Parallel Programming
Exercise Session 10

Outline

1. Feedback: Assignment 9
2. Lecture Recap: Semaphores
3. Lecture Recap: Monitors
4. Lecture Recap: Conditional locks
5. Assignment 10

Feedback: Assignment 9

Recap: Critical Section Properties

- **Mutual exclusion**: No more then one process executing in the critical section
- **Progress**: When no process is in the critical section, any process that requests entry must be permitted without delay
- **No starvation (bounded wait)**: If any process tries to enter its critical section then that process must eventually succeed.

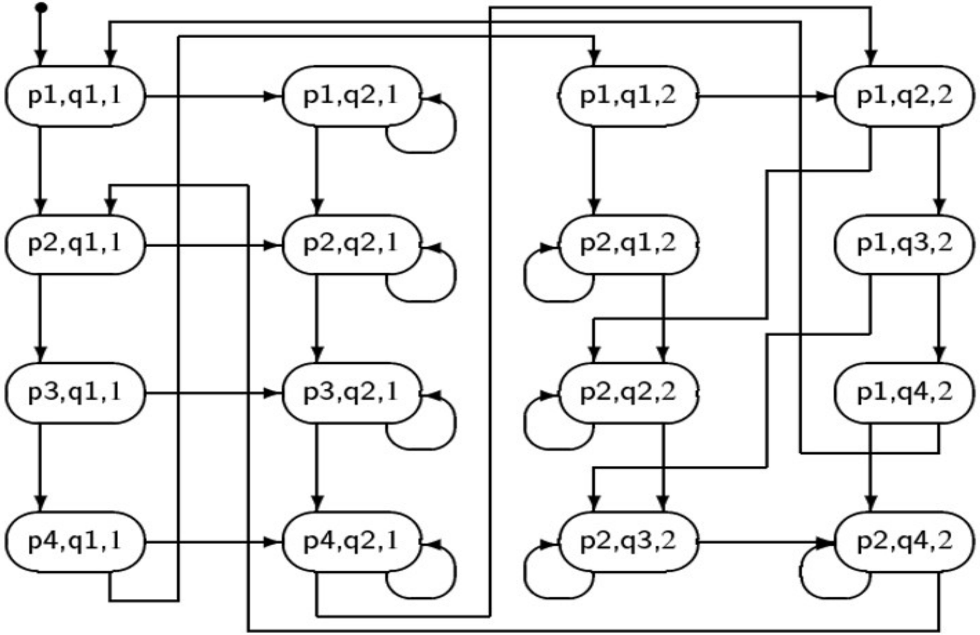
P

```
p1:     Non-critical section P
p2:     while turn != 1
p3:     Critical section
p4:     turn = 2
```

turn = 1

Q

```
q1:     Non-critical section Q
q2:     while turn != 2
q3:     Critical section
q4:     turn = 1
```



- **Mutual exclusion**: E.g. State (p3,q3,_) is not reachable
- **Progress**: E.g. There exists a path for P such that state (P3, _ , _) is reachable from (P2,_,_). Typical counterexamples: deadlocks and livelocks
- **No starvation (bounded wait)**: Possible starvation reveals itself as cycles in the state diagram.

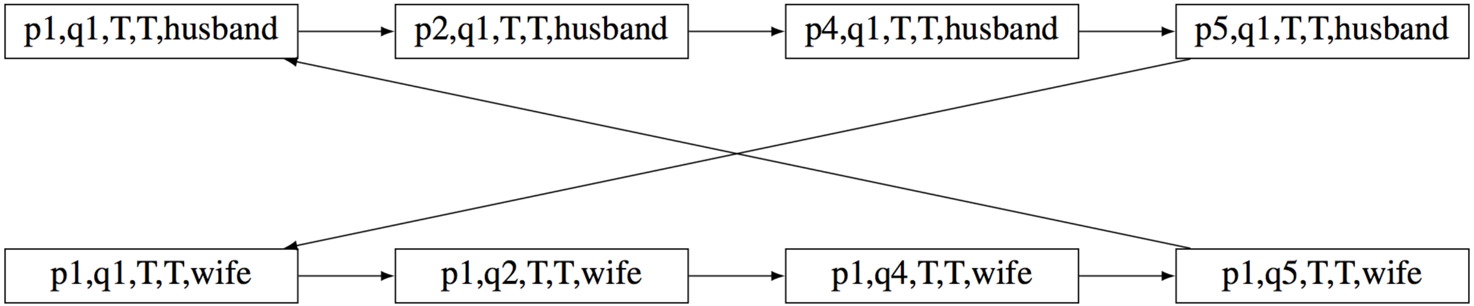
P	
p1:	Non-critical section P
p2:	while turn != 1
p3:	Critical section
p4:	turn = 2

turn = 1

Q	
q1:	Non-critical section Q
q2:	while turn != 2
q3:	Critical section
q4:	turn = 1

Feedback for Assignment 9

owner			
husband.hungry = true			
wife.hungry = true			
husband		wife	
p1:	while hungry	q1:	while hungry
p2:	owner != me	q2:	owner != me
p3:	sleep	q3:	sleep
p4:	spouse == hungry	q4:	spouse == hungry
p5:	owner = spouse	q5:	owner = spouse
p6:	CR	q6:	CR
p7:	hungry = false	q7:	hungry = false
p8:	owner = spouse	q8:	owner = spouse



Feedback for Assignment 9

- One way to solve the livelock problem is to impose an ordering when acquiring the lock on the shared resource.

