Comp 346 – Fall 2019
Tutorial # 4

Synchronization using Semaphores

TOPICS

- Synchronization in Details
- Semaphores
- Introducing `Semaphore.java`

Synchronization

- What is it?
- **An act of communication** between *unrelated* processes to sync their activities to achieve some goals and solve some common problems of multi-programming.

INTRODUCING THE SEMAPHORE CLASS

- NOTE: Operations **`Signal`** and **`Wait`** are ***guaranteed*** to be atomic!

```
class Semaphore
{
      private int value;

      public Semaphore(int value)
      {
            this.value = value;
      }

      public Semaphore()
      {
            this(0);
      }

      ...
```

Introducing the Semaphore Class (Con't)

```
...
public synchronized void Wait()
{
        while(this.value <= 0)
        {
                try
                {
                        wait();
                }
                catch(InterruptedException e)
                {
                 System.out.println
("Semaphore::Wait()
                                   …" + e.getMessage());

                        e.printStackTrace();
                }
        }

        this.value--;
}
 ...
```


Université Concordia University

Introducing the Semaphore Class (con't)

```
        ...
        public synchronized void Signal()
        {
                ++this.value;
                notify();
        }


}
```

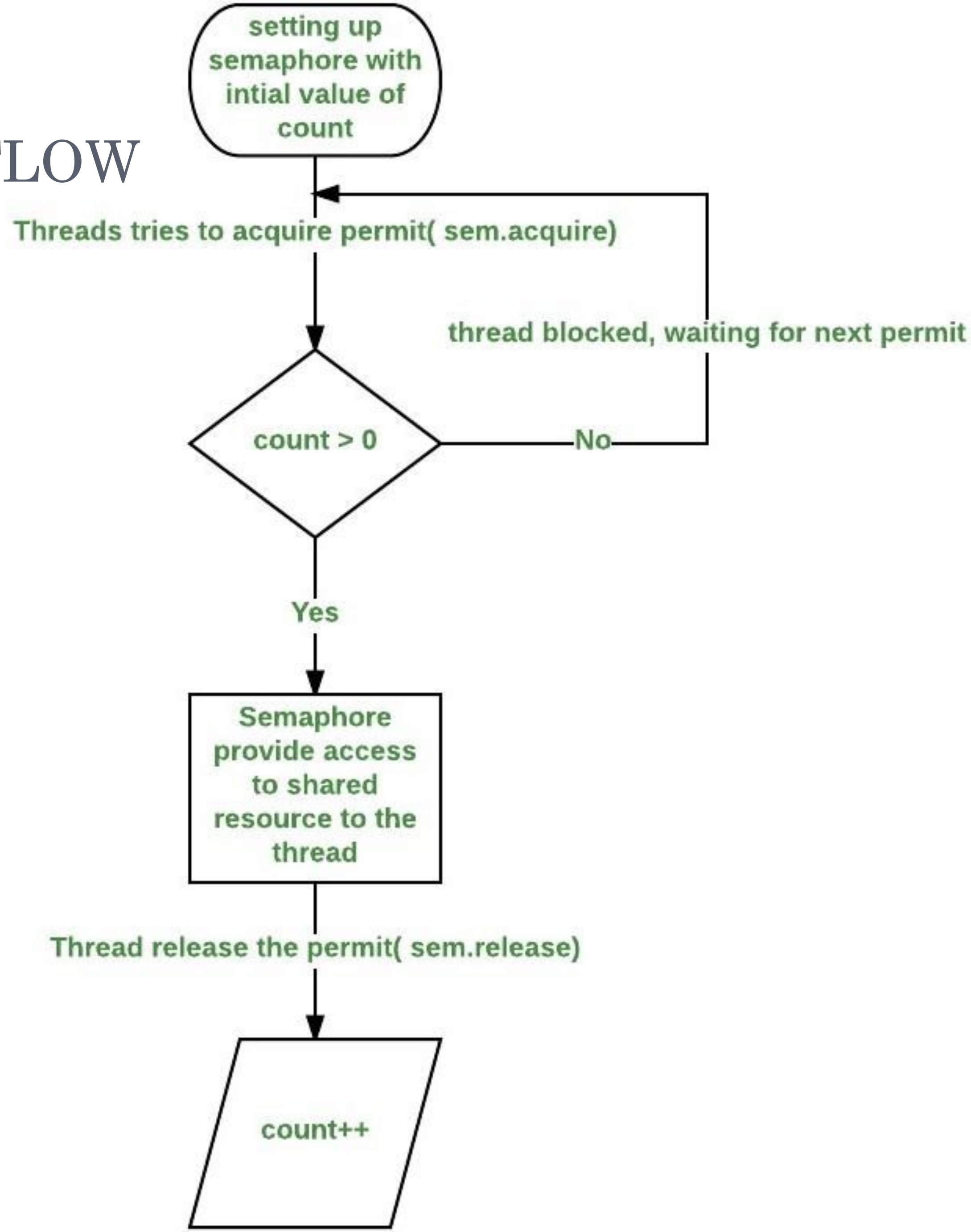
```java
public class Semaphore {
    private int value;
    public Semaphore(){value = 0;}
    public Semaphore(int v){value = v;}

    public synchronized void P( ){
        while( value <= 0){
            try{ wait( );
            } catch(InterruptedException e){ }
        }
                value--;
    }
    public synchronized void V( ) {
        ++value;
        notify( );
    }
}
```


UNIVERSITÉ Concordia UNIVERSITY

Semaphore Initial Values

- The mutex is initialized to 1 to allow only one thread into the critical section at any time.

