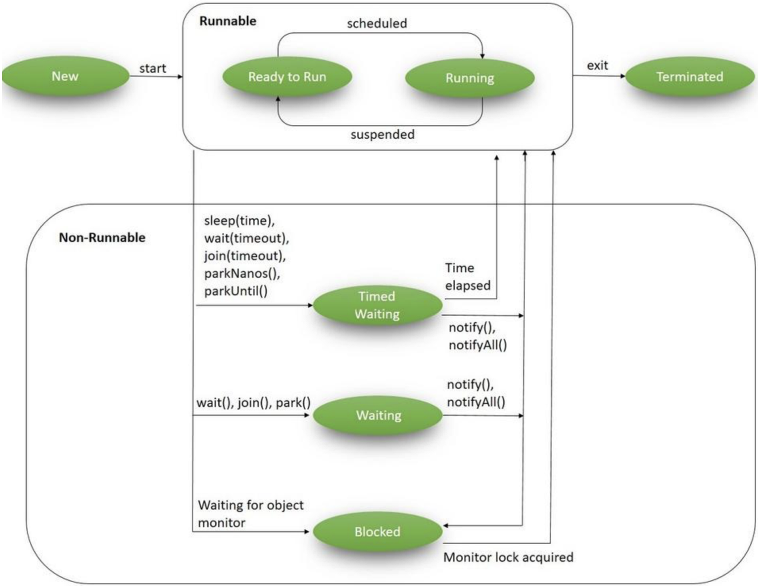
Semaphores

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: integer`
  - `processes: set of processes`

Acquire

`Acquire` consumes a permission, waits if none are available

```
atomic acquire() {
  currentThread = Thread.currentThread();
  if (permissions > 0) {
    permissions--;
  } else {
    processes.add(thread);
    currentThread.state = BLOCKED;
  }
}
```

Release

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

```
atomic release() {
  if (processes.empty() || permissions<0) {
    permissions++;
  } else {
    wakingThread = processes.removeAny();
    wakingThread.state = READY;
  }
}
```

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

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:

```
atomic release() {
  if (permissions == 1) {
    // do nothing
  } else if (processes.empty()) {
    permissions = 1;
  } else {
    wakingThread = processes.removeAny();
    wakingThread.state = READY;
  }
}
```

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()`

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

```
Thread.start { //P
  // non-critical section
  mutex.acquire();
  // critical section
  mutex.release();
  // non-critical section
}
```

```
Thread.start { //Q
  // non-critical section
  mutex.acquire();
  // critical section
  mutex.release();
  // non-critical section
}
```

- 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`

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

```java
/** Acquires a permit from this Semaphore,
    blocking until one is available */
void acquire()
```

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

Semaphores in Java

Class `Semaphore` in `java.util.concurrent`

- `java.util.concurrent.Semaphore`

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

```java
/** Acquires a permit from this Semaphore,
    blocking until one is available */
void acquire()
```

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

Example:

```java
Semaphore mutex = new Semaphore(1);
mutex.acquire()
// critical section
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. $\texttt{permissions} \geq 0$
2. $\texttt{permissions} = k + \texttt{\#releases} - \texttt{\#acquires}$

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. $\texttt{\#criticalSection} + \texttt{permissions} = 1$

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

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

1. $\texttt{\#criticalSection} + \texttt{permissions} = 1$
2. $\texttt{\#criticalSection} = \texttt{\#acquires} - \texttt{\#releases}$

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

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

1. $\texttt{\#criticalSection} + \texttt{permissions} = 1$
2. $\texttt{\#criticalSection} = \texttt{\#acquires} - \texttt{\#releases}$

Item 1 guarantees:

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

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

1. $\texttt{\#criticalSection} + \texttt{permissions} = 1$
2. $\texttt{\#criticalSection} = \texttt{\#acquires} - \texttt{\#releases}$

Item 1 guarantees:

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

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

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

1. $\texttt{\#criticalSection} + \texttt{permissions} = 1$
2. $\texttt{\#criticalSection} = \texttt{\#acquires} - \texttt{\#releases}$

Item 1 guarantees:

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

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

$\texttt{\#criticalSection}$: number of processes in their critical sections

1. $\texttt{\#criticalSection} + \texttt{permissions} = 1$
2. $\texttt{\#criticalSection} = \texttt{\#acquires} - \texttt{\#releases}$

Item 1 guarantees:

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

The Turnstile Problem using Binary Semaphores

```
import java.util.concurrent.Semaphore

Semaphore mutex = new Semaphore(1)
counter=0 // shared variable

P = Thread.start {
      50.times {
        mutex.acquire()
        counter++
        mutex.release()
      }
}
Q = Thread.start {
      50.times {
        mutex.acquire()
        counter++
        mutex.release()
      }
    }

P.join() // wait for P to finish
Q.join() // wait for Q to finish

println(counter) // print value of counter
```

The Turnstile Problem using Binary Semaphores (Java)

```java
public class Turnstile extends Thread {
  static volatile int counter = 0;
  static Semaphore mutex = new Semaphore(1);
  public void run() {
    for(int i = 0; i < 50; i++){
      mutex.acquire();
      counter++;
      mutex.release();
      System.out.println(id+"- In comes: "+i );
    }
  }

  public static void main(String args[]) {
    try{
      Thread m1 = new Turnstile(1);
      m1.start();
      Thread m2 = new Turnstile(2);
      m2.start();
    } catch(Exception e){}
  }
}
```

Counting Example in Java using Semaphores

