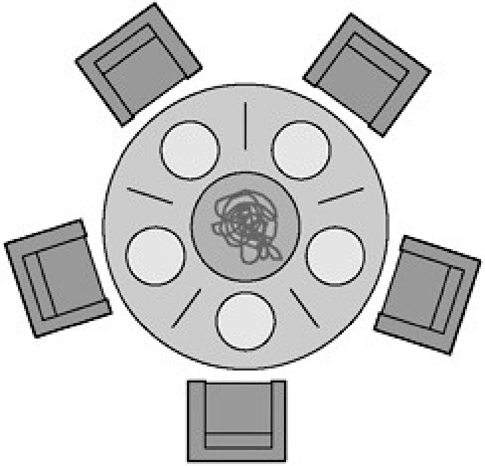
# Parallel Programming
# Exercise Session 10

# Outline

1. Feedback: Assignment 9
2. Assignment 10

# Feedback: Assignment 9

# Task 1 - Dining Philosophers



- Example deadlock

Each philosopher picks up the left fork first

- Makes deadlocks impossible

Any solution that breaks the cyclic dependency

- More than one parallel eating philosopher is possible

Bundle the forks in one place such that they are always picked up together.

# Task 2 – Better than Dijkstra

```
C0: b(i) := false;
C1: if k != i then begin
C2: if !b(j) then go to C2;
     else k := i; go to C1; end;
     else CS;
     b(i) := true
```

# Task 2 – Better than Dijkstra

Lets add some indention

```
C0: b(i) := false;
C1: if k != i then
        begin
C2:         if !b(j) then go to C2;
            else k := i; go to C1;
        end;
        else CS;
        b(i) := true
```

# Task 2 – Better than Dijkstra

Lets translate gotos into loops

```
S1:    b(i) = false;
S2:    while (k != i) {
S3:        while (!b(j)) {};
S4:        k = i;
       }
S5:   // CS
S6:   b(i) = true
```

Now we need to decide what initial values k and b have. Lets assume k=0, b = [true, true]

# Task 2 – Better than Dijkstra

For both threads to be in the CS, the following must happen (assume wlog. the process with i=0, j=1 enters the CS first):

P0:W(b[0]=false) > P0:R(k=0) > P0:CR
P1:W(b[1]=false) > P1:R(k=0) > P1:R(b[0]=true) > P1:W(k=1) > P1:R(k=1) > P1:CR

It is simple to construct a valid interleaving of these actions:

P1:W(b[1]=false) > P1:R(k=0) > P1:R(b[0]=true) > P0:W(b[0]=false) > P0:R(k=0) > P0:CR > P1:W(k=1) > P1:R(k=1) > P1:CR

Thus the lock does not work correctly.

# Task 3 – Transitive Closure

Relation: “can fly from A to B directly”

Transitive closure: If we can fly from A to B and from B to C then A and C are in the transitive closure

→ Transitive closure tells us which places are reachable.

# Task 4 – Synchronization Actions

Synchronization actions are:

- A volatile read of a variable.
- A volatile write of a variable.
- Lock
- Unlock
- The (synthetic) first and last action of a thread.
- Actions that start a thread or detect that a thread has terminated