SEMAPHORE FLOW



What's the Problem?

Example 1:

- A cup of coffee
- A “pourer” (producer)
- A “drinker” (consumer)

```
pourer:
while (true)
{
    pour();
}
```

```
Drinkers:
while (true)
{
    drink();
}
```

Result: A mess!

A Typical Example

```
class John extends Thread {
  run() {
    balance = ATM.getBalance();
    if(balance >= $200)
      ATM.withdraw($200);
  }
}
```

```
class Jane extends Thread {
  run() {
    balance = ATM.getBalance();
    if(balance >= $300)
      ATM.withdraw($300);
  }
}
```

```
class ATM{ …
  int withdraw(amount){
    if(amount <= balance) {
      balance = balance – amount;
      return amount;
  }
}
```

A trouble may occur at the points marked with the red arrows. The code **MUST NOT** be interrupted at those places.

Solution: Use Semaphores

- Obvious: make the critical section part **atomic**.
- One way of doing it: **Semaphores**
- Semaphores are **system-wide** OS objects (also resources) used to
  - Protect critical section (mutexes – for Mutual Exclusion),
  - Coordinate other process' activities.

Semaphores for CS

- There are two main operations on semaphores – **`Wait()`** and **`Signal()`**.
- A process wishing to enter the critical section tries to acquire the semaphore (a lock in a human world) by calling `Wait(sem)`,**`P()`**.
  - If the lock isn't there (i.e. "in use"), the execution of the process calling **`Wait()`** is suspended (put asleep).
  - Otherwise, it acquires the semaphore and does the critical section stuff.

SEMAPHORES FOR CS

- When a process is over with the critical section, it notifies the rest of the processes waiting on the same semaphore that they can go in by calling `Signal(sem)`,**`V()`**.
- The process goes back to the ready queue to compete again to enter the critical section.

Multiple drinkers? No problem!

- Using semaphores, we can make sure that only one drinker can ever drink at one time

```
Main:

// Semaphore initialization
Semaphore cup;
cup = new Semaphore(1);
```

```
Drinker i:

while (true)
{ cup.Wait();
 drink();
 cup.Signal();
}
```

- Important: Wait() and Signal() are guaranteed to be **atomic**!

A Typical Example Solution

```
class John extends Thread {
  run() {
    mutex.Wait();
    balance = ATM.getBalance();
    if(balance >= $200)
       ATM.withdraw($200);
    mutex.Signal();
  }
}
```

```
class Jane extends Thread {
  run() {
    mutex.Wait();
    balance = ATM.getBalance();
    if(balance >= $300)
       ATM.withdraw($300);
    mutex.Signal();
  }
}
```

```
class ATM{ …
  Semaphore mutex = 1;
  int withdraw(amount) {
     if(amount <= balance) {
       balance = balance – amount;
       return amount;
  }
}
```

Université Concordia University

SEMAPHORES FOR BARRIER SYNC

- Take a look at the typical problem:

```
semaphore s1 = 0, s2 = 0;
process P1                process P2
  <phase I>                 <phase I>
  V (s1)                    V (s2)
  P (s2)                    P (s1)
  <phase II>                <phase II>
```

  - all processes must finish their phase I before any of them starts phase II
  - processes must proceed to their phase II in a specific order, for example: 1, 2, 3…
- This is called **barrier synchronization**.

UNIVERSITÉ Concordia UNIVERSITY

Barrier Sync for Three Processes

- A possible copy-cat attempt:

```
semaphore s1 = 0, s2 = 0, s3 = 0;
process P1      process P2        process P3
  <phase I>       <phase I>         <phase I>
  V (s1)          V (s2)            V (s3)
  P (s3)          P (s1)            P (s2)
  <phase II>      <phase II>        <phase II>
```

- Why it doesn’t work?

Barrier Sync for Three Processes

- A possible copy-cat attempt:

```
  semaphore s1 = 0, s2 = 0, s3 = 0;
  process P1       process P2        process P3
3   <phase I>        <phase I>     1    <phase I>
4   V (s1)           V (s2)        2    V (s3)
5   P (s3)           P (s1)             P (s2)
6   <phase II>       <phase II>         <phase II>
```

- Why it doesn't work?
  - Scenario: 1-6 , process 2 even hasn't started!
  - **None** of the requirements are met

Barrier Sync for Three Processes (2)

- Another attempt:

```
semaphore s1 = 0, s2 = 0, s3 = 0;
process P1          process P2          process P3
  <phase I>          <phase I>           <phase I>
  P (s1)              V (s1)              V (s1)
  P (s1)              P (s2)              P (s3)
  V (s2)              V (s3)
 <phase II>          <phase II>          <phase II>
```

- What's wrong now?

