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

- Upcoming deadlines
  - Homework 5 is due **this Friday**
  - Project 1 is due on Friday 2/17 at 11:59 pm
  - Lab 2 is due on Tuesday 2/28 at 11:59 pm

Previous lecture …

- Dining philosophers
- Deadlock prevention
- Banker's Algorithm for deadlock detection and avoidance

Banker's Algorithm

How to detect deadlocks?

How to avoid deadlocks?

Banker's Algorithm

We can use the same algorithm for both detecting and avoiding deadlocks

Banker's Algorithm

	A	B	C	D
Avail	2	3	0	1

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
1	3	2	1	0
2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```
current=avail;
for (j = 0; j < N; j++) {
 for (k=0; k<N; k++) {
  if (finished[k])
   continue;
  if (want[k] <= current) {
   finished[k] = 1;
   current += hold[k];
   break;
  }
 }
 if (k==N) {
   printf "Deadlock!\n";
   // finished[k]==0 means process is in
   // the deadlock
   break;
 }
}
```

Note: want[j], hold[j], current, avail are arrays!

Banker's Algorithm Insights

- It is possible that some event sequences lead to a deadlock
- What we are looking for is **at least one** event sequence that can make all processes finish
  - If such sequence exists, the state is safe
  - The Banker's algorithm finds such sequence if it exists

Using the Banker's Algorithm for Deadlock Avoidance

- Call the algorithm on the following ``What-if" state instead of the current state

avoiding deadlocks

Request from Process i

avail` = avail - Request

hold`[i] = hold[i] + Request

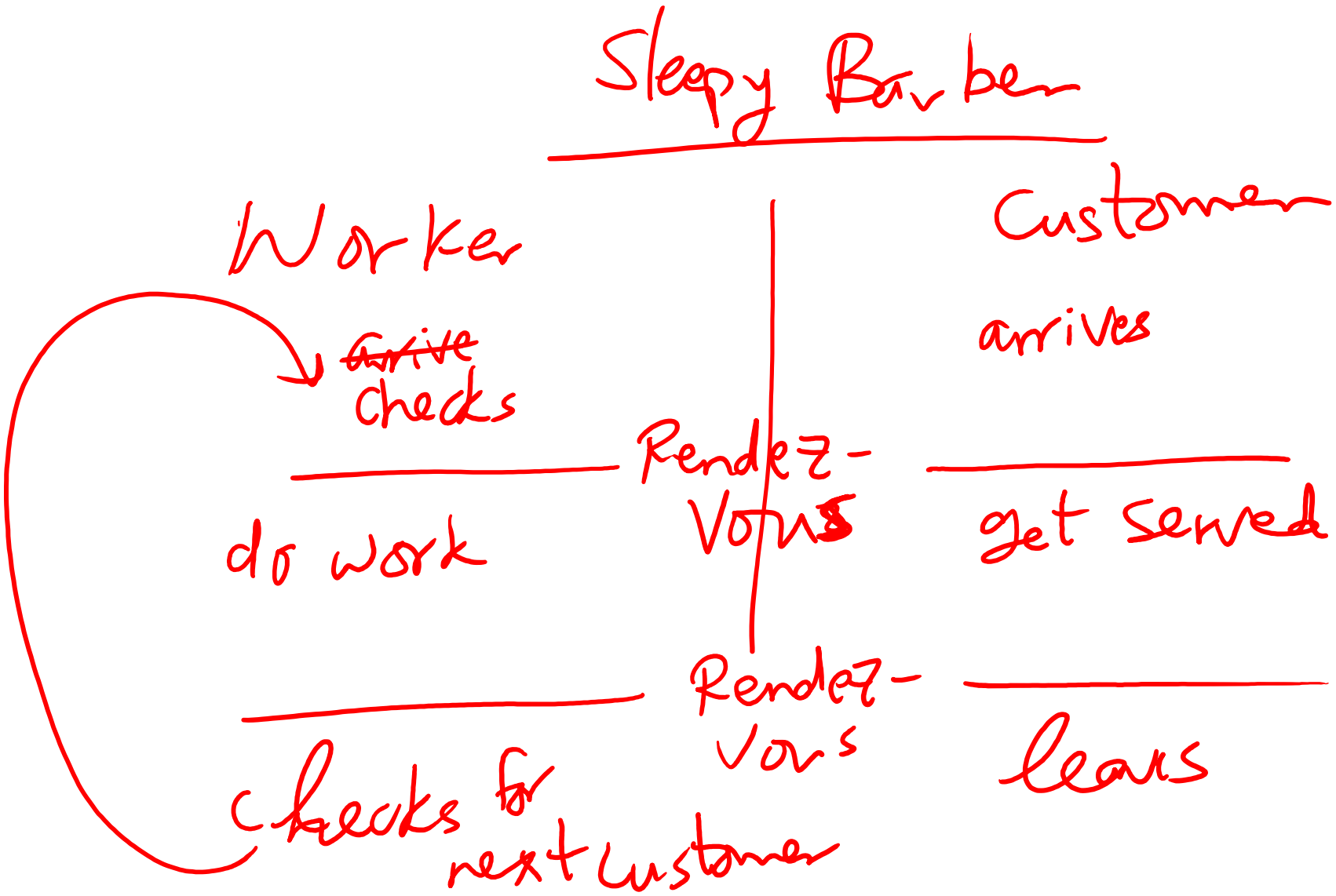
Want`[i] = want[i] - Request

Run algo on avail`, hold`, Want`

Problem of the Day: Sleepy Barbers

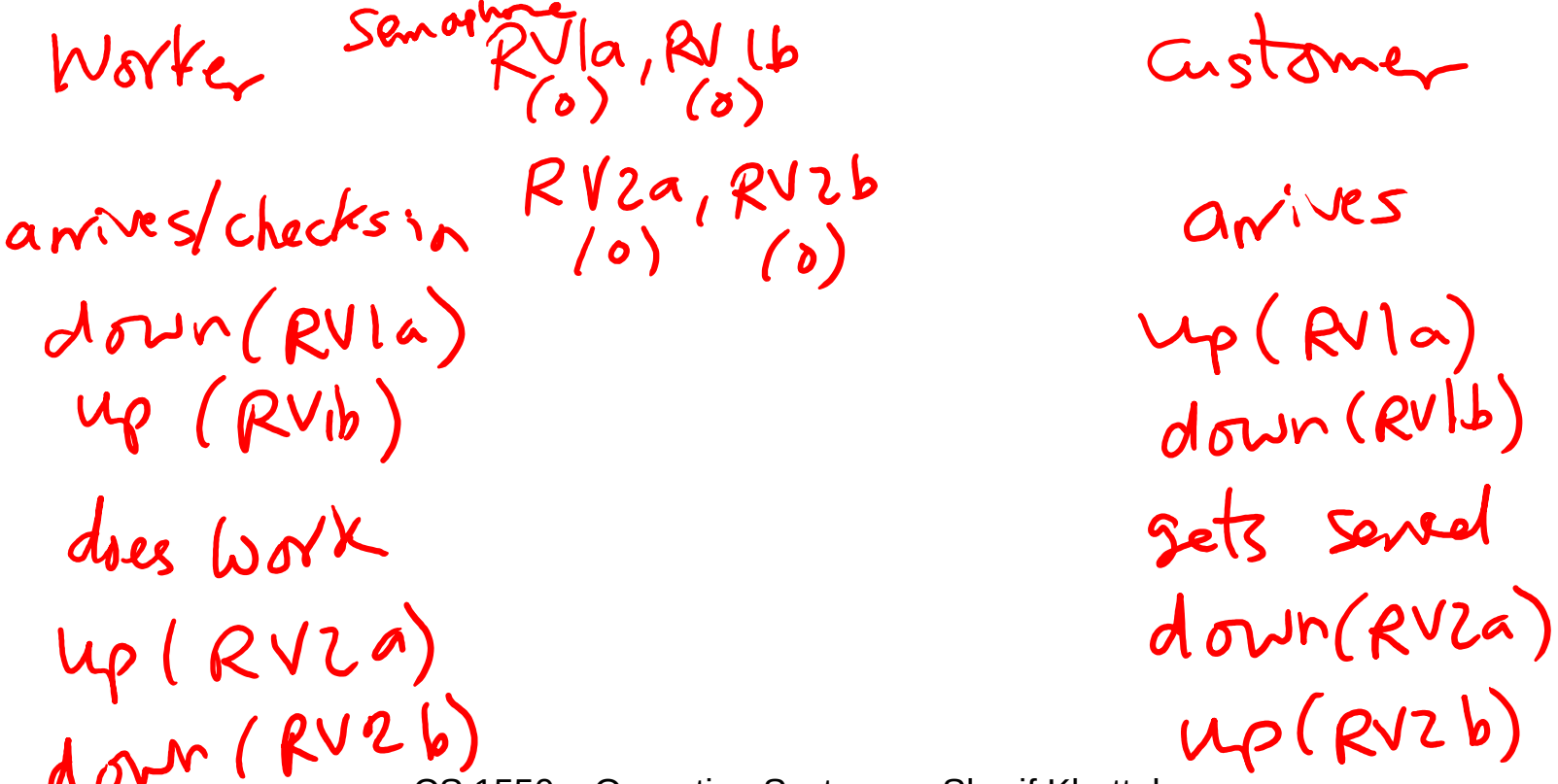
- We have two sets of processes
  - Worker processes (e.g., barbers)
  - Customer processes
- Customer processes may arrive at anytime
- Worker processes check in when they are not serving any customers
- Each worker process must wait until it gets matched with a customer process
- Each customer process 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
- Many applications in the real-world

Rendezvous Pattern



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



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