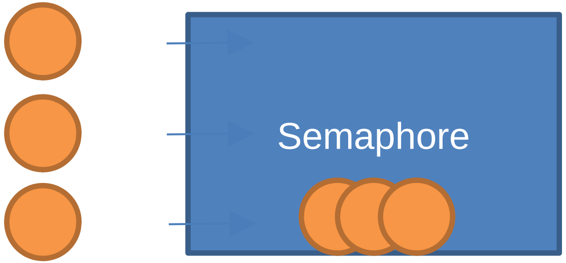
# Assignment 10

# Lecture Recap: Semaphores

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

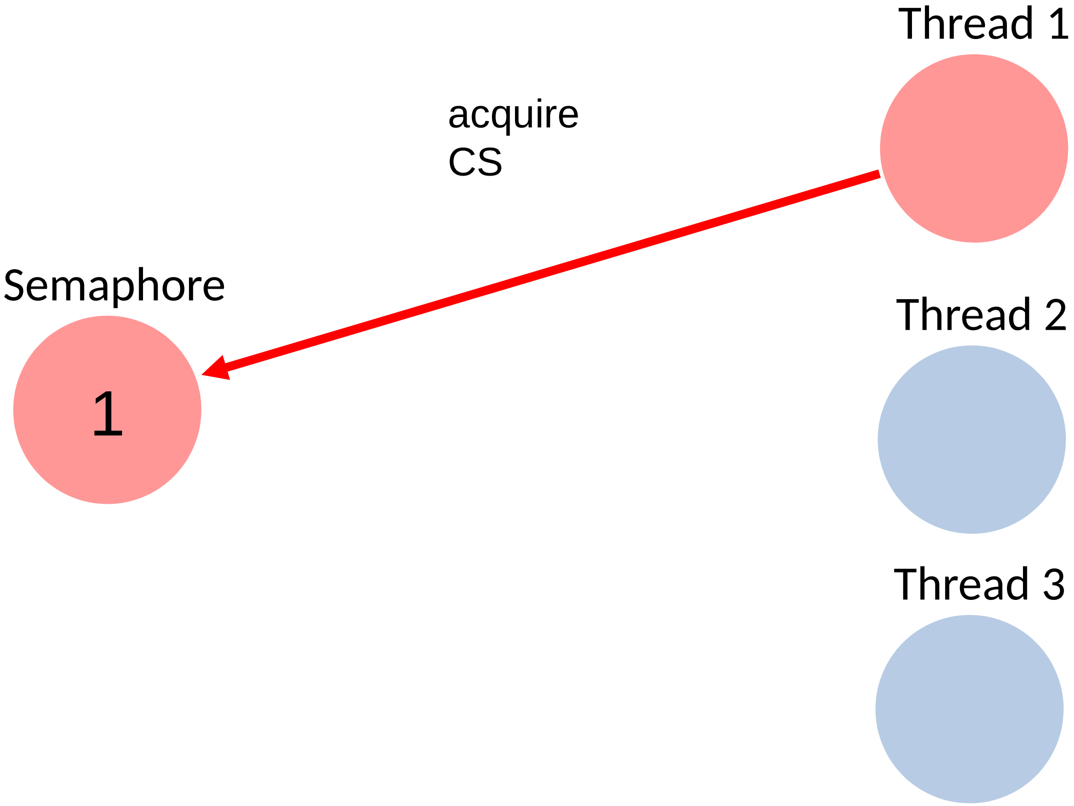
- acquire() gets a permit, if no permit available block
- release() gives up permit, releases a blocking acquirer