Barrier Sync for Three Processes (2)

- Another attempt:

```
semaphore s1 = 0, s2 = 0, s3 = 0;
 process P1        process P2        process P3
7  <phase I>     4  <phase I>     1  <phase I>
8  P (s1)        5  V (s1)        2  V (s1)
9  P (s1)        6  P (s2)        3  P (s3)
10 V (s2)           V (s3)
   <phase II>       <phase II>       <phase II>
```

- What's wrong now?
  - Scenario: 1-10, so far so good, but after…
  - The second requirement isn't met

Barrier Sync for Three Processes (3)

- Last attempt:

```
semaphore s1 = 0, s2 = 0, s3 = 0;
process P1        process P2          process P3
  <phase I>         <phase I>           <phase I>
  P (s1)            V (s1)              V (s1)
  P (s1)            P (s2)              P (s3)
  <phase II>        <phase II>          <phase II>
  V (s2)            V (s3)
```

- A bit "optimized":

```
semaphore s1 = -1, s2 = 0, s3 = 0;
process P1        process P2          process P3
  <phase I>         <phase I>           <phase I>
                    V (s1)              V (s1)
  P (s1)            P (s2)              P (s3)
  <phase II>        <phase II>          <phase II>
  V (s2)            V (s3)
```

Barrier Sync: Need for the General Solution

- Problem with the proposed solution: # of semaphores == # of processes.
- Semaphores as any other resource are limited and take space => overhead
- Imagine you need to sync 10 processes in the same manner? 100? 1000?
  - Complete mess and a high possibility of a deadlock!

Barrier Sync: Need for the General Solution (2)

- Attempt for the first requirement:

```
semaphore s1 = -n + 2, s2 = 0;
process P1        process P2    ...    process Pn
  <phase I>         <phase I>   ...      <phase I>
  P (s1)            V (s1)      ...      V (s1)
  V (s2)            P (s2)      ...      P (s2)
                    V (s2)      ...      V (s2)
  <phase II>        <phase II>  ...      <phase II>
```

- The second requirement is left as an exercise to the curious student :-)

Counting vs Binary Semaphores

- Counting Semaphores allow arbitrary resource count (semaphore values that can be <0 and >1)
- Binary Semaphores only allow two values: 0 and 1

Q1

- Sometimes it is necessary to synchronize two or more processes so that all processes must finish their first phase before any of them is allowed to start its second phase. This is called Barrier synchronization.
- For two processes, we might write:

```
process P1 {              process P2 {
<phase I>                   <phase I>
V (s1)                    V (s2)
P (s2)                    P (s1)
<phase II>                 <phase II>
          }                          }
```

- Give a solution to the problem for three processes P1, P2, and P3 using 3 semaphores.

UNIVERSITÉ Concordia UNIVERSITY

POSSIBLE S1

```
semaphore s1 = 0, s2 = 0, s3=0;
process P1 {
<phase I>
V (s1)
P (s2)
P(s3)
<phase II>
}

process P2 {
<phase I>
V (s2)
P(s1)
P (s3)
<phase II>
}

process P3 {
<phase I>
V(s3)
P(s1)
P(s2)
<phase II>
}
```

This solution still has a fatal error! Can you find it?

Hint: are phase II's of all processes guaranteed to be executed?

Revised S1

semaphore s1 = 0, s2 = 0, s3 = 0

Process 1	Process 2	Process 3
<phase I>	<phase I>	<phase I>
V(s1)	V(s2)	V(s2)
P(s2)	P(s1)	P(s3)
P(s2)	P(s1)	
<phase II>	<phase II>	<phase II>
V(s1)	V(s3)	

Q2

- Consider the following solution to the producer and consumer problem. Does the solution respect the critical section requirements? Explain.

Producer : …
<produce a resource>
wait(mutex);
wait(empty);
<deposit a resource in buffer>
signal(full);
signal(mutex);
<remainder section>

Consumer : …
wait(mutex);
wait(full);
<remove a resource from buffer>
signal(empty);
signal(mutex);
<consume a resource>
<remainder section>

UNIVERSITÉ Concordia UNIVERSITY

S2

- *Three critical sections,*
  - *one to access the buffer (synchronized by mutex),*
  - *one to access an empty buffer slot (synchronized by empty),*
  - *and one to access a full buffer slot (synchronized by full).*
- *Progress requirement is violated because a process that is blocked on an empty or full semaphore will block the other process because it had previously locked the mutex semaphore.*
- *For example, if the producer blocks on an empty semaphore it will prevent the consumer to access an available full buffer.*
- *Consequently a deadlock situation (causing indefinite waiting) will occur because the producer and consumer processes will be both blocked waiting for an event that may be caused only by one of them.*

References

- [1]http://users.encs.concordia.ca/~mokhov/comp346/
- [2]http://programmingexamples.wikidot.com/java-barrier
- [3]http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html
- [4]Sample question are from, Theory assignment 2, COMP 346, Operating Systems, Kerly Titus

UNIVERSITÉ Concordia UNIVERSITY