- Or one of the spouses can actually take the spoon after certain number of retries

Feedback for Assignment 9

Optimistic vs Pessimistic concurrency control

```java
@Override
public int nextInt() {
    // get the current seed value
    long next;
    synchronized (this) {
        long orig = state;
        // using recurrence equation to generate next
        next = (a * orig + c) & (~0L >>> 16);
        // store the updated seed
        state = next;
    }
    return (int) (next >>> 16);
}
```

```java
@Override
public int nextInt() {
    while (true) {
        // get the current seed value
        long orig = state.get();
        // using recurrence equation to generate next seed
        long next = (a * orig + c) & (~0L >>> 16);
        // store the updated seed
        if (state.compareAndSet(orig, next)) {
            return (int) (next >>> 16);
        }else{
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {

            }
        }
    }
}
```

Lecture Recap

Lecture Recap: Bakery Lock

Filter Lock is not fair.
The Bakery Lock fixes this issue.

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Lecture Recap: Bakery Lock

Filter Lock is not fair.
The Bakery Lock fixes this issue.

Remember fairness?

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
      flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Lecture Recap: Bakery Lock

Filter Lock is not fair.
The Bakery Lock fixes this issue.

Remember fairness

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Lecture Recap: Bakery Lock

Filter Lock is not fair.
The Bakery Lock fixes this issue.

Remember fairness

What is the doorway section and the waiting section?

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Lecture Recap: Bakery Lock

Filter Lock is not fair.
The Bakery Lock fixes this issue.

Remember fairness

What is the doorway section and the waiting section?

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
      flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Lecture Recap: Bakery Lock

Filter Lock is not fair.
The Bakery Lock fixes this issue.

Remember fairness

What is the doorway section and the waiting section?

How would you prove fairness?

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Lecture Recap: Bakery Lock

Filter Lock is not fair.
The Bakery Lock fixes this issue.

Remember fairness

What is the doorway section and the waiting section?

How would you prove deadlock-freedom?

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Lecture Recap: Bakery Lock

Filter Lock is not fair.
The Bakery Lock fixes this issue.

Remember fairness

What is the doorway section and the waiting section?

