

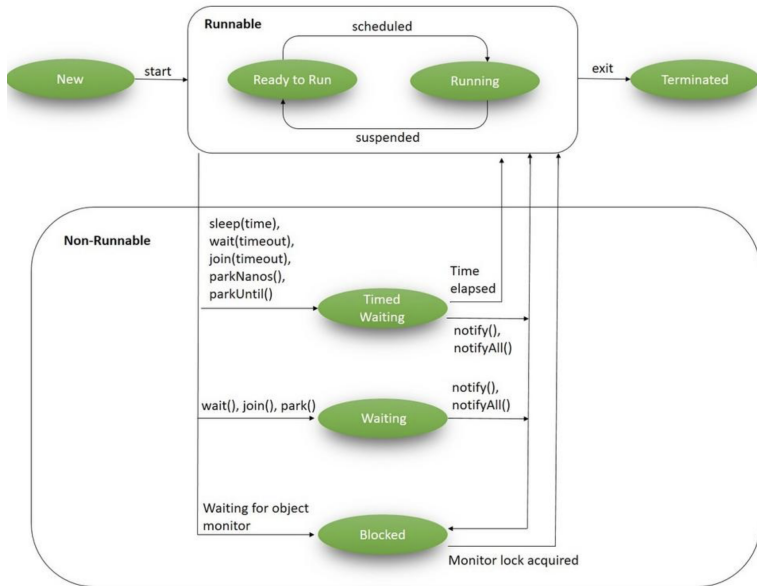
# Semaphores

CS511

# Motivation

- ▶ Algorithms for mutex seen up until now run on any machine (they only use standard instructions)
- ▶ These are too low-level to be used reliably
- ▶ Semaphores are higher-level constructs
  - ▶ Usually implemented by the OS
  - ▶ Widely used in many PLs

# States of a Process<sup>1</sup>



<sup>1</sup>Source for diagram: [www.baeldung.com/java-thread-lifecycle](http://www.baeldung.com/java-thread-lifecycle)

# Semaphore

A semaphore is an Abstract Data Type with:

- ▶ (Atomic!) Operations:
  - ▶ acquire (or wait)
  - ▶ release (or signal)
- ▶ Data fields:
  - ▶ permissions: integer
  - ▶ processes: set of processes

# Acquire

Acquire consumes a permission, waits if none are available

```
1  atomic acquire() {  
2      currentThread = Thread.currentThread();  
3      if (permissions > 0) {  
4          permissions--;  
5      } else {  
6          processes.add(thread);  
7          currentThread.state = WAITING;  
8      }  
9  }
```

# Release

Release frees a permission (wakens a waiting thread, if there are any)

```
1  atomic release() {  
2      if (processes.empty() || permissions < 0) {  
3          permissions++;  
4      } else {  
5          wakingThread = processes.removeAny();  
6          wakingThread.state = READY_TO_RUN;  
7      }  
8  }
```

Note: if initial number of permissions is positive ( $\geq 0$ ), then the acquire/release operations maintain that invariant

# Binary vs Counting Semaphores

- ▶ A **binary** semaphore is one whose permissions are always in the range  $[0..1]$ 
  - ▶ Sometimes also called **mutex**
  - ▶ Example of use: MEP or critical sections
- ▶ A **counting** semaphore is one that allows arbitrary values for permissions
  - ▶ Example of use: Producers/Consumers, Readers/Writers, Barriers, etc.

# Mutual Exclusion using mutex

The MEP for two processes becomes trivial if we use a `mutex`

- ▶ Entry protocol `mutex.acquire()`
- ▶ Exit protocol `mutex.release()`

```
1 Semaphore mutex = new Semaphore(1);
```

```
1 Thread.start { //P      1 Thread.start { //Q
2   while (true) {        2   while (true) {
3       // non-CS         3       // non-CS
4       mutex.acquire();  4       mutex.acquire();
5       // CS             5       // CS
6       mutex.release();  6       mutex.release();
7       // non-CS         7       // non-CS
8   }                    8   }
9 }                      9 }
```

- ▶ This solution does not use busy waiting: a process that blocks in the acquire goes into the WAITING state and only returns to the READY\_TO\_RUN state once it is given permission to do so.



