University of Pittsburgh

Introduction to Operating Systems
CS 1550

Spring 2023

Sherif Khattab

ksm73@pitt.edu

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

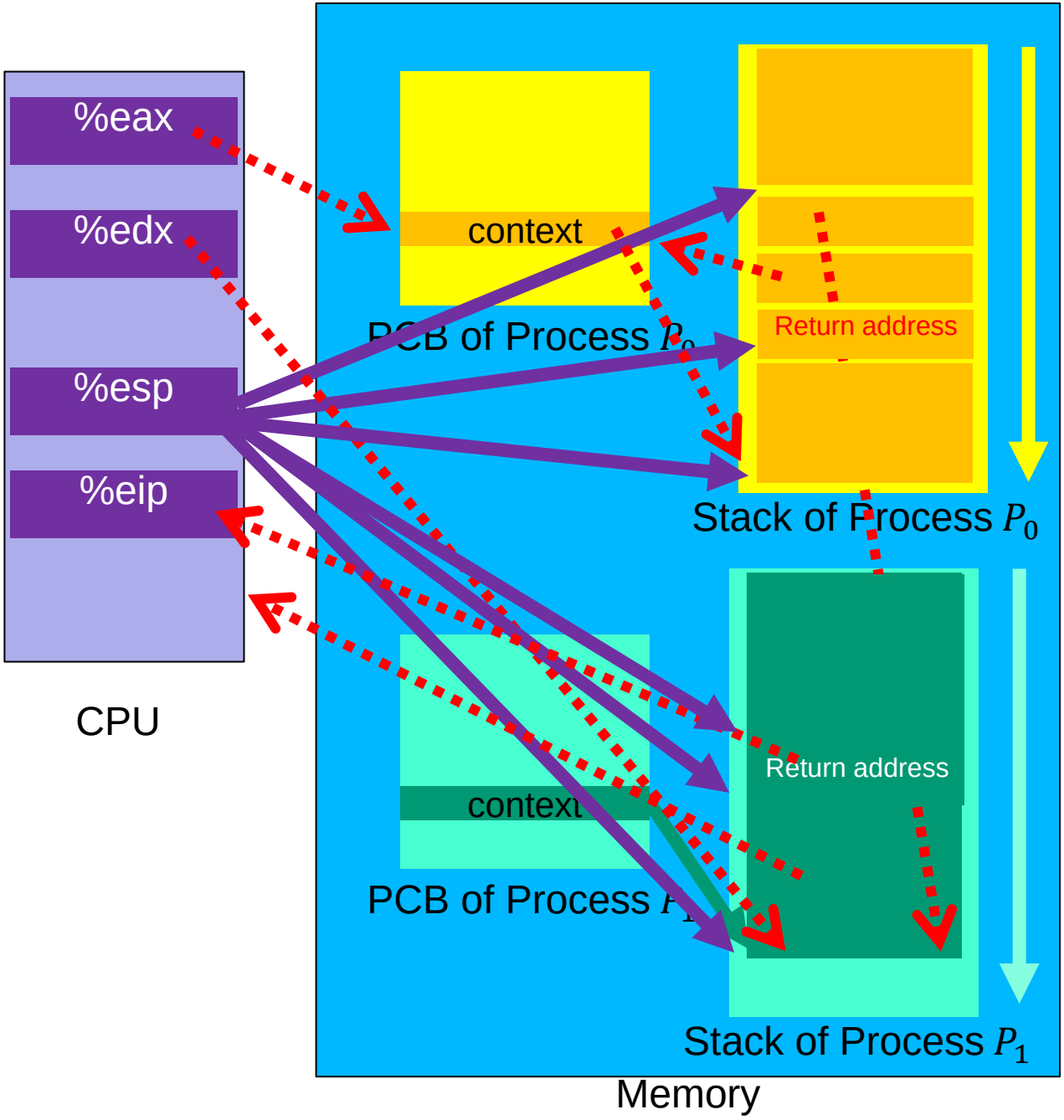
Announcements

- 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

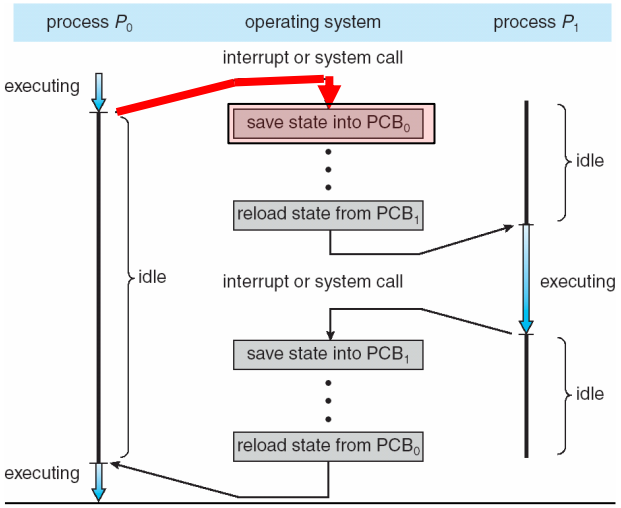
Agenda

- 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?
- How to solve it?
  - Sempahores

Context Switching in Xv6