Deadlock-freedom + “first-come-first-serve” implies Starvation-freedom

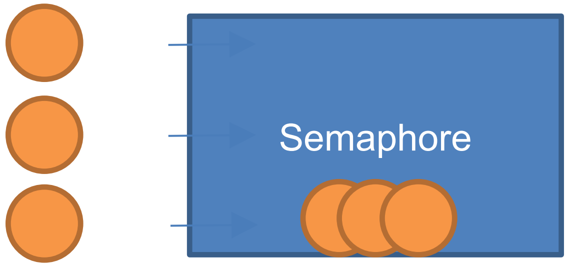
```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Lecture Recap: Semaphores

Used to restrict the number of threads that can access a specific resource.

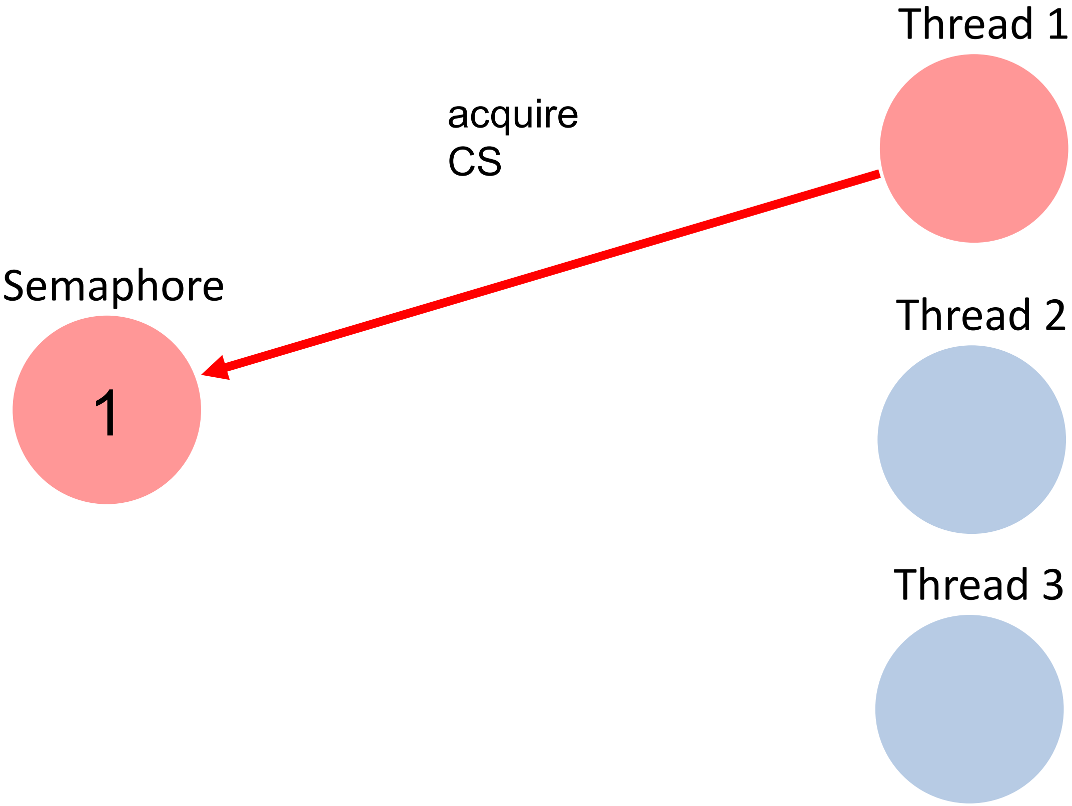
- acquire() gets a permit, if no permit available block
- release() gives up permit, releases a blocking acquirer

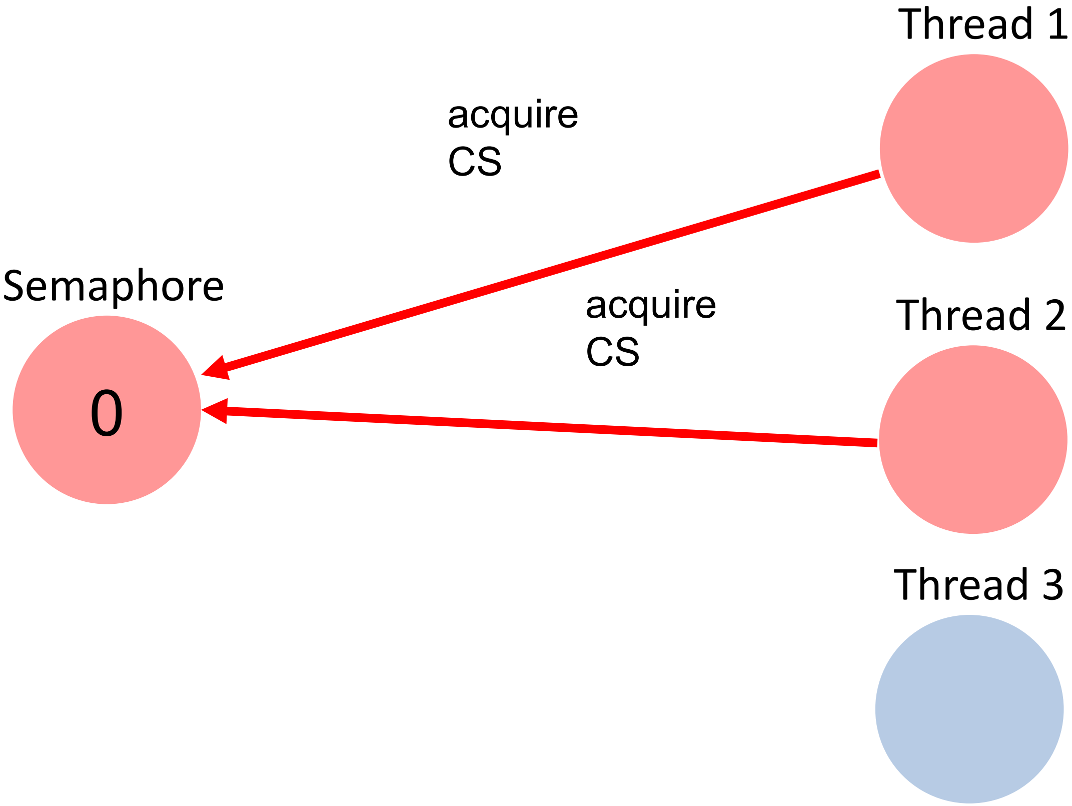
Lecture Recap: Semaphores

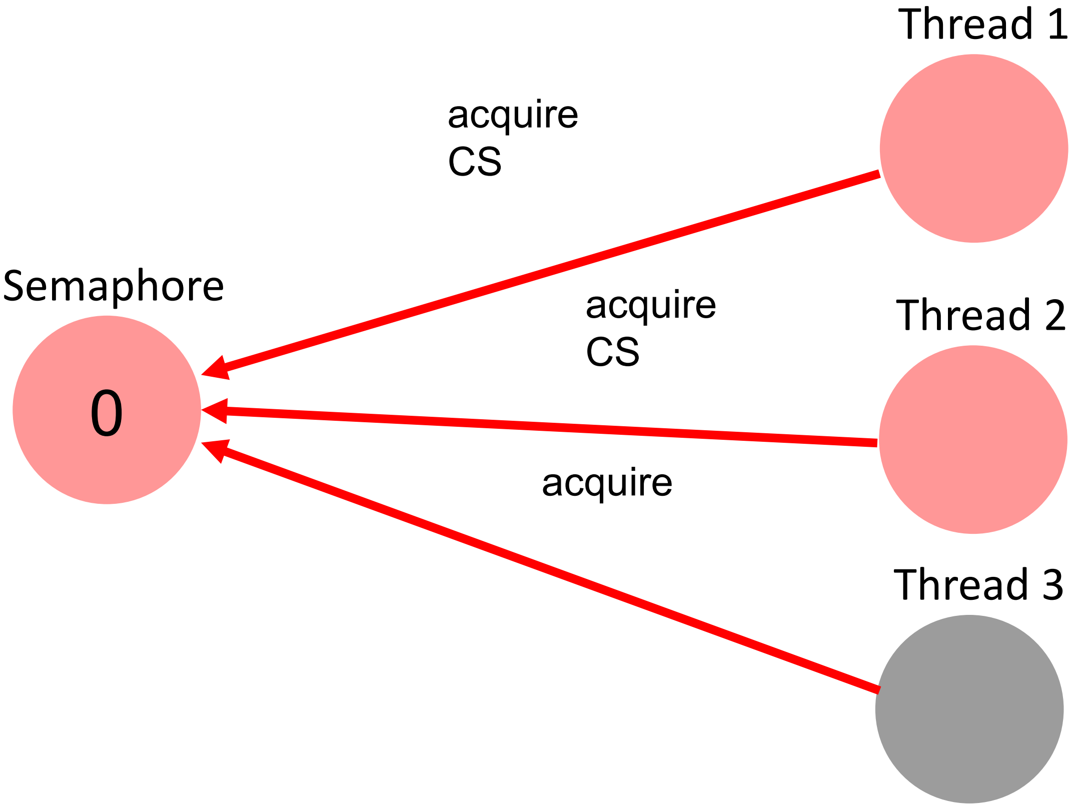


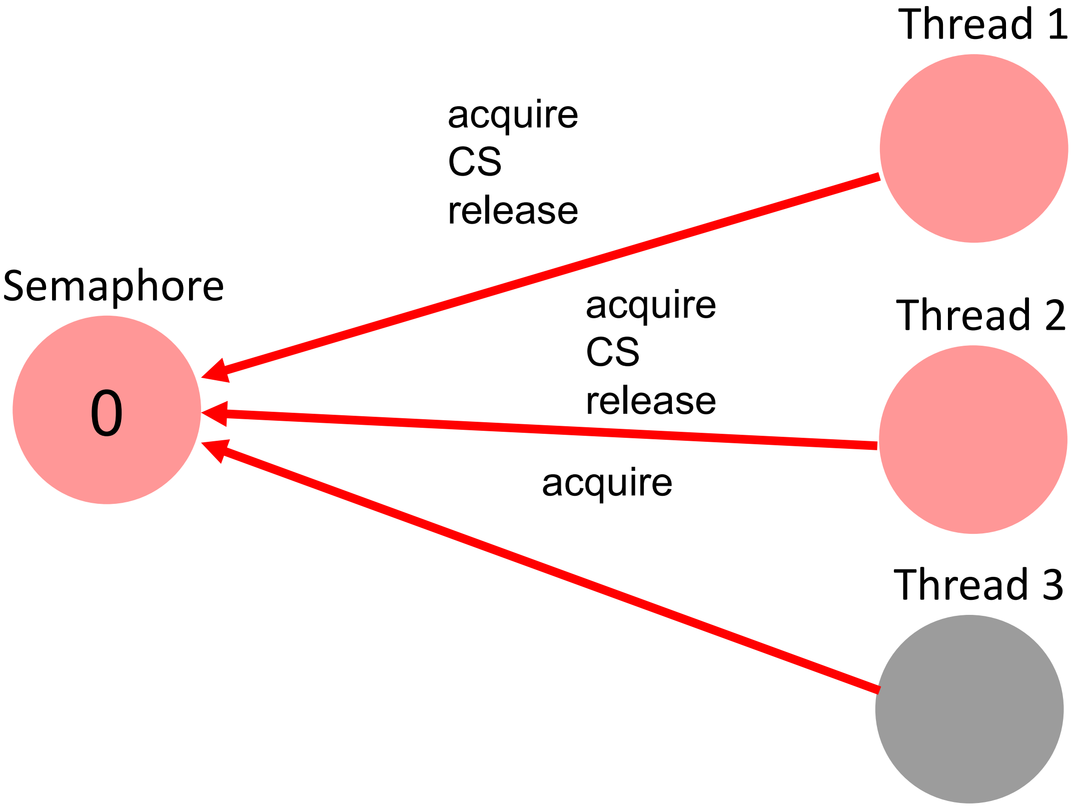
N Threads have permit to a semaphore,
others will wait (blocked) until someone leaves the semaphore