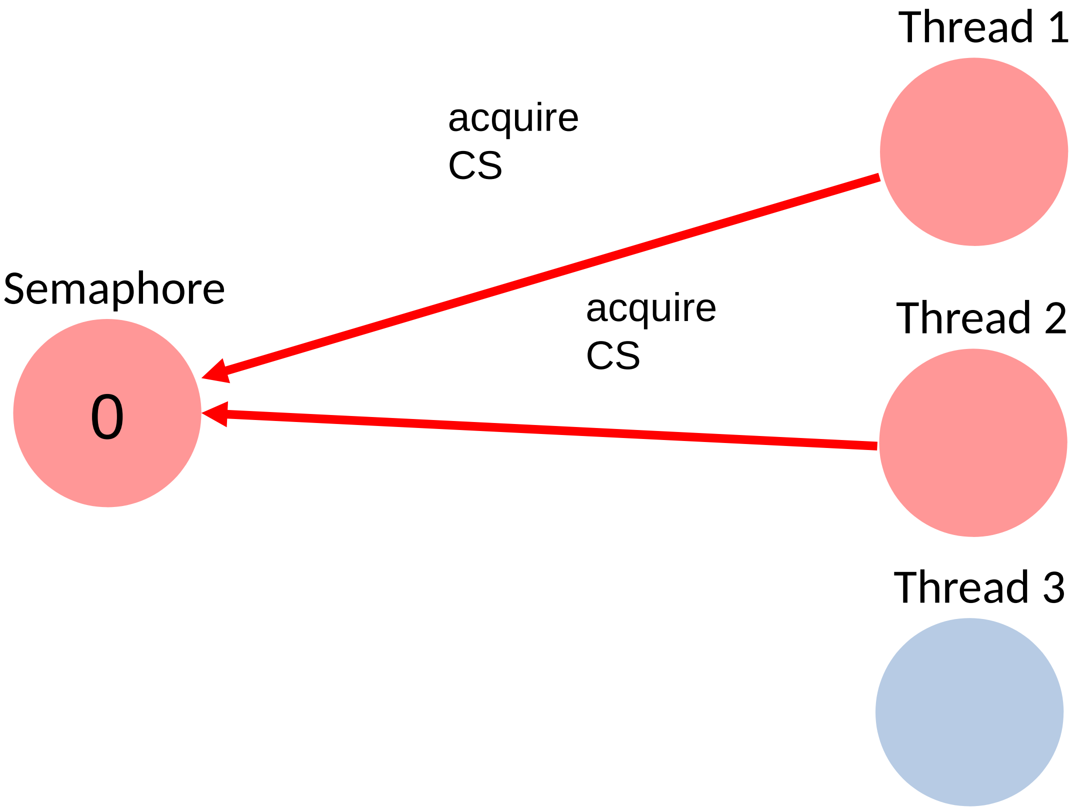
# Lecture Recap: Semaphores

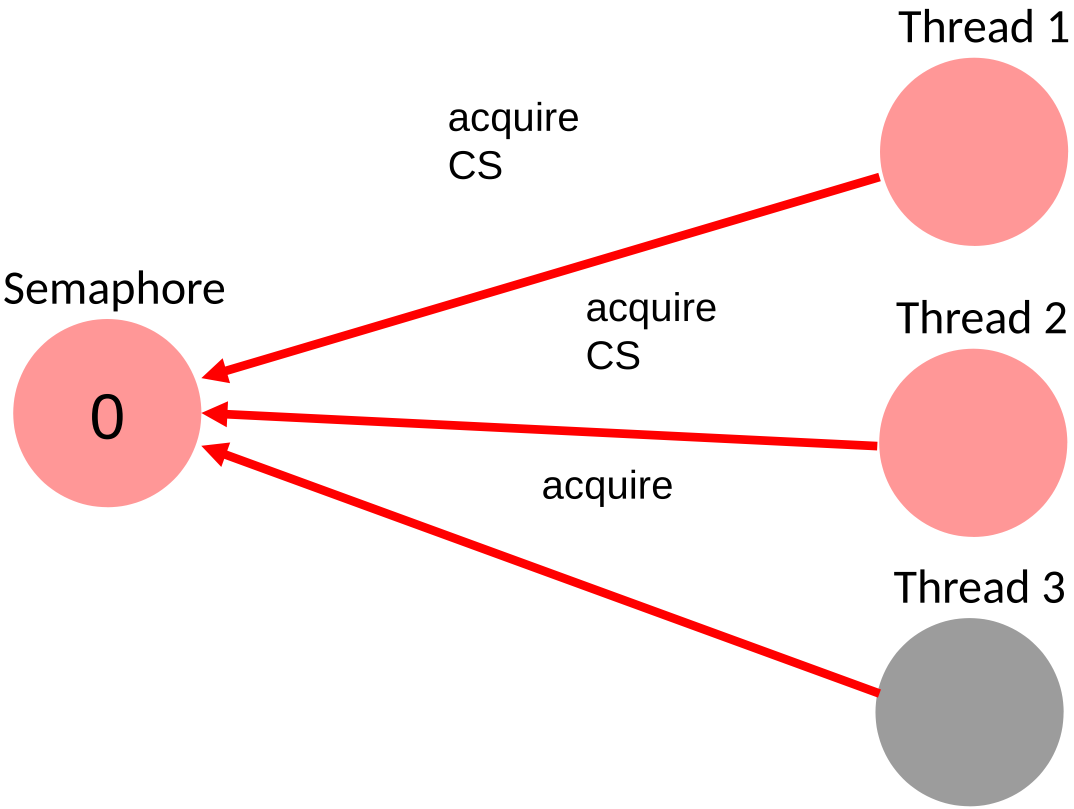