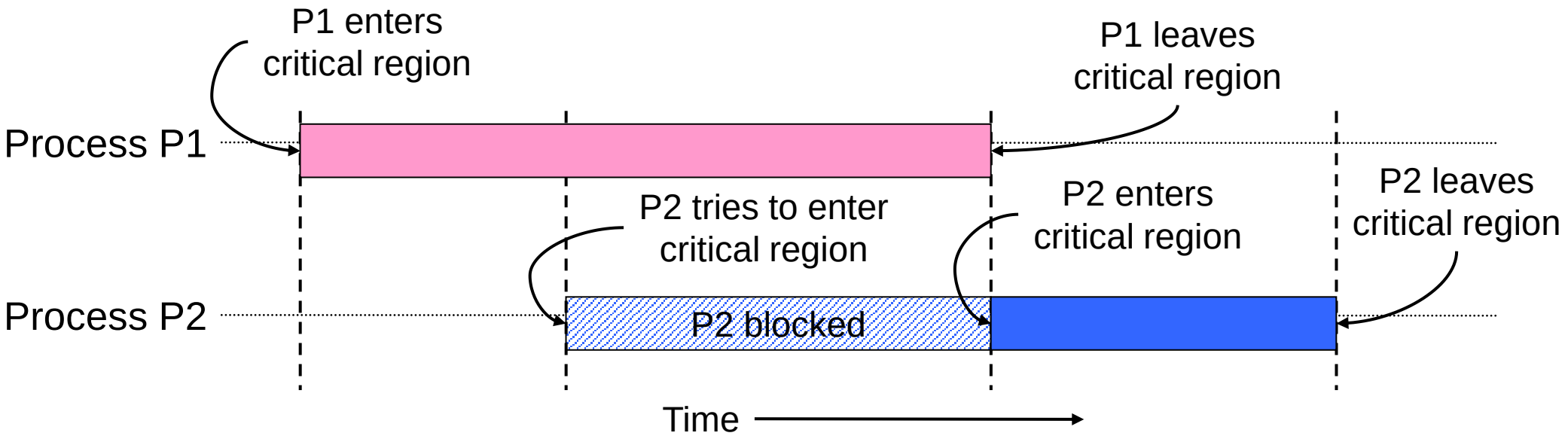



```
1  .globl swtch
2  swtch:
3    movl 4(%esp), %eax
4    movl 8(%esp), %edx
5
6    # Save old callee-saved registers
7    pushl %ebp
8    pushl %ebx
9    pushl %esi
10   pushl %edi
11
12   # Switch stacks
13   movl %esp, (%eax)
14   movl %edx, %esp
15
16   # Load new callee-saved registers
17   popl %edi
18   popl %esi
19   popl %ebx
20   popl %ebp
21   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;
}
```