Semaphore
2
Thread 1
Thread 2
Thread 3

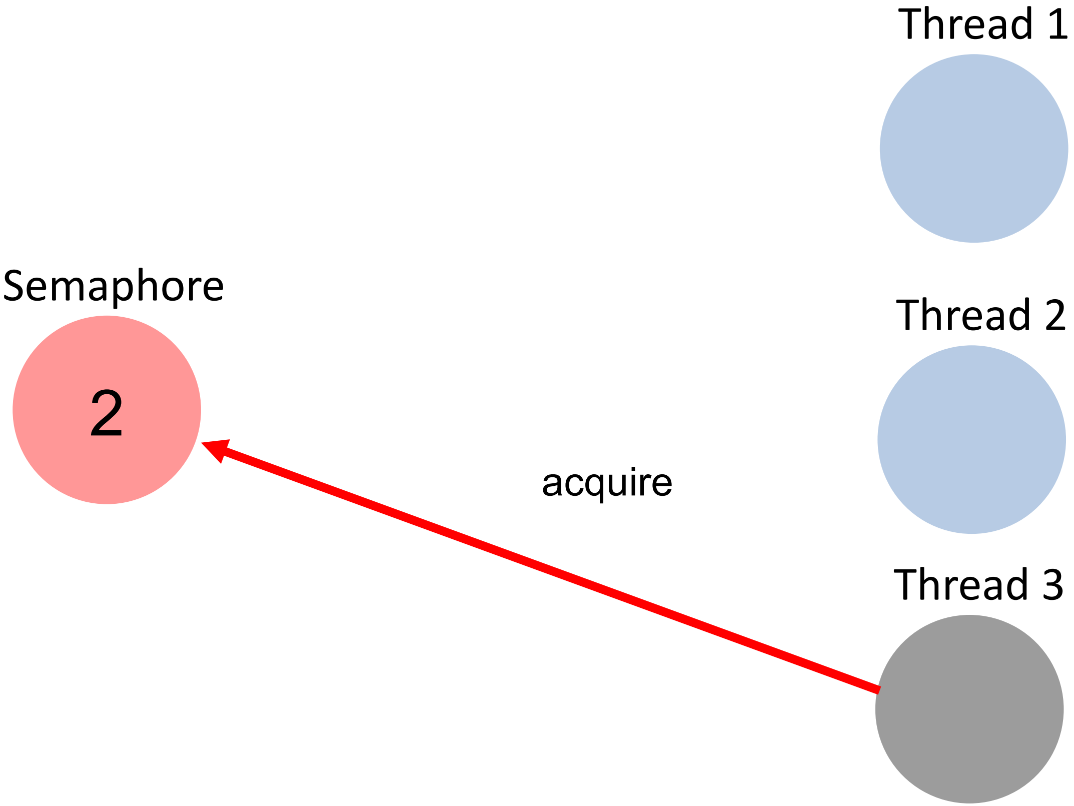

Thread 1
acquire
CS
Semaphore
1
Thread 2
Thread 3

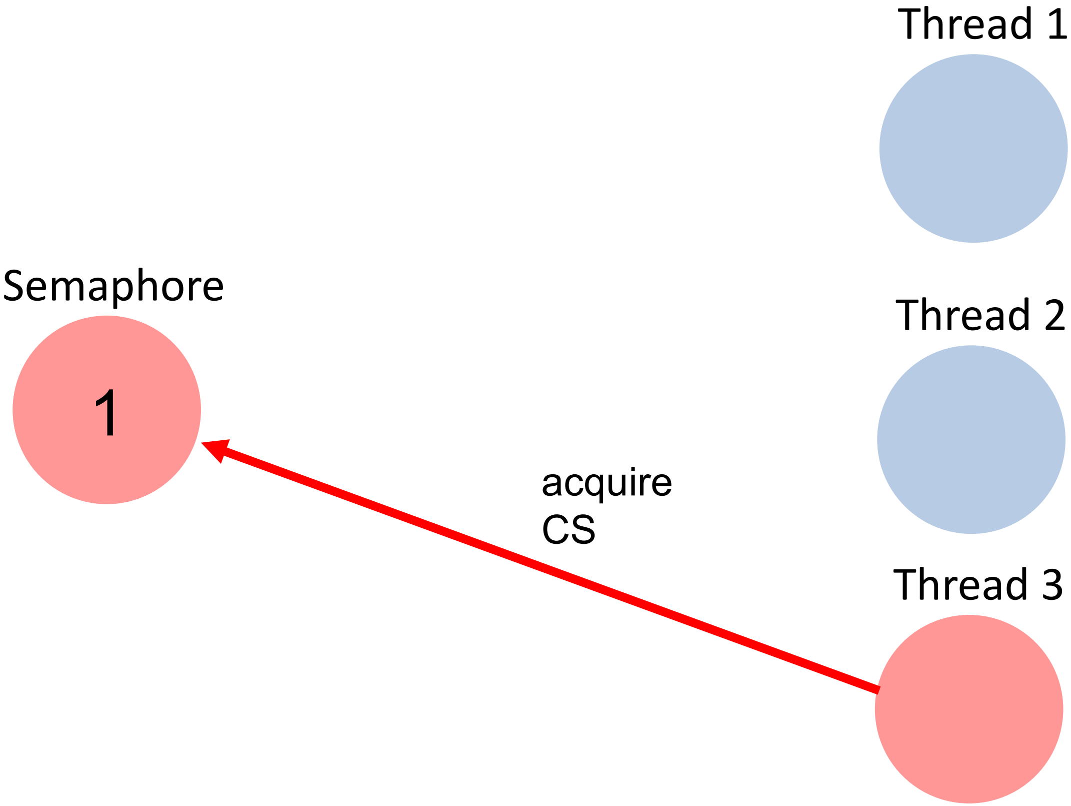

Thread 1
acquire
CS
Semaphore
0
acquire
CS
Thread 2
Thread 3


Thread 1
acquire
CS
Semaphore
0
acquire
CS
Thread 2
acquire
Thread 3



Thread 1
acquire
CS
release
Semaphore
0
acquire
CS
release
Thread 2
acquire
Thread 3


Thread 1
Semaphore
2
Thread 2
acquire
Thread 3



Think of semaphores as bike rentals

Semaphores: Implementation

Semaphore: integer-valued abstract data type S with some initial value $s \geq 0$ and the following **atomic** operations:

```
acquire(S) {
        wait until S > 0
        dec(S)
}

release(S) {
        inc(S)
}
```

Semaphores: Implementation

Semaphore: integer-valued abstract data type S with some initial value s≥0 and the following **atomic** operations:

```
acquire(S) {
        wait until S > 0
        dec(S)
}

release(S) {
        inc(S)
}
```

What is the difference between a Lock and a Semaphore?

Semaphores: Implementation

Semaphore: integer-valued abstract data type S with some initial value $s \geq 0$ and the following **atomic** operations:

```
acquire(S) {
        wait until S > 0
        dec(S)
}

release(S) {
        inc(S)
}
```

When would you use a semaphore?

Semaphores: Usage example

```
class Pool {
        private static final int MAX_AVAILABLE = 100;
        private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

        public Object getItem() throws InterruptedException {
          available.acquire();
          return getNextAvailableItem();
        }

        public void putItem(Object x) {
          if (markAsUnused(x))
            available.release();
        }

        //...
```

Semaphores: Usage example

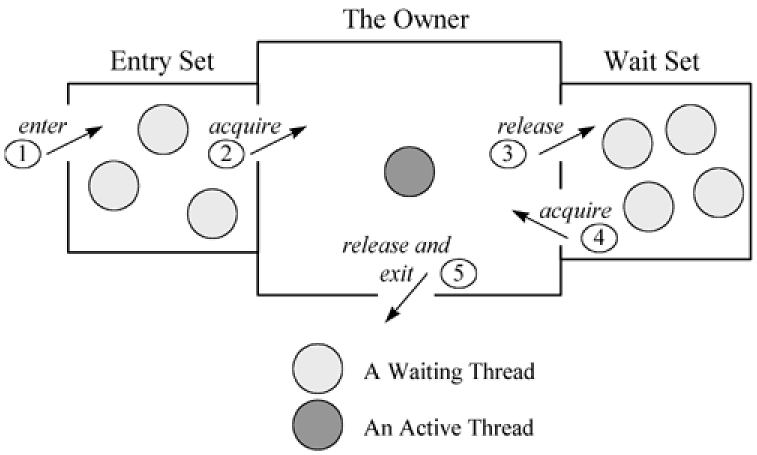
```
protected Object[] items = new Object[MAX_AVAILABLE];
protected boolean[] used = new boolean[MAX_AVAILABLE];

protected synchronized Object getNextAvailableItem() {
  for (int i = 0; i < MAX_AVAILABLE; ++i) {
    if (!used[i]) {
       used[i] = true;
       return items[i];
    }
  }
  return null; // not reached
}