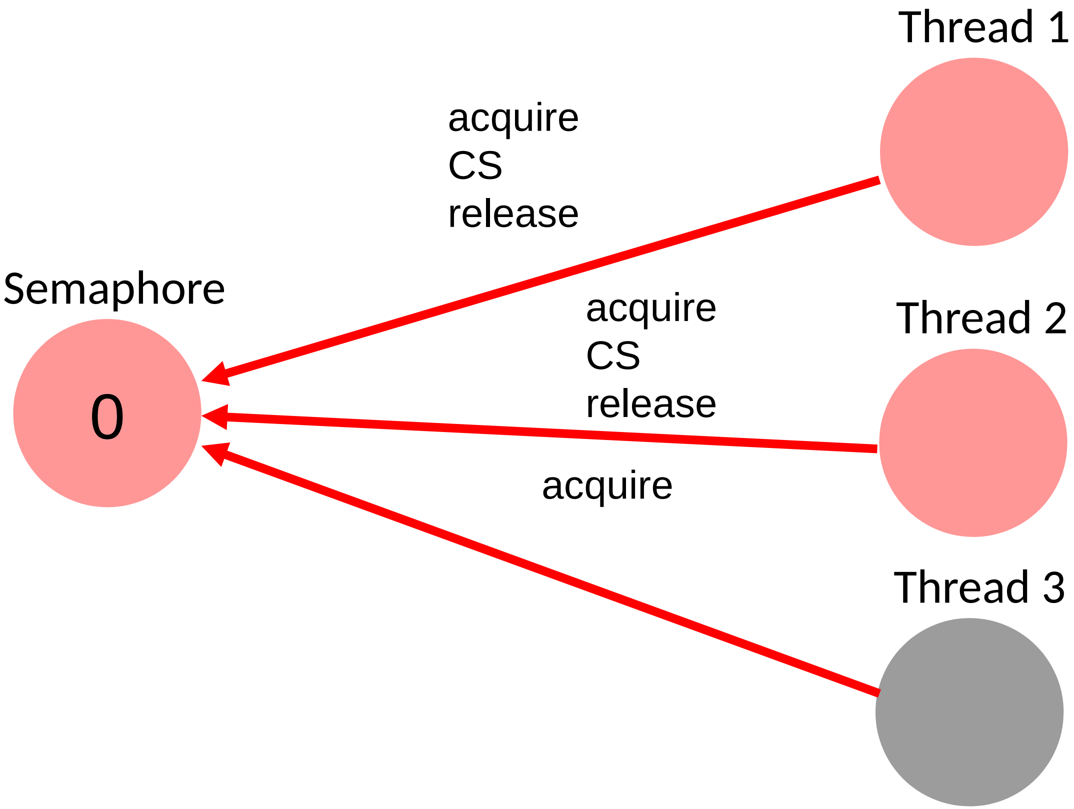

N Threads have permit to a semaphore,
others will wait (blocked) until someone leaves the semaphore