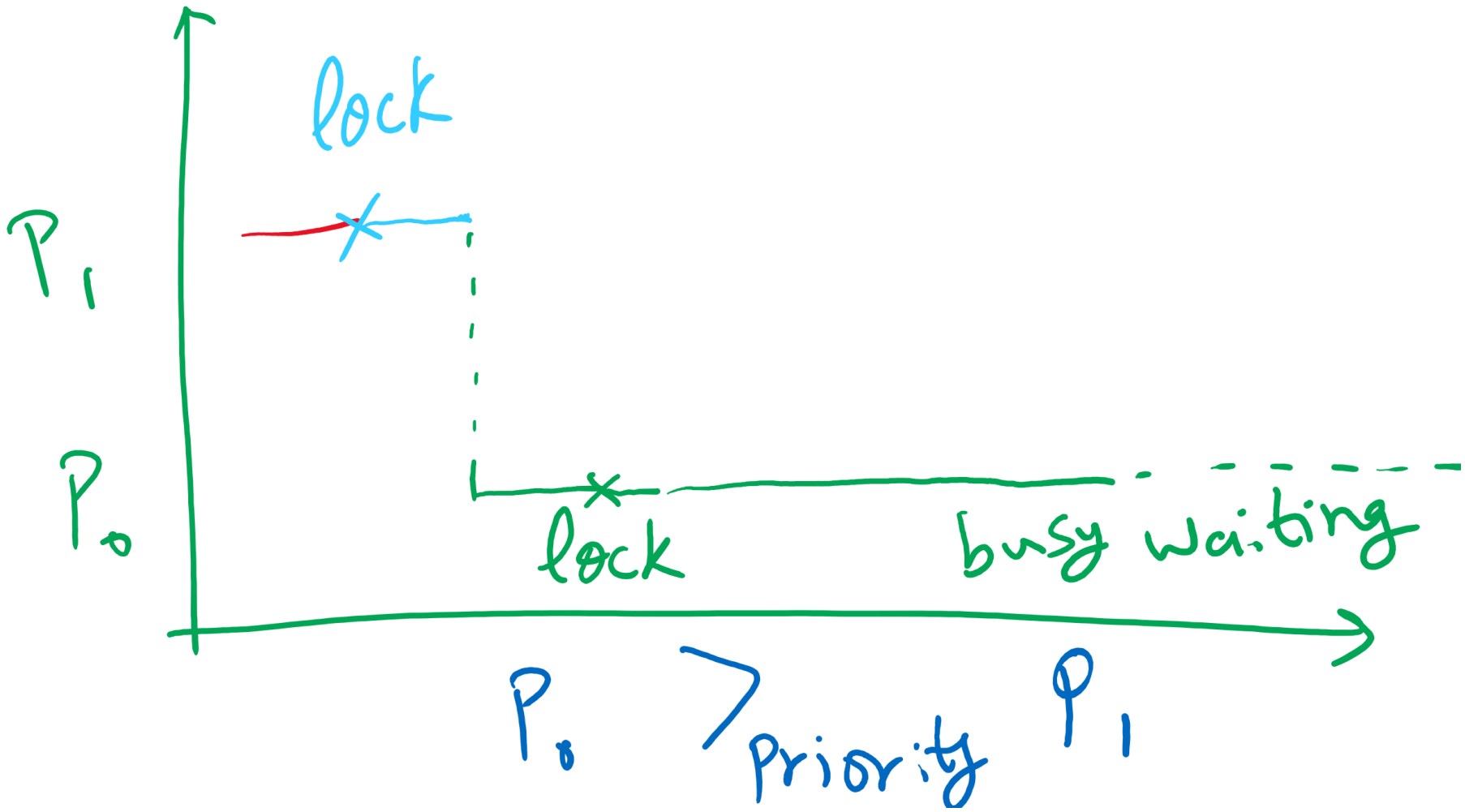
Spinlock implementation using Swap

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

```
int lock = 0;

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

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

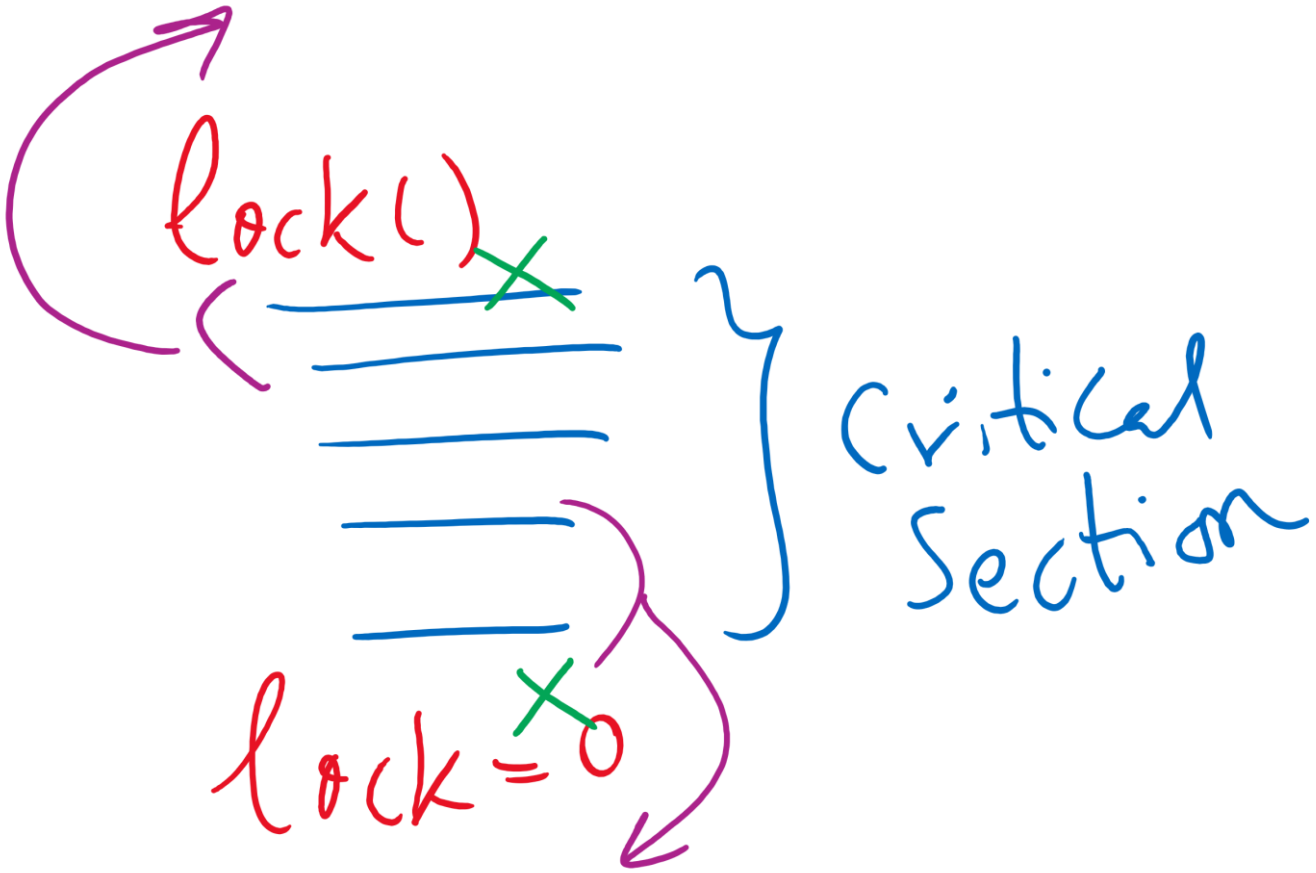
Implication of Busy Waiting

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



Implementation Detail

compiler and/or hardware may reorder instructions



Xv6 Walkthrough

- Spinlocks
  - __sync_synchronize() is a memory barrier instruction

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?


};
```

```
void
initlock(struct spinlock *lk, char *name)
{

  lk->locked = 0;

}
```

