Introduction to Operating Systems
CS 1550

Spring 2023

Sherif Khattab

ksm73@pitt.edu

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

Announcements

- Upcoming deadlines
  - Homework 2 is due this Friday at 11:59 pm
  - Lab 1 is due on Tuesday 2/7 at 11:59 pm
  - Project 1 is due on Friday 2/17 at 11:59 pm
- Student Support Hours available on the syllabus page

Previous Lecture …

- Critical Region
- Using Spinlock to implement Critical Regions
  - Busy Waiting problem

Xv6 Walkthrough of Spinlock Implementation

__sync_synchronize() is a memory barrier instruction

```
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

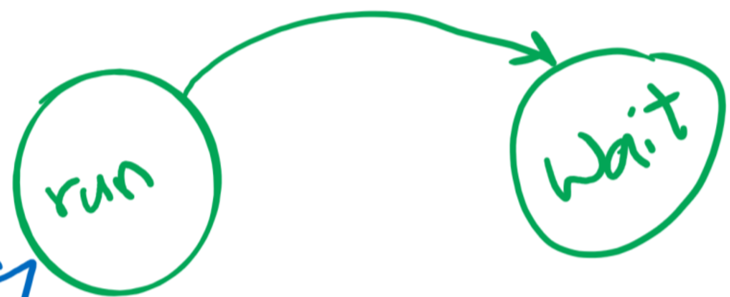
- Solution to the Busy Waiting problem: use semaphores
  - Semaphores are Synchronization mechanism that doesn’t require busy waiting
- Implementation
  - Semaphore S accessed by two atomic operations
    - Down(S): decrement the semaphore if > 0; *block* otherwise
    - Up(S): increment the semaphore and *wakeup* one blocked process if any
  - Down() is another name for P()
    - also called wait
  - Up() is another name for V()
    - also called signal

Busy waiting vs. Blocking

busy waiting

```
while (some condition)
    ;
```

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
}
```

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;



};
```

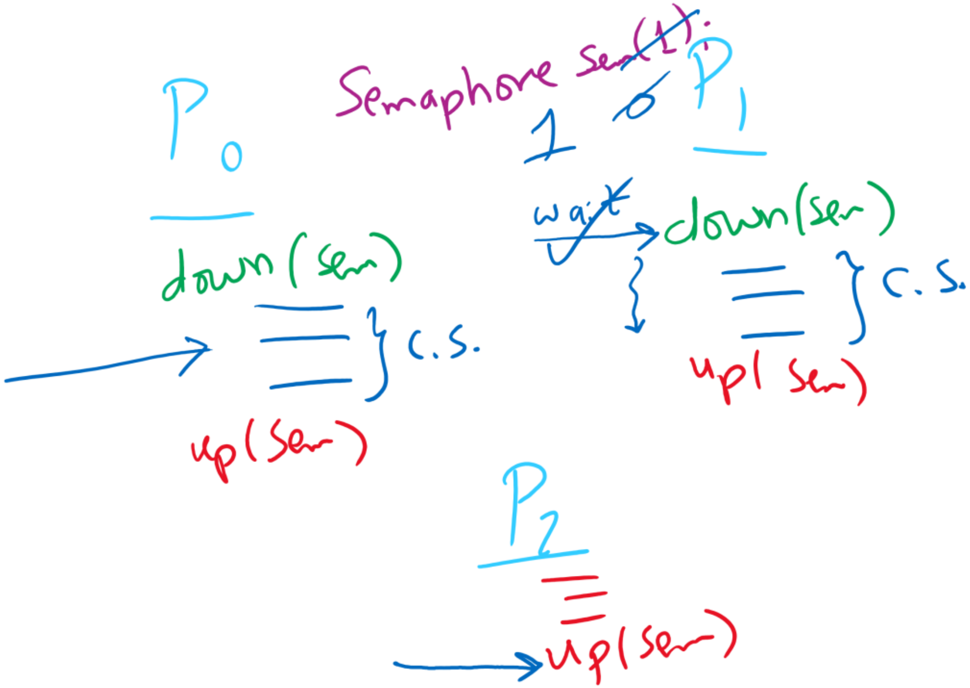
```
down ()
{


  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

Semaphore Usage Problem: Compromising Mutual Exclusion

- Any process can up() the semaphore



- Solution: A **Mutex** can be up()'d only by the same process that down()'d it

Semaphore Usage Problem: Deadlock and Starvation

- Deadlock: two or more processes are waiting indefinitely for an event that can only be caused by a waiting process
  - P0 gets A, needs B
  - P1 gets B, needs A
  - Each process waiting for the other to signal
- Starvation: indefinite blocking
  - Process is never removed from the semaphore queue in which its suspended
  - May be caused by ordering in queues (priority)