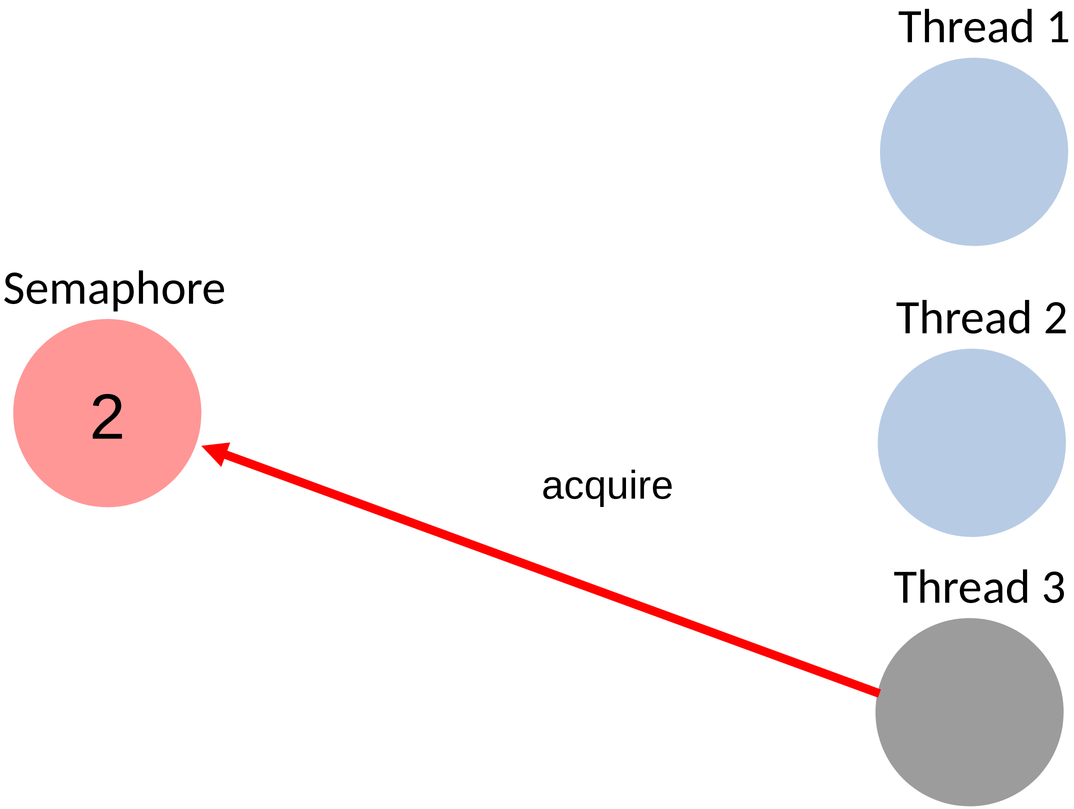

Semaphore
2
Thread 1
Thread 2
Thread 3

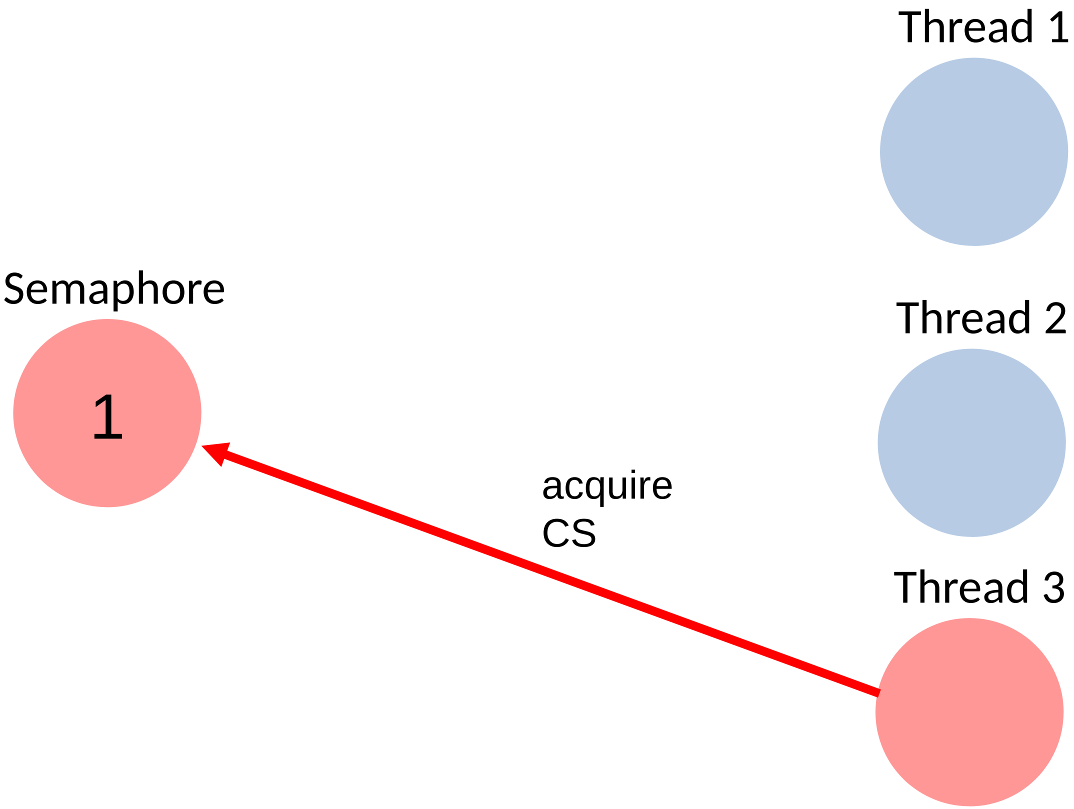

Thread 1
acquire
CS
Semaphore
1
Thread 2
Thread 3


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≥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

