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 5 is due **this Friday**
  - Project 1 is due **this Friday** at 11:59 pm
  - Lab 2 is due on Tuesday 2/28 at 11:59 pm
  - Project 2 will be posted this Friday

Previous lecture …

- Deadlock detection and avoidance using the Banker’s algorithm
- Sleepy Barbers problem

Problem of the Day: Sleepy Barbers

- We have two sets of processes
  - Worker processes (e.g., barbers)
  - Customer processes
- Customer processes may arrive at anytime
- Worker processes check in when they are not serving any customers
- Each worker process must **wait** until it gets matched with a customer process
- Each customer process must **wait** until it gets matched with a worker process
- The customer process cannot leave until the matched worker process finishes the work
- The worker process cannot check in for the next customer until the matched customer process leaves

Rendezvous Pattern

Sleepy Barber

Worker

Customer

~~arrive~~ checks

arrives

Rendez-vous

do work

get served

Rendez-vous

checks for next customer

leaves

Solution Using Semaphores: Take 1

- One pair of semaphores per rendezvous
  - RV1a and RV1b
  - RV2a and RV2b
- Notice the flipped order of the down and up calls in the two processes

Worker

Semaphore RV1a, RV1b
(0) (0)
RV2a, RV2b
(0) (0)

Customer

arrives/checks in
down(RV1a)
up(RV1b)
does work
up(RV2a)
down(RV2b)

arrives
up(RV1a)
down(RV1b)
gets served
down(RV2a)
up(RV2b)

Solution Using Semaphores: Take 1

- This solution doesn't work for multiple workers and multiple customers
    - In that case, a customer can leave before its associated worker finishes

Sleepy Barbers Solution: Take 2

```
struct mysems {
   Semaphore RV1a(0), RV1b(0), RV2a(0), RV2b(0);
};
SharedBuffer buff; //From producers-consumers problem
```

Worker Process	**Customer Process**
struct mysems sems = buff.consume();	struct mysems sems = new struct mysems
up(sems.RV1a);	buff.produce(sems);
down(sems.RV1b);	down(sems.RV1a);
//do work	up(sems.RV1b);
down(sems.RV2a);	//get work
up(sems.RV2b);	up(sems.RV2a);
//check-in for next customer	down(sems.RV2b);
	//leave

Solution using Mutex and Condition Variables

- https://cs1550-2214.github.io/cs1550-code-handouts/ProcessSynchronization/Slides/

How to implement Condition Variables?

- How to implement condition variables?
- Reflect more on all the solutions/problems that we have studied

User-level implementation of Condition Variables

A Lock with two waiting queues

```
struct Lock {
  Semaphore mutex(1);
   Semaphore next(0);
  int nextCount = 0;
}
```

Acquire(){

```
  mutex.down();
}
```

Release(){

```
   if(nextCount > 0){
      next.up();
      nextCount--;
  } else mutex.up();
}
```

