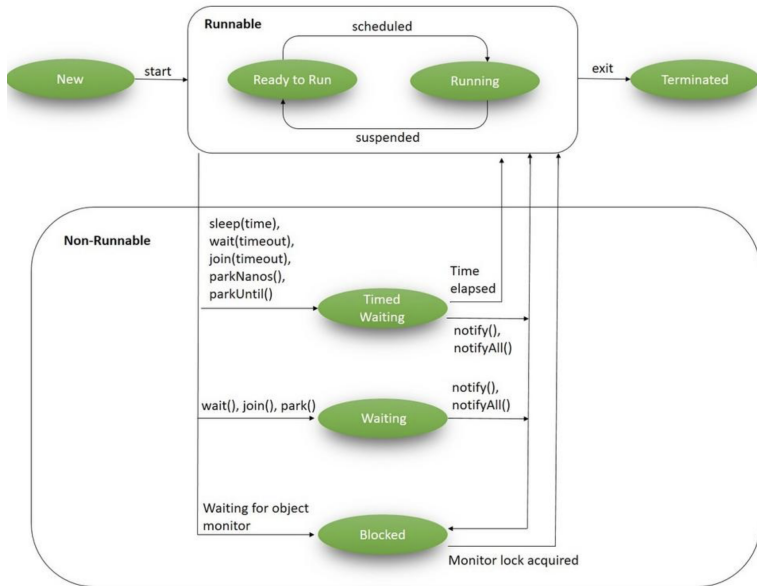# Semaphores

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[1]

---
[1]Source for diagram: 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:  non-negative 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 = BLOCKED;
8    }
9  }
```

Release

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

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

Mutex or Binary Semaphore

- ▶ A semaphore that only admits 0 or 1 permissions.
    - ▶ Semaphores that allow arbitrary values of permission are called counting semaphores
- ▶ Initialized to $(0, \emptyset)$ or $(1, \emptyset)$
- ▶ The acquire operation is unchanged
- ▶ The release operation is now defined as:

```
1   atomic release() {
2     if (permissions == 1) {
3       // do nothing
4     } else if (processes.empty()) {
5       permissions = 1;
6     } else {
7       wakingThread = processes.removeAny();
8       wakingThread.state = READY;
9     }
10  }
```

Note: if permissions is 1, succesive calls to `release` are lost

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   // non-critical section     2   // non-critical section
3   mutex.acquire();            3   mutex.acquire();
4   // critical section         4   // critical section
5   mutex.release();            5   mutex.release();
6   // non-critical section     6   // non-critical section
7 }                             7 }
```

▶ This solution does not use busy waiting: a process that blocks
  in the acquire goes into the BLOCKED state and only returns
  to the READY 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` ≥ 0

Semaphore Invariants

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

1. permissions ≥ 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. $\text{\#criticalSection} + \text{permissions} = 1$

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 (#criticalSection ≤ 1 since
$0 \leq$ 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 ≤ 1 since $0 \leq$ 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` ≤ 1 since $0 \leq$ `permissions`)

▶ Absence of deadlock (it never happens that `permissions` $= 0$ and `#criticalSection` $= 0$)

▶ No starvation between two processes

The Turnstile Problem using Binary Semaphores

```java
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)

```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

```java
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"; and
  - 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     // non-critical section
4     mutex.acquire();
5     // critical section
6     mutex.release();
7     // non-critical section
8   }
9 }
```

- ▶ 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

```java
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

- ▶ Philosophors think and eat, in turns

Dining Philosophers

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

Dining Philosophers

- ▶ Philosophors 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 Philiosophers

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

Dining Philiosophers

```
1    Thread.start { // Philosopher(id)
2      while (true)
3        // think
4        // pick 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 Philiosophers (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 (general 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
 6 N.times {
 7     int id = it
 8     Thread.start { // Phil(id)
 9         int left = id;
10         int right = (id+1) % N;
11         println id + " Started "+left+","+right
12
13         while (true) {
14             // think
15             chairs.acquire();
16             println id + " Eating..."
17             forks[left].acquire();
18             forks[right].acquire();
19             // eat
20             forks[left].release();
21             forks[right].release();
22             println id + " Done Eating..."
23             chairs.release();
24         }
25     }
26 }
```

Dining Philosophers (breaking the symmetry)

```
 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,right;
 9         if (id == 0) {
10             left = 1;
11             right = 0;
12         } else {
13             left = id;
14             right = (id+1) % N;
15         }
16         println id + "Started "+left+","+right
17         while (true) {
18             // think
19             println id + " Eating..."
20             forks[left].acquire();
21             forks[right].acquire();
22             // eat
23             forks[left].release();
24             forks[right].release();
25             println id + " Done Eating..."
26         }
```