# Semaphores in Java

Class Semaphore in `java.util.concurrent`

▶ `java.util.concurrent.Semaphore`

```
1  /** Creates a semaphore with the given number of permits */
2  Semaphore(int permits)

1  /** Acquires a permit from this Semaphore,
2      blocking until one is available */
3  void acquire()

1  /** Releases a permit, returning it to the semaphore */
2  void release()
```

# Semaphores in Java

Class Semaphore in java.util.concurrent

▶ java.util.concurrent.Semaphore

```
1  /** Creates a semaphore with the given number of permits */
2  Semaphore(int permits)

1  /** Acquires a permit from this Semaphore,
2      blocking until one is available */
3  void acquire()

1  /** Releases a permit, returning it to the semaphore */
2  void release()
```

Example:

```
1  Semaphore mutex = new Semaphore(1);
2  mutex.acquire()
3  // critical section
4  mutex.release()
```

## Semaphore Invariants

Let  $k$  be the initial value of the `permissions` field of a semaphore  $s$

# Semaphore Invariants

Let  $k$  be the initial value of the `permissions` field of a semaphore  $s$

1. `permissions`  $\geq 0$

# Semaphore Invariants

Let  $k$  be the initial value of the `permissions` field of a semaphore  $s$

1. `permissions`  $\geq 0$
2. `permissions`  $= k + \text{\#releases} - \text{\#acquires}$

where

- ▶ `\#releases` is the number of `s.release()` statements executed
- ▶ `\#acquire` is the number of `s.acquire()` statements executed
- ▶ A blocked process is considered not to have executed an acquire operation.

## The Semaphore Solution for the MEP ( $k = 1$ )

`#criticalSection`: number of processes in their critical sections

# The Semaphore Solution for the MEP ( $k = 1$ )

`#criticalSection`: number of processes in their critical sections

1. `#criticalSection + permissions = 1`

## The Semaphore Solution for the MEP ( $k = 1$ )

`#criticalSection`: number of processes in their critical sections

1. `#criticalSection + permissions = 1`
2. `#criticalSection = #acquires - #releases`



# The Semaphore Solution for the MEP ( $k = 1$ )

`#criticalSection`: number of processes in their critical sections

1. `#criticalSection + permissions = 1`
2. `#criticalSection = #acquires - #releases`

Item 1 guarantees:

# The Semaphore Solution for the MEP ( $k = 1$ )

#criticalSection: number of processes in their critical sections

1. #criticalSection + permissions = 1
2. #criticalSection = #acquires - #releases

Item 1 guarantees:

- ▶ **Mutual Exclusion** ( $\text{\#criticalSection} \leq 1$  since  $0 \leq \text{permissions}$ )

# The Semaphore Solution for the MEP ( $k = 1$ )

#criticalSection: number of processes in their critical sections

1. #criticalSection + permissions = 1
2. #criticalSection = #acquires - #releases

Item 1 guarantees:

- ▶ **Mutual Exclusion** (#criticalSection  $\leq 1$  since  $0 \leq \text{permissions}$ )
- ▶ **Absence of deadlock** (it never happens that permissions = 0 and #criticalSection = 0)

# The Semaphore Solution for the MEP ( $k = 1$ )

`#criticalSection`: number of processes in their critical sections

1. `#criticalSection + permissions = 1`
2. `#criticalSection = #acquires - #releases`

Item 1 guarantees:

- ▶ **Mutual Exclusion** (`#criticalSection`  $\leq 1$  since  $0 \leq \text{permissions}$ )
- ▶ **Absence of deadlock** (it never happens that `permissions = 0` and `#criticalSection = 0`)
- ▶ **No starvation** between two processes

# The Turnstile Problem using Binary Semaphores

```
1 import java.util.concurrent.Semaphore
2
3 Semaphore mutex = new Semaphore(1)
4 counter=0 // shared variable
5
6 P = Thread.start {
7     50.times {
8         mutex.acquire()
9         counter++
10        mutex.release()
11    }
12 }
13 Q = Thread.start {
14     50.times {
15         mutex.acquire()
16         counter++
17         mutex.release()
18     }
19 }
20
21 P.join() // wait for P to finish
22 Q.join() // wait for Q to finish
23
24 println(counter) // print value of counter
```

# The Turnstile Problem using Binary Semaphores (Java)

```
1 public class Turnstile extends Thread {
2     static volatile int counter = 0;
3     static Semaphore mutex = new Semaphore(1);
4     public void run() {
5         for(int i = 0; i < 50; i++){
6             mutex.acquire();
7             counter++;
8             mutex.release();
9             System.out.println(id+"- In comes: "+i );
10        }
11    }
12
13    public static void main(String args[]) {
14        try{
15            Thread m1 = new Turnstile(1);
16            m1.start();
17            Thread m2 = new Turnstile(2);
18            m2.start();
19        } catch(Exception e){}
20    }
21 }
```

# Counting Example in Java using Semaphores

```
1 public class Turnstile extends Thread {  
2     static volatile int counter = 0;  
3     ...  
}
```

- ▶ The `volatile` keyword is recommended for variables that are shared
- ▶ It guarantees that
  - ▶ Its value will never be cached thread-locally: all reads and writes will go straight to “main memory”;
  - ▶ Instructions involving such variables will not be reordered;
  - ▶ Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself (more later).