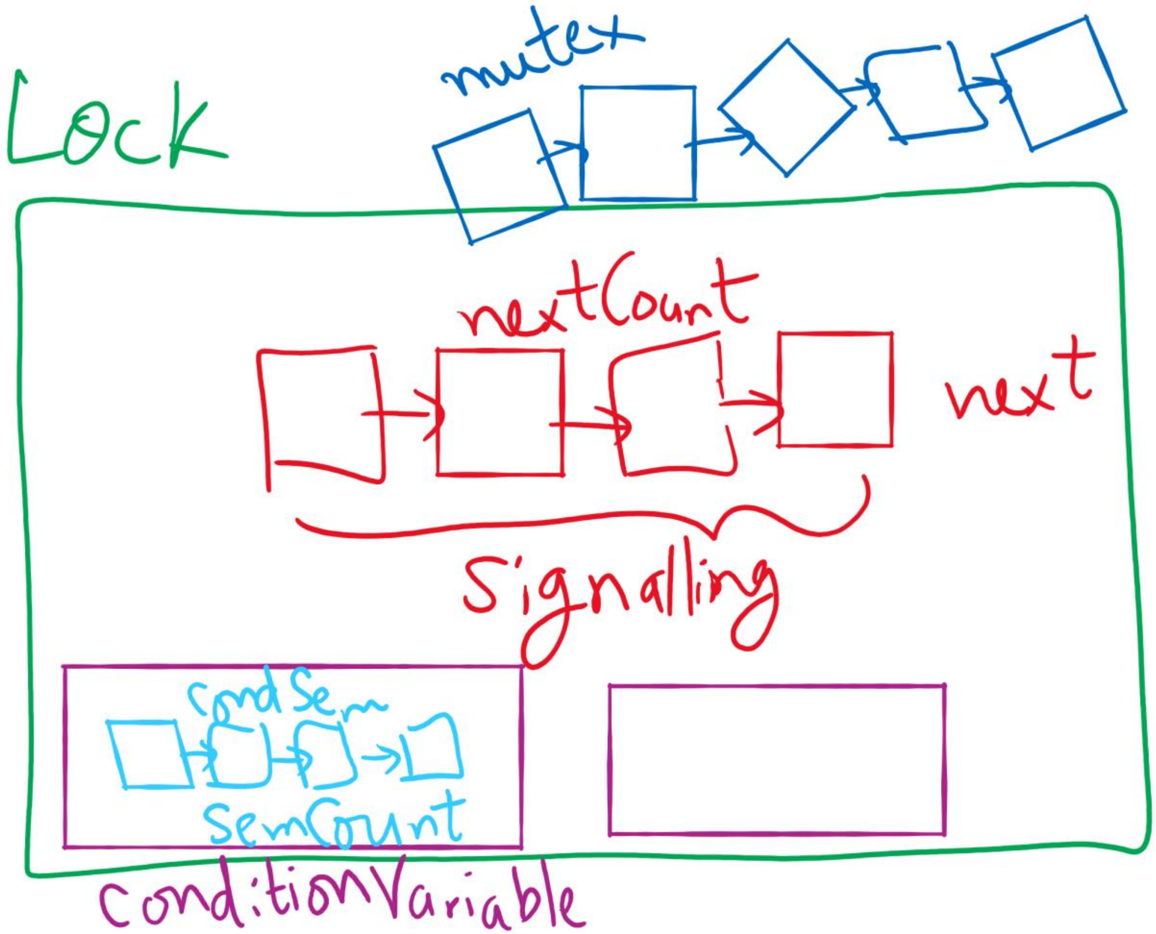
Condition Variable

```
struct ConditionVariable {
  Semaphore condSem(0);
   int semCount = 0;
  Lock *lk;
}
```

```
Wait(){
  if(lk->nextCount > 0)
    lk->next.up();
    lk->nextCount--;
  else {
    lk->mutex.up();
  }
  semCount++;
  condSem.down();
   semCount--;
}
```

```
Signal(){
  if(semCount > 0){
    condSem.up()
    lk->nextCount++
    lk->next.down();
    lk->nextCount—
  }
}
```

Lock and Condition Variable Implementation



Implementing locks with semaphores

- Use mutex to ensure exclusion within the lock bounds
- Use next to give lock to processes with a higher priority (why?)
- nextCount indicates whether there are any higher priority waiters

```
class Lock {
  Semaphore mutex(1);
  Semaphore next(0);
  int nextCount = 0;
};
```

```
Lock::Acquire()
{
  mutex.down();
}
```

```
Lock::Release()
{
  if (nextCount > 0)
    next.up();
  else
    mutex.up();
}
```

