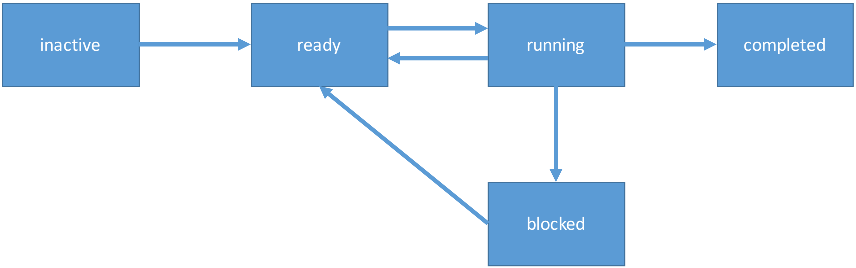
Semaphores

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



- A scheduler decides which of the ready processes it should run
  - Arbitrary interleavings = we assume nothing about the scheduler

Semaphore

A semaphore is an Abstract Data Type with the atomic operations:

- `acquire` (or wait)
- `release` (or signal)

It has two data fields:

- `permissions: non-negative 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)

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

Mutex or Binary Semaphore

- A semaphore that only admits 0 or 1 permissions.
- Initialized to $(0, \emptyset)$ or $(1, \emptyset)$
- The acquire operation is unchanged
- The release operation is now defined as:

```
atomic release() {
  if (permissions == 1) {
    // undefined
  } else if (processes.empty()) {
    permissions = 1;
  } else {
    wakingThread = processes.removeAny();
    wakingThread.state = READY;
  }
}
```

Mutual Exclusion using `mutex`

- We now revisit the critical section problem for two processes
- Using a semaphore `mutex` the solution is trivial
  - P wishes to enter: preprotocol `mutex.acquire()`
  - When P exits: postprotocol `mutex.release()`

Mutual Exclusion using mutex

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

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

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

This solution does not use busy waiting since 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`

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

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

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

Example:

```
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`

1. `permissions` ≥ 0
2. `permissions` $= k +$ `#releases` $-$ `#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 ($k = 1$)

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

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

Item 1 guarantees:

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

```
global int counter = 0;
global Semaphore mutex = new Semaphore(1);

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

repeat (2)
  thread turnstile();
```

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

```
repeat (N) {
   thread {
     // 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 I wish to print the counter total for N turnstiles

```
repeat (N)
  thread turnstile();
  print("Total = " + counter);
```

What happens when I run this code?

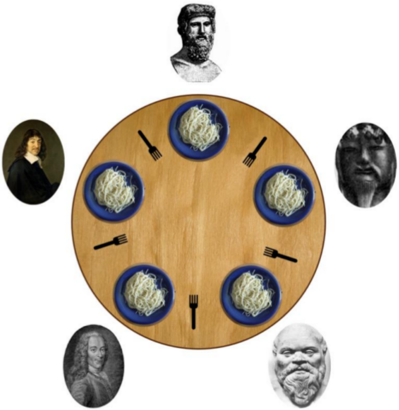
Revisiting the Turnstile Problem

```
global int counter = 0;
global Semaphore mutex = new Semaphore(1);
global Semaphore finish = new Semaphore(-N+1);

turnstile() {
  repeat (100) {
    mutex.acquire();
    counter++;
    mutex.release();
  }
  finish.release();
}

repeat (N)
  thread turnstile();
  finish.acquire();
  print("Total = " + counter);
```

Dining Philiosophers



- 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

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

- Mutex: at any given moment only one philosppher can have a fork
- Synchronization: a philosopher can only eat if she/he has both forks
- Deadlock
- Livelock
- Starvation

Dining Philiosophers (naive attempt)

```
global Semaphore[] forks = [1,...,1]; // N

Philosopher(id) {
  left = id;
  right = (id+1) % N;

  while (true) {
    // think
    forks[left].acquire();
    forks[right].acquire();
    // eat
    forks[left].release();
    forks[right].release();
  }
}
```

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

Dining Philosophers (general semaphore)

```
global Semaphore[] forks = [1,...,1]; // N
global Semaphore chairs = new Semaphore(N-1);

Philosopher(id) {
  left = id;
  right = (id+1) % N;

  while (true) {
    // think
    chairs.acquire();
    forks[left].acquire();
    forks[right].acquire();
    // eat
    forks[left].release();
    forks[right].release();
    chairs.release();
  }
}
```

Dining Philosophers (breaking the symmetry)

```
global Semaphore[] forks = [1,...,1]; // N

Philosopher(id) {
  if (i == 0) {
    left = 1;
    right = 0;
  } else {
    left = id;
    right = (id+1) % N;
  }

  while (true) {
    // think
    forks[left].acquire();
    forks[right].acquire();
    // eat
    forks[left].release();
    forks[right].release();
  }
}
```