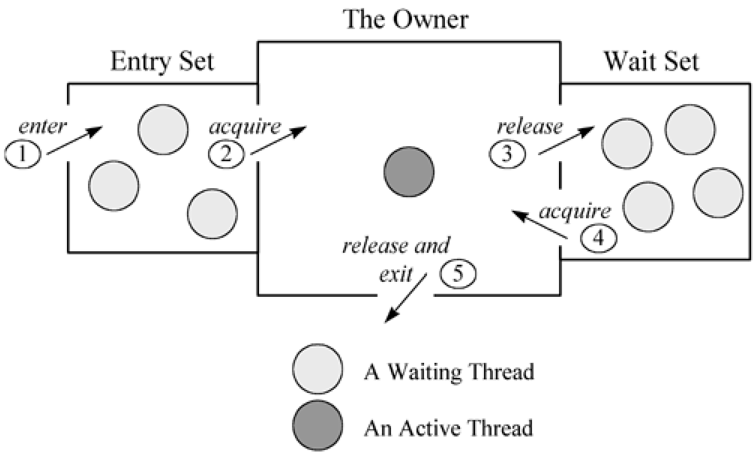
```
protected Object[] items = new Object[MAX_AVAILABLE];
protected boolean[] used = new boolean[MAX_AVAILABLE];

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

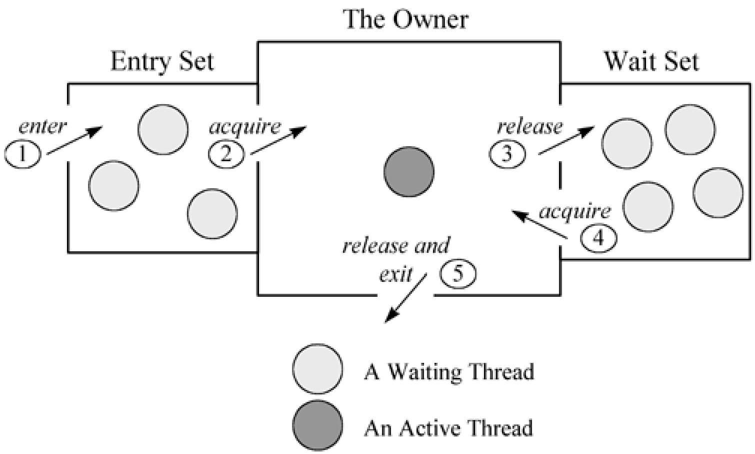
Monitors provide two kinds of thread synchronization: **mutual exclusion** and **cooperation** using a lock




- 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



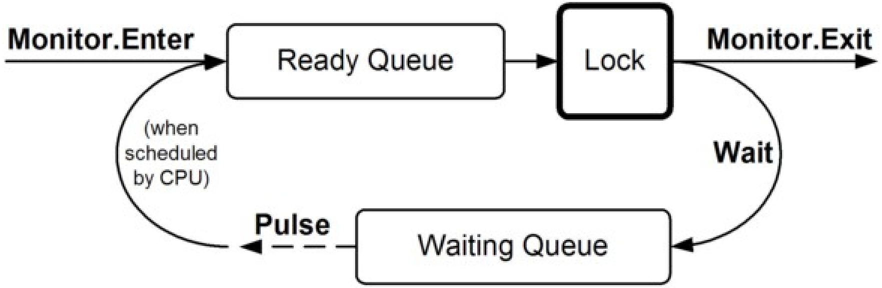

- the possibility to make a thread waiting for a condition
- signal one or more threads that a condition has been met