```
struct mysems sems = buff.consume();
up(sems.RV1a);
down(sems.RV1b);
//do work
down(sems.RV2a);
up(sems.RV2b);
//check-in for next customer
```

Customer Process

```
struct mysems sems = new struct mysems
buff.produce(sems);
down(sems.RV1a);
up(sems.RV1b);
//get work
up(sems.RV2a);
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();
}
```

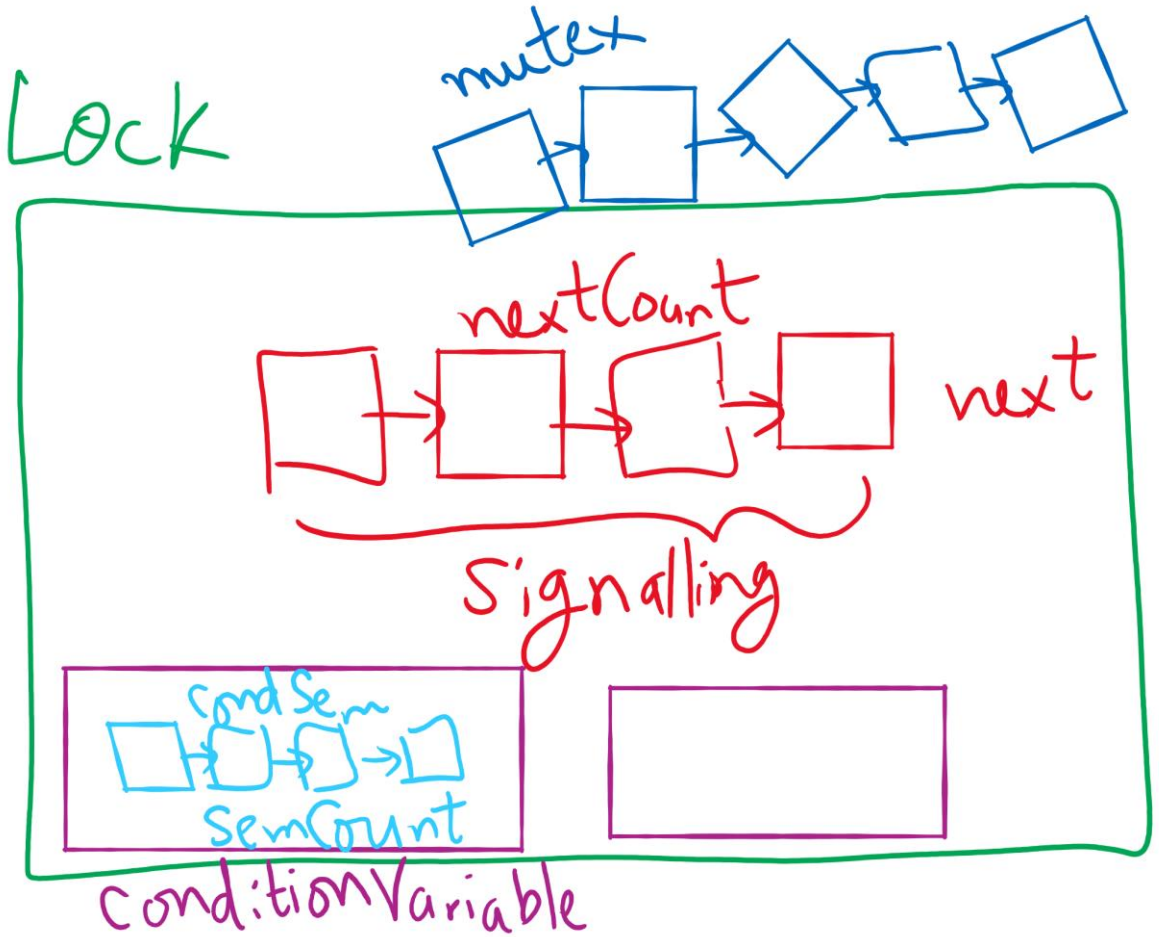
Condition Variable