# Strong Semaphores

The same solution above for the critical section also works for  $N$  processes

```
1 N.times {
2   Thread.start {
3     while (true) {
4       // non-CS
5       mutex.acquire();
6       // CS
7       mutex.release();
8       // non-CS
9     }
10  }
11 }
```

- ▶ But there is the possibility of [starvation](#).
- ▶ The problem is caused by the fact that blocked processes are placed in a [set](#) of processes



# Strong Semaphores

- ▶ This can be remedied by changing the set to be a queue
- ▶ In Java this is indicated by the second argument of the constructor

```
1  /** Creates a Semaphore with the given number of permits
2      and the given fairness setting. */
3  Semaphore(int permits, boolean fair)
```

- ▶ When fairness is set to `true`, the semaphore gives permits to access mutual resources in the order the threads have asked for it (FIFO)

Semaphores

Synchronization Among Processes

# Synchronization Problems

- ▶ The **critical section problem** is an abstraction of the synchronization problems that occur when multiple processes compete for the same resource
- ▶ Another type of synchronization problem is when processes must coordinate the **order of execution**

# Revisiting the Turnstile Problem

Suppose we wish to print the counter total for  $N$  turnstiles

```
1 import java.util.concurrent.Semaphore;
2
3 counter = 0;
4 mutex = new Semaphore(1);
5
6 def turnstile() {
7     50.times {
8         mutex.acquire();
9         counter++;
10        mutex.release();
11    }
12 }
13
14 2.times {
15     Thread.start {
16         turnstile();
17     }
18 }
19
20 println(counter);
```

What happens when we run this code?

# Revisiting the Turnstile Problem

```
1 import java.util.concurrent.Semaphore;
2
3 counter = 0;
4 mutex = new Semaphore(1);
5 f = new Semaphore(0);
6
7 def turnstile() {
8     50.times {
9         mutex.acquire();
10        counter++;
11        mutex.release();
12    }
13    f.release();
14 }
15
16 2.times {
17     Thread.start {
18         turnstile();
19     }
20 }
21
22 f.acquire();
23 f.acquire()
24 println(counter);
```

# Dining Philosophers



# Dining Philosophers



- Philosophers think and eat, in turns

# Dining Philosophers



- ▶ Philosophers think and eat, in turns
- ▶ They can only eat if they have both forks



# Dining Philosophers



- ▶ Philosophers think and eat, in turns
- ▶ They can only eat if they have both forks
- ▶ They can only grab the forks to their left and right

# Dining Philosophers

```
1  Thread.start { // Philosopher(id)
2      while (true)
3          // think
4          // pick up forks
5          // eat
6          // leave forks
7  }
```

# Dining Philosophers

```
1  Thread.start { // Philosopher(id)
2      while (true)
3          // think
4          // pick up forks
5          // eat
6          // leave forks
7  }
```

- ▶ Shared resource: the forks
- ▶ Mutex: at any given moment only one philosopher can have a fork
- ▶ Synchronization: a philosopher can only eat if she/he has both forks
- ▶ Absence of deadlock, livelock and starvation

## Dining Philosophers (naive attempt)

```
1 final int N=5
2 List<Semaphore> forks=[]; // N
3 N.times { forks.add(new Semaphore(1)) }
4
5 N.times {
6     int id = it
7     Thread.start {
8         int left = id;
9         int right = (id+1) % N;
10        println id + "Started "+left+", "+right
11
12        while (true) {
13            // think
14            println id + " Eating..."
15            forks[left].acquire();
16            forks[right].acquire();
17            // eat
18            forks[left].release();
19            forks[right].release();
20            println id + " Done Eating..."
21        }
22    }
23 }
```

**Deadlock:** If they all take the left fork, circular waiting

# Dining Philosophers (counting semaphore)

```
1 final int N=5
2 List<Semaphore> forks=[]
3 N.times { forks.add(new Semaphore(1)) }
4 Semaphore chairs = new Semaphore(N-1)
5 N.times {
6     int id = it
7     Thread.start { // Phil(id)
8         int left = id;
9         int right = (id+1) % N;
10        println id + " Started "+left+", "+right
11
12        while (true) {
13            // think
14            chairs.acquire();
15            forks[left].acquire();
16            forks[right].acquire();
17            println id + " Eating..."
18            // eat
19            forks[left].release();
20            forks[right].release();
21            println id + " Done Eating..."
22            chairs.release();
23        }
24    }
25 }
```

# Dining Philosophers (breaking the symmetry)

```
1 final int N=5
2 List<Semaphore> forks=[]; // N
3 N.times { forks.add(new Semaphore(1)) }
4 N.times {
5     int id = it
6     Thread.start {
7         int left,right;
8         if (id == 0) {
9             left = 1;
10            right = 0;
11        } else {
12            left = id;
13            right = (id+1) % N;
14        }
15        println id + "Started "+left+", "+right
16        while (true) {
17            // think
18            forks[left].acquire();
19            forks[right].acquire();
20            println id + " Eating..."
21            // eat
22            forks[left].release();
23            forks[right].release();
24            println id + " Done Eating..."
25        }
26    }
```