protected synchronized boolean markAsUnused(Object item) {
  for (int i = 0; i < MAX_AVAILABLE; ++i) {
    if (item == items[i]) {
       if (used[i]) {
         used[i] = false;
         return true;
       } else
         return false;
    }
  }
  return false;
}
}
```

Lecture Recap: Monitors

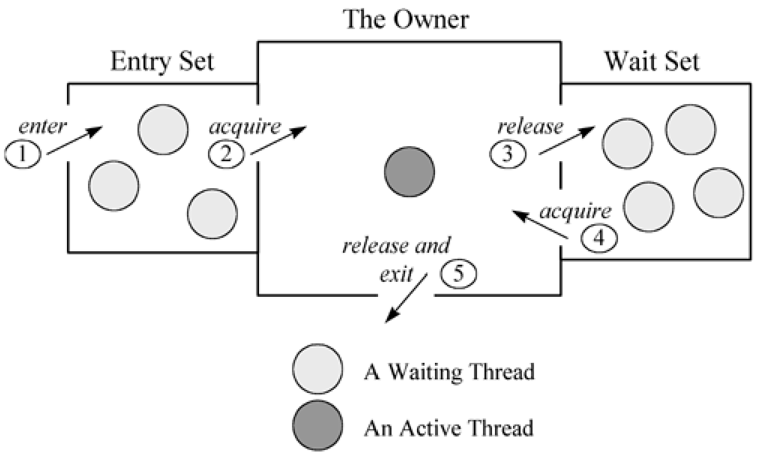
Monitors provide two kinds of thread synchronization: **mutual exclusion** and **cooperation** using a lock



- higher level mechanism than semaphores and more powerful
- instance of a class that can be used safely by several threads
- all methods of a monitor are executed with mutual exclusion

Lecture Recap: Monitors

Monitors provide two kinds of thread synchronization: **mutual exclusion** and **cooperation** using a lock



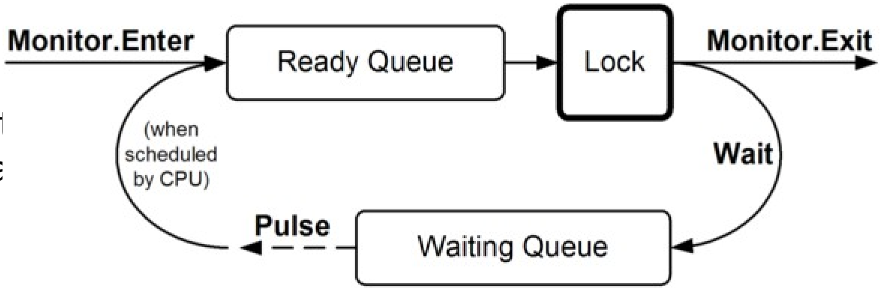
- the possibility to make a thread waiting for a condition
- signal one or more threads that a condition has been met

When thread is sent to wait we release the lock!
Can a monitor induce a deadlock?

Monitors in Java

Uses intrinsic lock (synchronized) of an object

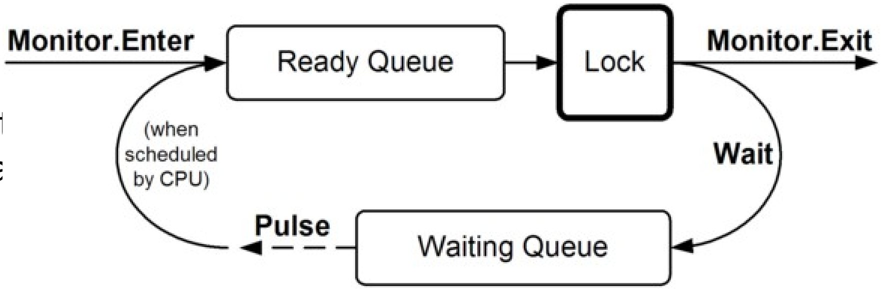
wait() – the current thread waits unt
notify() – wakes up one waiting threa
notifyAll() – wakes up all waiting threads



Monitors in Java

Uses intrinsic lock (synchronized) of an object

wait() – the current thread waits unt
notify() – wakes up one waiting threa
notifyAll() – wakes up all waiting threads



When do you use notify, when notifyAll?

Monitors in Java: Signal & Continue

- signalling process continues running
- signalling process moves signalled process to entry queue

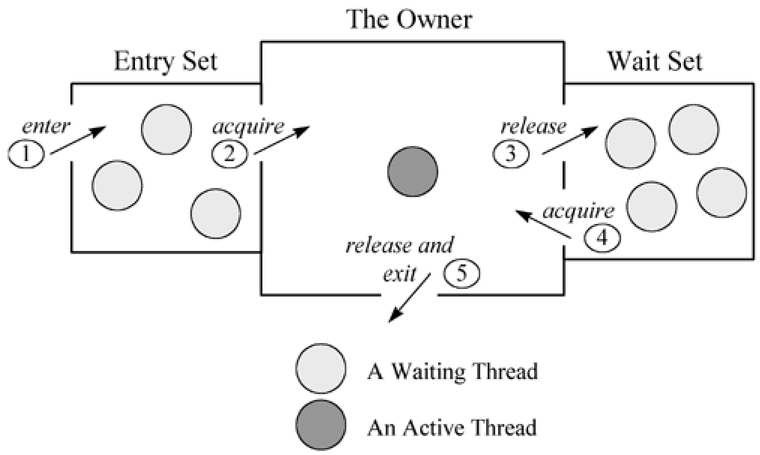
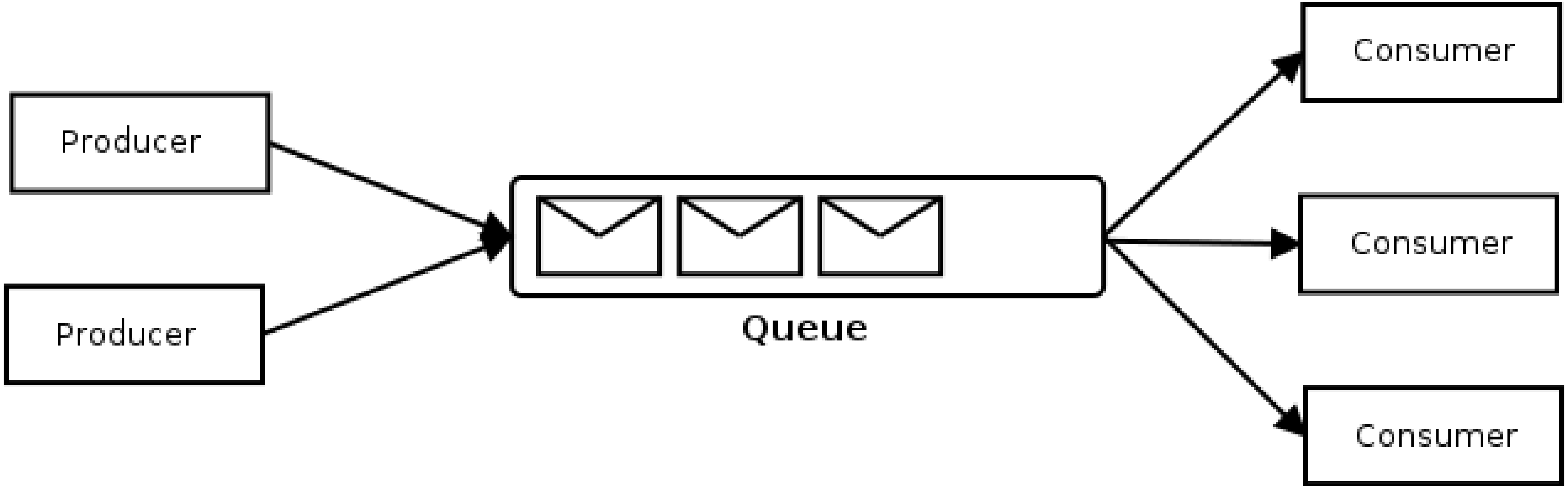


Figure 20-1. A Java monitor.

More theory:

- **Signal & Continue (SC)** : The process who signal keep the mutual exclusion and the signaled will be awaken but need to acquire the mutual exclusion before going. (Java's option)
- **Signal & Wait (SW)** : The signaler is blocked and must wait for mutual exclusion to continue and the signaled thread is directly awaken and can start continue its operations.
- **Signal & Urgent Wait (SU)** : Like SW but the signaler thread has the guarantee that it would go just after the signaled thread
- **Signal & Exit (SX)** : The signaler exits from the method directly after the signal and the signaled thread can start directly.

Monitors in Java: Example P/C Queue



Monitors in Java: Example P/C Queue