```
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

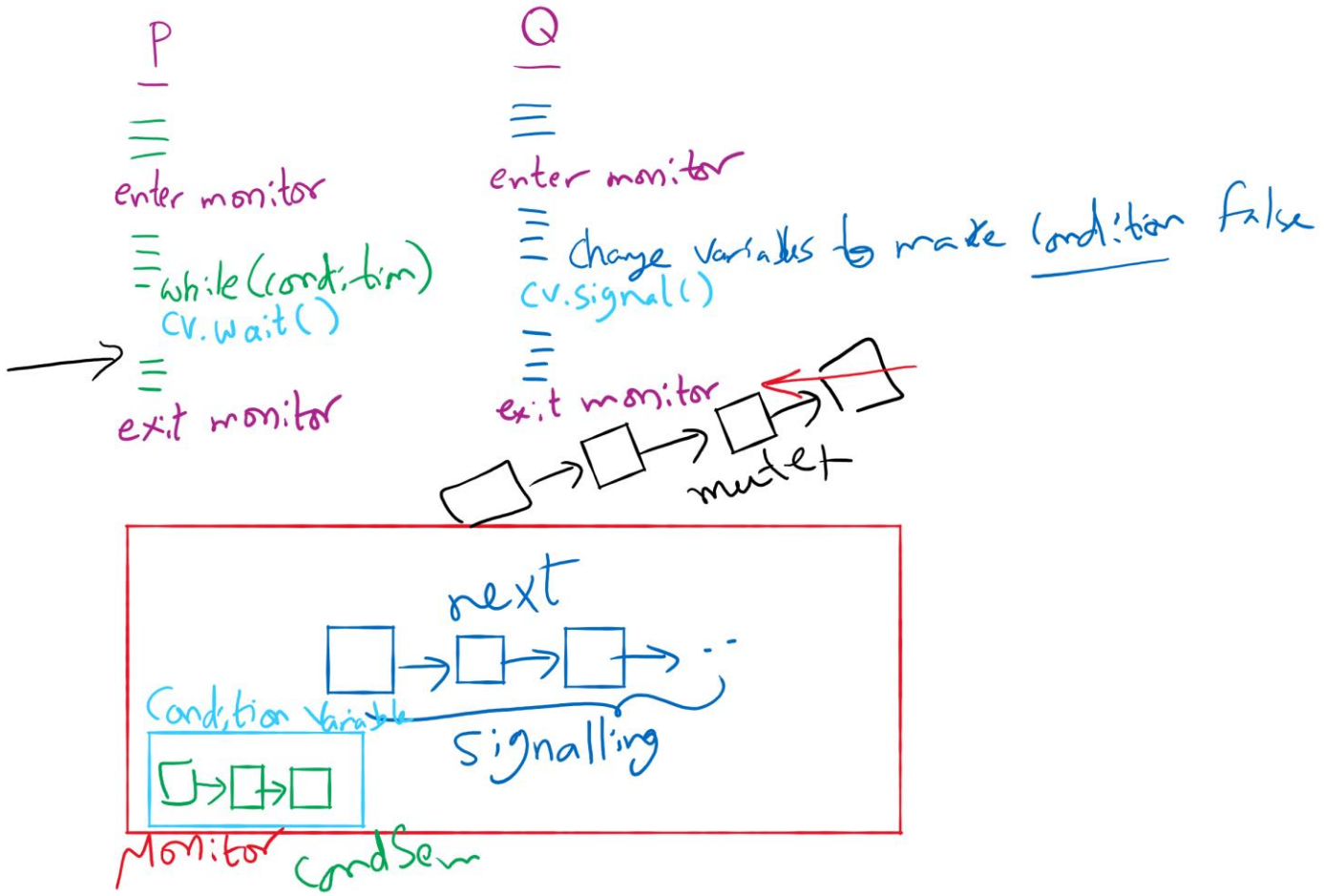
- Are these Hoare or Mesa semantics?
- Can there be multiple condition variables for a single Lock?

```
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



Condition Variable-based Solutions

- Code Walkthrough at:

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

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