```java
public class Turnstile extends Thread {
  static volatile int counter = 0;
  ...
```

- 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

```
N.times {
   Thread.start {
     // non-critical section
     mutex.acquire();
     // critical section
     mutex.release();
     // non-critical section
   }
}
```

- 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

```
/** Creates a Semaphore with the given number of permits
    and the given fairness setting. */
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

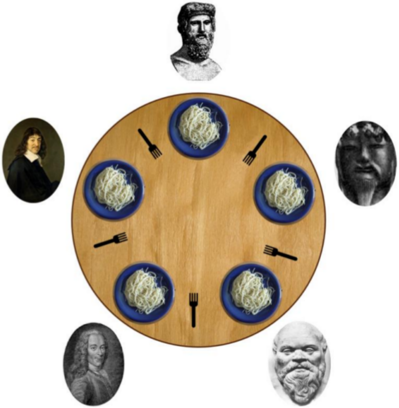
```
import java.util.concurrent.Semaphore;

counter = 0;
mutex = new Semaphore(1);

def turnstile() {
  50.times {
    mutex.acquire();
    counter++;
    mutex.release();
  }
}

2.times {
  Thread.start {
    turnstile();
  }
}

println(counter);
```

What happens when we run this code?

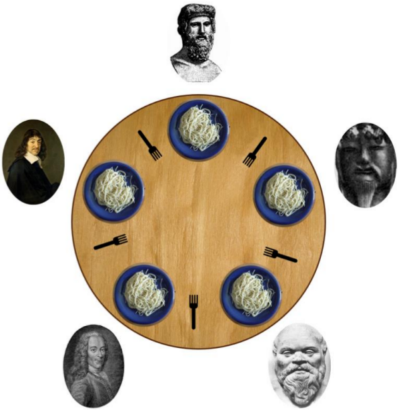
Revisiting the Turnstile Problem

```
import java.util.concurrent.Semaphore;

counter = 0;
mutex = new Semaphore(1);
f = new Semaphore(0);

def turnstile() {
  50.times {
    mutex.acquire();
    counter++;
    mutex.release();
  }
  f.release();
}

2.times {
  Thread.start {
    turnstile();
  }
}

f.acquire();
f.acquire()
println(counter);
```

Dining Philosophers

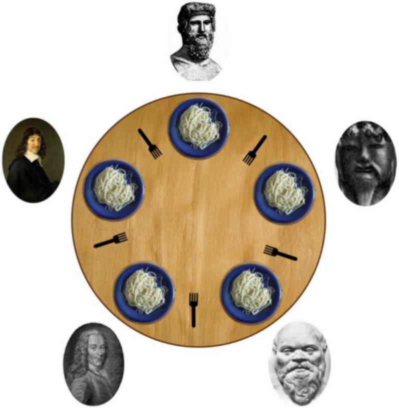


Dining Philosophers



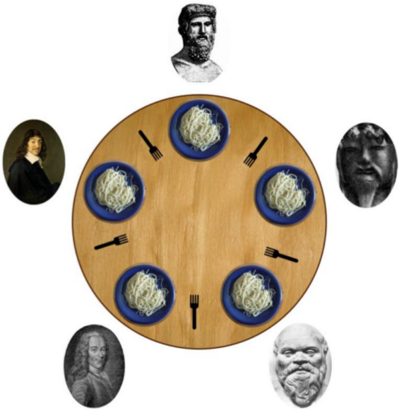
- Philosophors think and eat, in turns

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

```
Thread.start { // Philosopher(id)
  while (true)
    // think
    // pick forks
    // eat
    // leave forks
}
```

Dining Philiosophers

```
Thread.start { // Philosopher(id)
  while (true)
    // think
    // pick forks
    // eat
    // leave forks
}
```

- 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)

```
final int N=5
List<Semaphore> forks=[]; // N
N.times { forks.add(new Semaphore(1)) }

N.times {
    int id = it
    Thread.start {
        int left = id;
        int right = (id+1) % N;
        println id + "Started "+left+","+right

        while (true) {
            // think
            println id + " Eating..."
            forks[left].acquire();
            forks[right].acquire();
            // eat
            forks[left].release();
            forks[right].release();
            println id + " Done Eating..."
        }
    }
}
```

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

Dining Philosophers (general semaphore)

```
final int N=5
List<Semaphore> forks=[]
N.times { forks.add(new Semaphore(1)) }
Semaphore chairs = new Semaphore(N-1)

N.times {
    int id = it
    Thread.start { // Phil(id)
        int left = id;
        int right = (id+1) % N;
        println id + " Started "+left+","+right

        while (true) {
            // think
            chairs.acquire();
            println id + " Eating..."
            forks[left].acquire();
            forks[right].acquire();
            // eat
            forks[left].release();
            forks[right].release();
            println id + " Done Eating..."
            chairs.release();
        }
    }
}
```

Dining Philosophers (breaking the symmetry)

```
final int N=5
List<Semaphore> forks=[]; // N
N.times { forks.add(new Semaphore(1)) }

N.times {
    int id = it
    Thread.start {
        int left,right;
        if (id == 0) {
            left = 1;
            right = 0;
        } else {
            left = id;
            right = (id+1) % N;
        }
        println id + "Started "+left+","+right
        while (true) {
            // think
            println id + " Eating..."
            forks[left].acquire();
            forks[right].acquire();
            // eat
            forks[left].release();
            forks[right].release();
            println id + " Done Eating..."
        }
```