```
synchronized void enqueue(long x) {
 if (isFull()){
  try {
    wait();
  } catch (InterruptedException e) {}
 }
 doEnqueue(x);
 notifyAll();
}
```

```
synchronized long dequeue() {
 long x;
 if (isEmpty()){
  try {
    wait();
  }
  catch (InterruptedException e) {}
 }
 x = doDequeue();
 notifyAll();
 return x;
}
```

Monitors in Java: Example P/C Queue

```
synchronized void enqueue(long x) {
 if (isFull()){
  try {
    wait();
  } catch (InterruptedException e) {}
 }
 doEnqueue(x);
 notifyAll();
}
```

```
synchronized long dequeue() {
 long x;
 if (isEmpty()){
  try {
    wait();
  }
  catch (InterruptedException e) {}
 }
 x = doDequeue();
 notifyAll();
 return x;
}
```

Exercise: What can go wrong?

Monitors in Java: Example P/C Queue

```
synchronized void enqueue(long x) {
 if (isFull()){
  try {
    wait();
  } catch (InterruptedException e) {}
 }
 doEnqueue(x);
 notifyAll();
}
```

1. Queue is full
2. Process Q enters enqueue(), sees `isFull()`, and goes to the waiting list.
3. Process P enters dequeue()
4. In this moment process R wants to enter enqueue() and blocks
5. P signals Q and thus moves it into the ready queue, P then exits dequeue()
6. R enters the monitor before Q and sees `! isFull()`, fills the queue, and exits the monitor
7. Q resumes execution assuming `isFull()` is `false`

=> Inconsistency!

Monitors in Java: Example P/C Queue

```
synchronized void enqueue(long x) {
 while (isFull()){
  try {
    wait();
  } catch (InterruptedException e) {}
 }
 doEnqueue(x);
 notifyAll();
}
```

```
synchronized long dequeue() {
 long x;
 while (isEmpty()){
  try {
    wait();
  }
  catch (InterruptedException e) {}
 }
 x = doDequeue();
 notifyAll();
 return x;
}
```

Lecture Recap: Lock Conditions

Can be used to implement monitors!

Java Locks provide conditions that can be instantiated Condition

```
notFull = lock.newCondition();
```

Java conditions offer

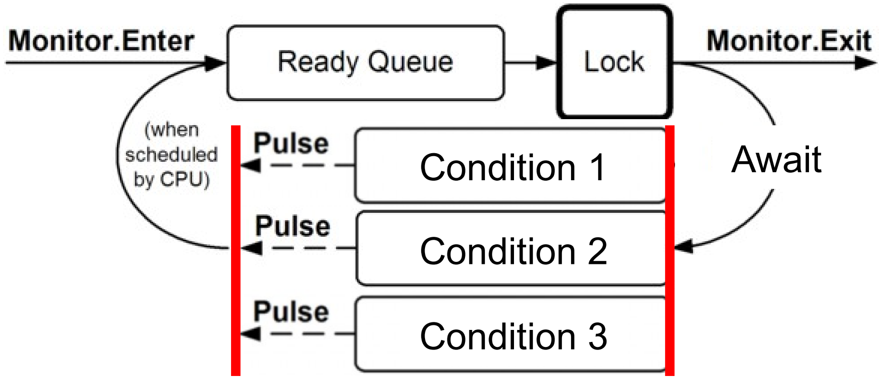
`.await()` – the current thread waits until condition is signaled

`.signal()` – wakes up one thread waiting on this condition

`.signalAll()` – wakes up all threads waiting on this condition

What is the difference to a Monitor?

Lock Conditions



Lock Conditions: Example P/C Queue