Implementing condition variables

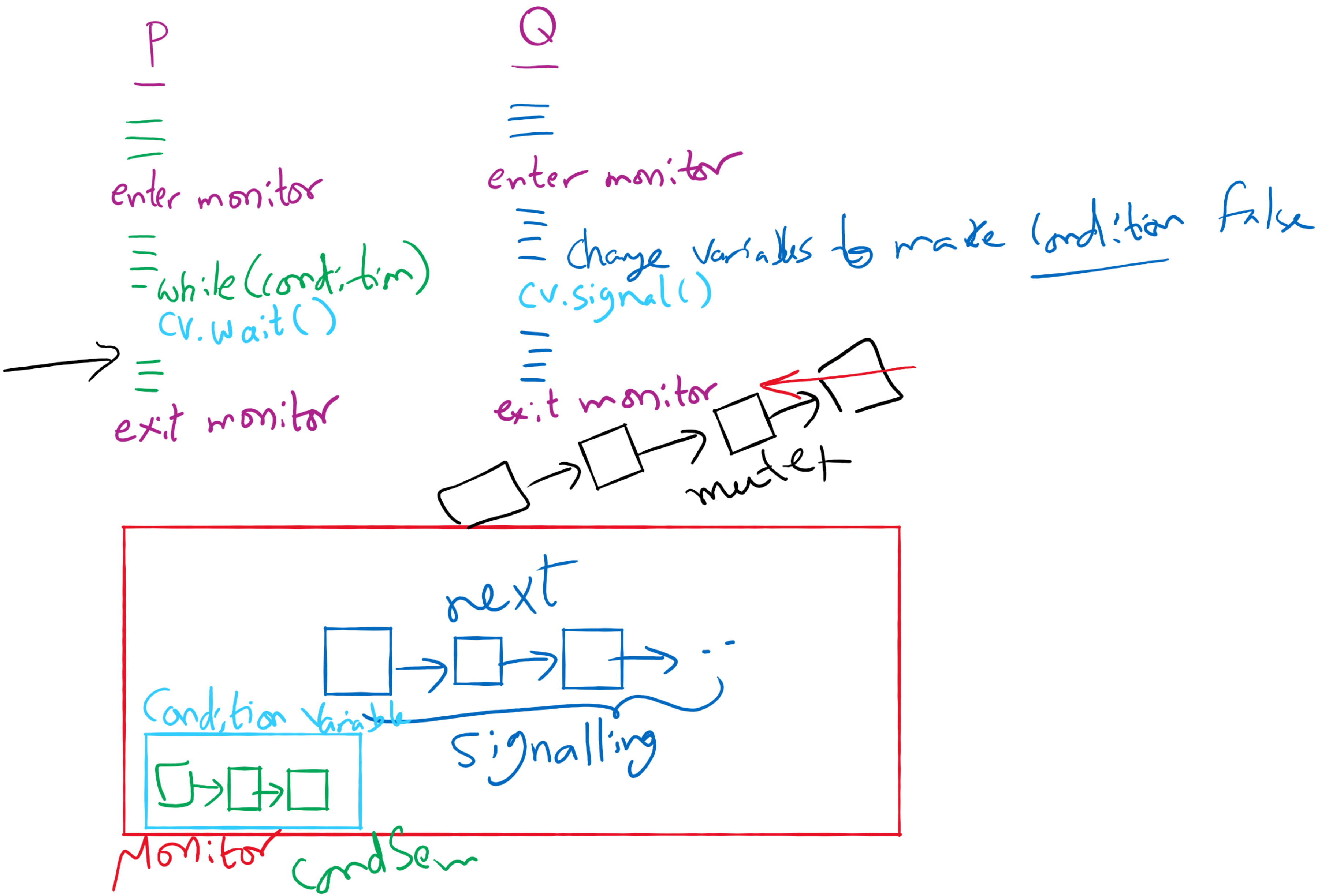
- Are these Hoare or Mesa semantics?
- Can there be multiple condition variables for a single Lock?

```
class Condition {
  Lock *lock;
  Semaphore condSem(0);
  int semCount = 0;
};
```

```
Condition::Wait ()
{
  semCount += 1;
  if (lock->nextCount > 0)
    lock->next.up();
  else
    lock->mutex.up();
  condSem.down ();
  semCount -= 1;
}
```

```
Condition::Signal ()
{
  if (semCount > 0) {
    lock->nextCount += 1;
    condSem.up ();
    lock->next.down ();
    lock->nextCount -= 1;
  }
}
```

Process Synchronization inside Monitors



Reflections on semaphore usage

- Semaphores can be used as
  - Resource counters
  - Waiting spaces
  - For mutual exclusion

Reflections on Condition Variables

- Define a class and put all shared variables inside the class
- Include a mutex and a condition variable in the class
- For each public method of the class
  - Start by locking the mutex lock
  - If need to wait, use a while loop and wait on the condition variable
  - Before **broadcasting** on the condition variable, make sure to change the waiting condition