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

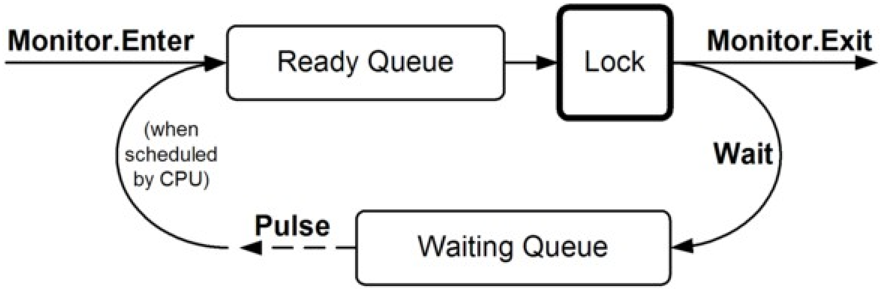
wait() – the current thread waits until it is
notify() – wakes up one waiting thread
notifyAll() – wakes up all waiting threads



# Monitors in Java

Uses intrinsic lock (synchronized) of an object

wait() – the current thread waits until it is
notify() – wakes up one waiting thread
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

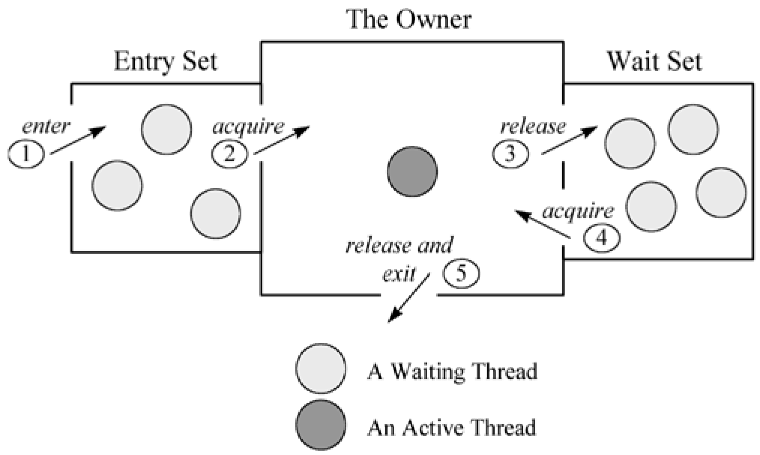



Figure 20-1. A Java monitor.

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 than 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: Signal & Continue