```
public class ProducerConsumer {
    private final Queue<Object> items;
    private final int capacity;

    private final Lock lock = new ReentrantLock();

    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();

    public ProducerConsumer(int capacity) {
        items = new ArrayDeque<Object>(capacity);
        this.capacity = capacity;
    }
```

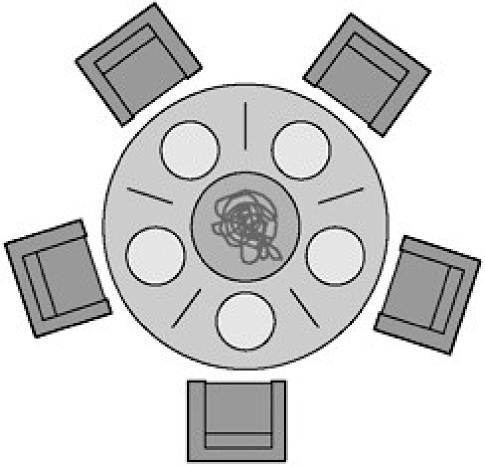
Lock Conditions: Example P/C Queue

```
public void produce(Object data) throws InterruptedException {
    lock.lock();
    try {
        while (items.size()==capacity) {
            notFull.await();
        }
        items.add(data);
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

```
public Object consume() throws InterruptedException {
    lock.lock();
    try {
        while (items.isEmpty()) {
            notEmpty.await();
        }
        Object result = items.remove();
        notFull.signal();
        return result;
    } finally {
        lock.unlock();
    }
}
```

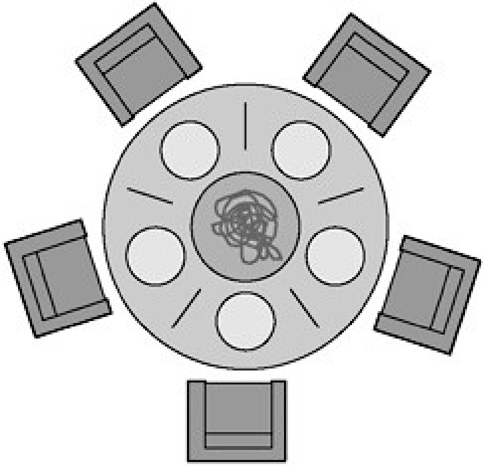
Assignment 9

Task 1 - Dining Philosophers



Originally proposed by E. W. Dijkstra
Imagine five philosophers who spend their lives thinking and eating.
They sit around a circular table with five chairs with a big plate of spaghetti.
However, there are only five chopsticks available.

Task 1 - Dining Philosophers

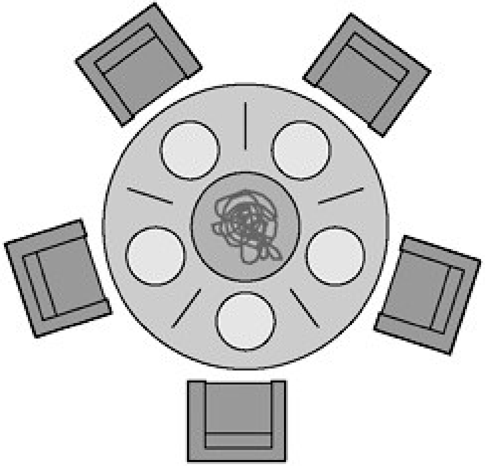


Each philosopher thinks and when he gets hungry picks up the two chopsticks closest to him.

• If a philosopher can pick up BOTH chopsticks, he eats for a while.

• After a philosopher finishes eating, he puts down the chopsticks and starts to think again.

Find a solution that…

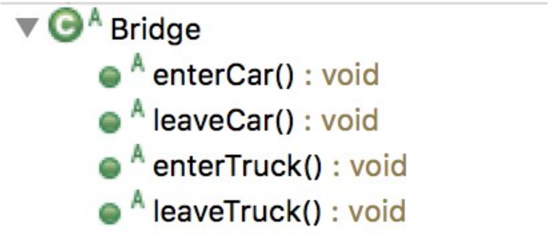


- Makes deadlocks impossible
- Has no starvation
- More than one parallel eating philosopher is possible

Task 2 – Monitors, Conditions and Bridges

Only either 3 cars or one truck may be on the bridge at each moment.

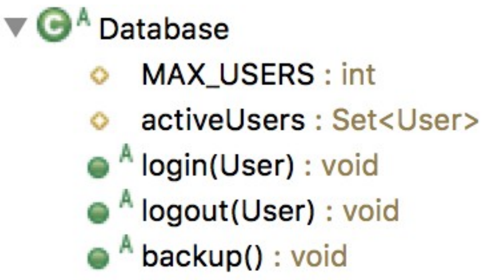
Implement Classes `BridgeMonitor` and `BridgeCondition`



How to Test my Implementation?
Implement method `invariant()` to check if the state is valid: at the end of a method there are never too many cars or trucks on the bridge

Task 3 – Semaphores and Databases

Use semaphores to implement login and logout database functionality that supports up to 10 concurrent users



Use barrier to implement 2-phase backup functionality.

Task 3 – Semaphores and Databases

Implement Classes MySemaphore and MyBarrier

Use monitors for both to avoid busy loop

- Put processes to sleep, when there is no entry into semaphore
- Wake up a waiting process when releasing a semaphore

```
acquire(S) {
        wait until S > 0
        dec(S)
}

release(S) {
        inc(S)
}
```

Try to understand the existing DatabaseJava implementation before implementing your own semaphore and barrier.