Shared variables

```
Semaphore A(1),B(1);
```

Process P_0

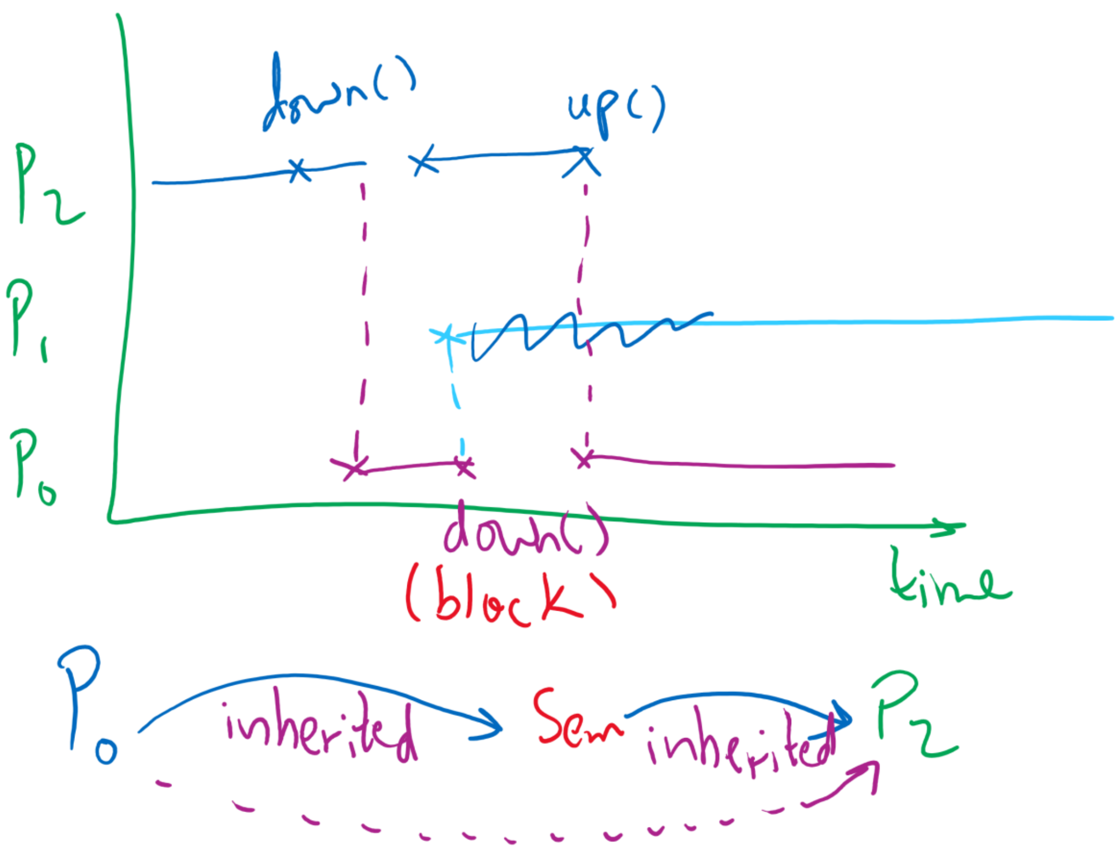
```
A.down();
B.down();
    .
    .
    .
B.up();
A.up();
```

Process P_1

```
B.down();
A.down();
    .
    .
    .
A.up();
B.up();
```

Semaphore Usage Problem: Priority Inversion

- Priority inversion is still possible using semaphores
  - Slightly less likely
  - Needs at least three processes



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

Question

How are processes created, maintained, and terminated?

Process Lifecycle (AKA Process States)