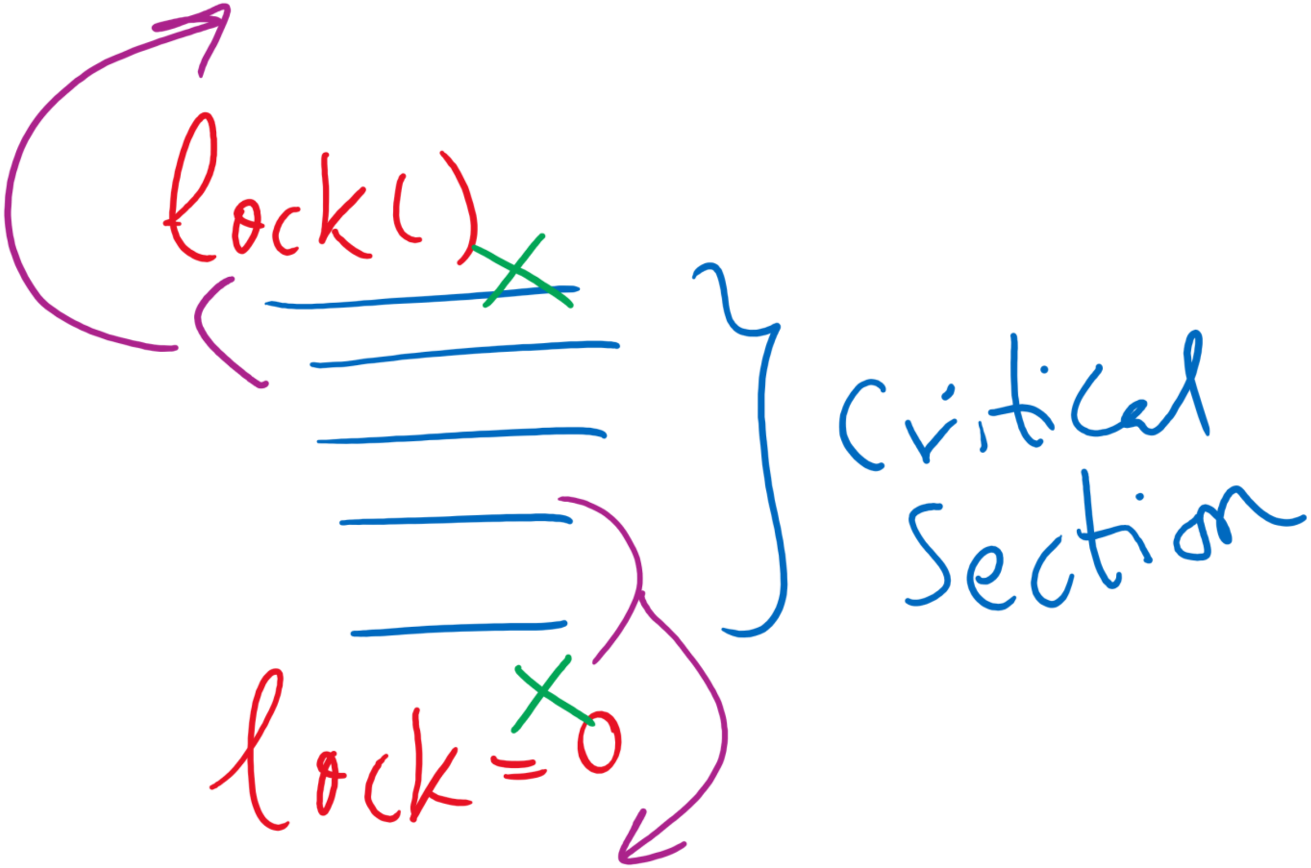
Implication of Busy Waiting

Subproblem: *priority inversion* (higher priority process busy waits for lower priority process)



Implementation Detail

compiler and/or hardware may reorder instructions


lock()
critical Section
lock = 0