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

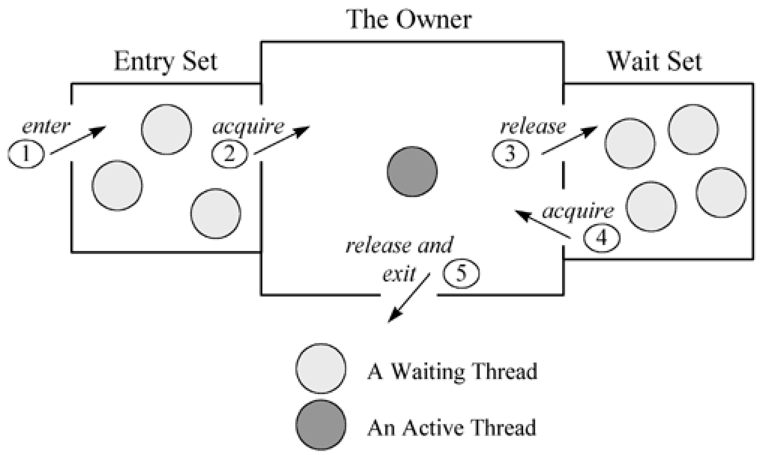
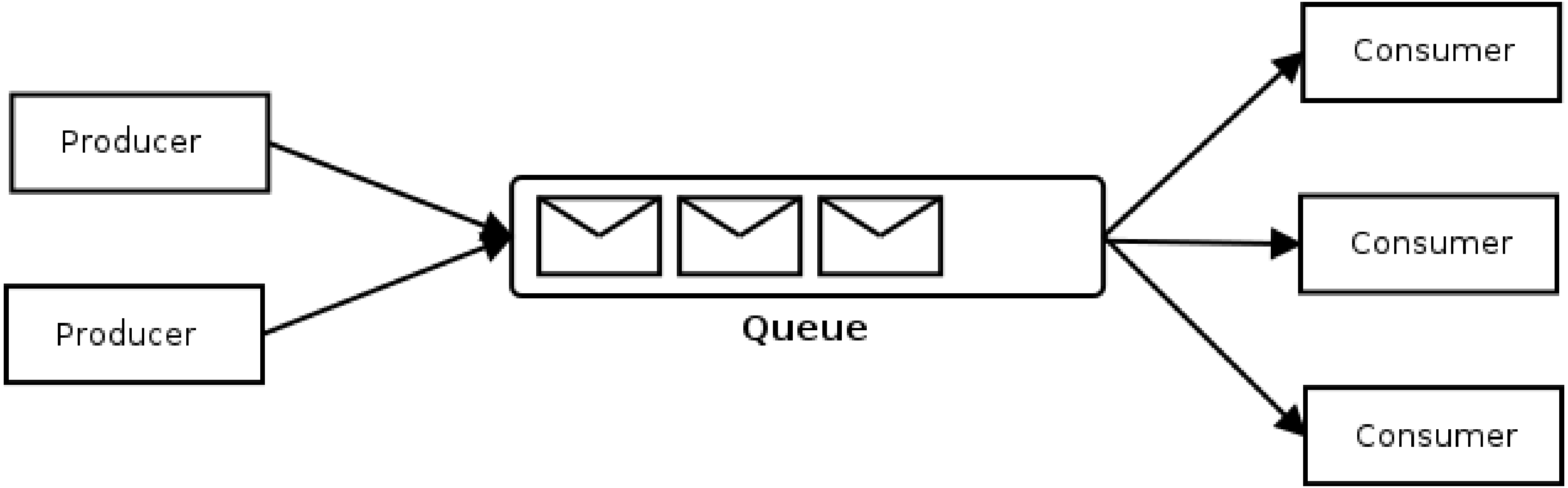



Figure 20-1. A Java monitor.

More abstractly there are 4 options:

- **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 than 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();
}
```

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

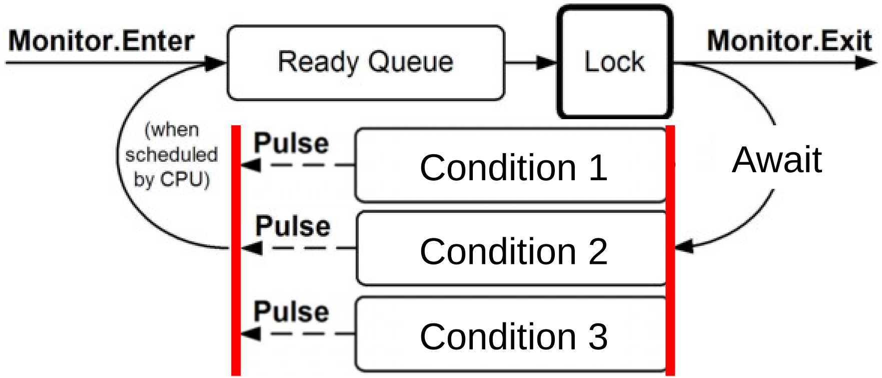
```
notFull = lock.newCondition();
```

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

```java
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();
    }
}
```