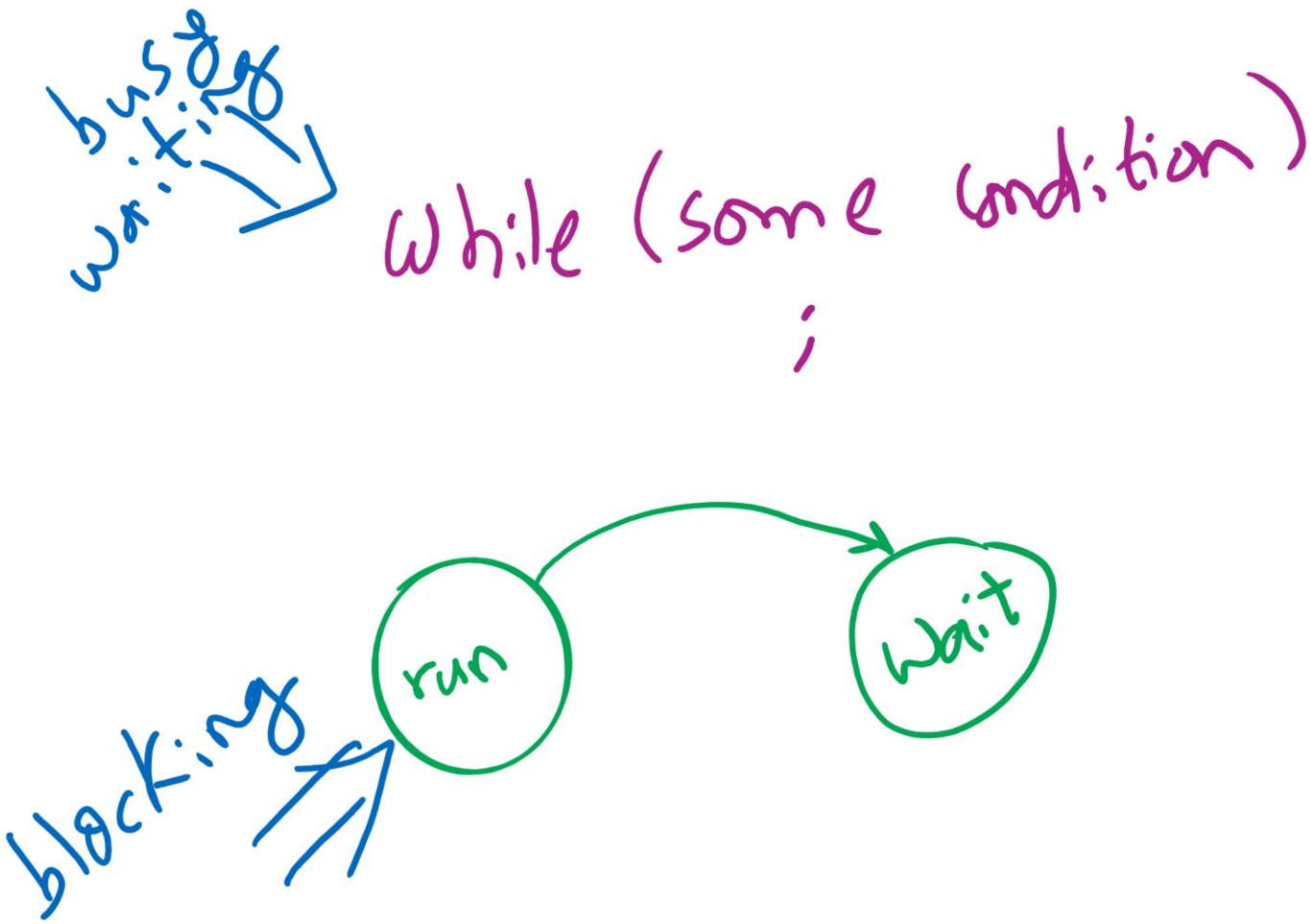
```
void
acquire(struct spinlock *lk)
{


  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();


}
```

```
void
release(struct spinlock *lk)
{

  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

}
```

Semaphores

- Solution: use semaphores
  - Synchronization mechanism that doesn't require busy waiting
- Implementation
  - Semaphore S accessed by two atomic operations
    - Down(S): decrement the semaphore if > 0; block otherwise
    - Up(S): increment the semaphore and wakeup one blocked process if any
  - Down() is another name for P()
  - Up() is another name for V()

Busy waiting vs. Blocking



Blocking involves 2 context switches

Critical sections using semaphores

Shared variables

```
Semaphore sem(1);
```

Code for process P_i

```
while (1) {
   // non-critical section
   down(sem);
   // critical section
   up(sem);
   // non-critical section
}
```

Types of semaphores

- Two different types of semaphores
  - Counting semaphores
  - Binary semaphores
- Counting semaphore
  - Value can range over an unrestricted range
- Binary semaphore
  - Only two values possible
    - 1 means the semaphore is available
    - 0 means a process has acquired the semaphore
  - May be simpler to implement
- Possible to implement one type using the other

Semaphore Implementation

But how do semaphores avoid busy waiting?

Implementing semaphores with blocking

- Assume two operations:
  - Sleep(): suspends current process
  - Wakeup(P): allows process P to resume execution
- Semaphore data structure
  - Tracks value of semaphore
  - Keeps a list of processes waiting for the semaphore

```
struct Semaphore {
  int value;
  ProcessList pl;
};
```

```
down ()
{
  value -= 1;
  if (value < 0) {
    // add this process to pl
    Sleep ();
  }
}

up () {
  Process P;
  value += 1;
  if (value <= 0) {
    // remove a process P
    // from pl
    Wakeup (P);
  }
}
```

How to protect these shared variables??

Spinlocks in Semaphores

```
struct Semaphore {
  int value;
  ProcessList pl;


};
```

```
down ()
{

  value -= 1;
  if (value < 0) {
    // add this process to pl
    Sleep ();
  }



}

up () {
  Process P;
  value += 1;
  if (value <= 0) {
    // remove a process P
    // from pl
    Wakeup (P);
  }
}
```

Spinlocks are sometimes better than Semaphores

- Very (very) short waiting time to enter the critical section < the 2 context switches needed for blocking
  - Multi-core
    - so that the spinlock can be unlocked while the process is busy waiting
  - Few contending processes for the critical section
  - Short critical section code