Concurrency 2
Shared Memory

Catuscia Palamidessi
INRIA Futurs and LIX - Ecole Polytechnique

The other lecturers for this course:

Jean-Jacques Lévy (INRIA Rocquencourt)
James Leifer (INRIA Rocquencourt)
Eric Goubault (CEA)

http://pauillac.inria.fr/~leifer/teaching/mpri-concurrency-2005/
Outline

1. Solution to some of the exercises in previous lecture
   - Semaphores in Java
   - Readers and Writers

2. Verification of Concurrent Software (by Jean-Jacques Lévy)
   - A case study: Ariane
A thread is a single sequential line of control. It may be execute in parallel/interleaving with other threads.

The states of a live thread in Java:
A few facts about Java (2/2)

Classes with synchronized methods

- Class whose objects may be shared by different threads need synchronized methods.
- Example: A bank account with two or more owners.

```java
class Account {
    private int balance;
    public Account(int initialDeposit) {
        balance = initialDeposit;
    }
    public synchronized void deposit(int amount) {
        balance = balance + amount;
    }
    ...
}
```

- Synchronized methods are handled using a lock mechanism. *A lock is per object.*
- When a thread suspends inside a synchronized method, it releases the lock.
A generalized semaphore $s$ is an integer variable with two operations:

- **acquire($s$):** If $s > 0$ then $s := s - 1$, otherwise suspend on $s$. (atomically)

- **release($s$):** If some process is suspended on $s$, wake it up, otherwise $s := s + 1$. (atomically)

Example of use: At beginning, $s = \text{max}$. Then

$$[	ext{⋯;} \text{acquire} (s); C_1; \text{release} (s); \text{⋯}] \parallel [\text{⋯;} \text{acquire} (s); C_2; \text{release} (s); \text{⋯}]$$
Use of a semaphore in Java

Creation of a Semaphore s
sSemaphore(max);

Thread 1
...
  s.acquire();
  C_1;
  s.release();
...

Thread 2
...
  s.acquire();
  C_2;
  s.release();
...

Semaphores in Java

Declaration of class Semaphore in Java

Use \textit{sus} to indicate the number of suspended threads on the semaphore

```java
class Semaphore {
    private int value, sus;
    public Semaphore(int initial) {
        value = initial; sus = 0;
    }
    public synchronized void acquire() {
        if (value == 0) { sus = sus + 1; wait(); sus = sus - 1; }
        else value = value - 1;
    }
    public synchronized void release() {
        if (sus > 0) { notify(); }
        else { value = value + 1; }
    }
}
```

However, this is not efficient (\textit{why?}) and it is not in the typical “Java style”.
Semaphore in Java (typical Java solution)

```java
class Semaphore {
    private int value;
    public Semaphore(int initial) {
        value = initial;
    }
    public synchronized void acquire() {
        while (value == 0) wait();
        value = value - 1;
    }
    public synchronized void release() {
        value = value + 1;
        notify();
    }
}
```
Problem: A certain resource (for instance a file) is shared by some readers and some writers. The readers cannot modify the resource, while the writers can.

We want that only one writer can access the resource at a time, while the readers are allowed to do it concurrently.
Readers and Writers

Readers and Writers in Java

**Reader**

```java
... 
r.acquireShared();
use r;
r.releaseShared();
... 
```

**Writer**

```java
... 
r.acquireExclusive();
use r;
r.releaseExclusive();
... 
```
Readers and Writers

The class Resource

class Resource {
    private int readers, writers;
    public Resource() {
        readers = 0;
        writers = 0;
    }
    public synchronized void acquireShared() { ... }
    public synchronized void releaseShared() { ... }
    public synchronized void acquireExclusive() { ... }
    public synchronized void releaseExclusive() { ... }
}
Readers and Writers

The methods of Resource

```java
acquireShared()
{
    while (writers == 1) {
        wait();
    }
    readers = readers + 1;
}
```

```java
releaseShared()
{
    readers = readers - 1;
    notify();
}
```

```java
acquireExclusive()
{
    while (writers == 1 || readers > 0) {
        wait();
    }
    writers = 1;
}
```

```java
releaseExclusive()
{
    writers = 0;
    notifyAll();
}
```

However, this solution is not efficient. (Why?)
Readers and Writers

A more efficient solution

- Use suspension conditions $cR$, $cW$
- Use $sR$ to indicate the number of readers suspended.

```java
acquireShared()
{
    while (writers == 1) {
        sR = sR + 1;
        wait(cR);
        sR = sR - 1;
    }
    readers = readers + 1;
}

releaseShared()
{
    readers = readers - 1;
    notify(cW);
}

acquireExclusive()
{
    while (writers == 1 || readers > 0) {
        wait(cW);
    }
    writers = 1;
}

releaseExclusive()
{
    writers = 0;
    if (sR > 0) { notifyAll(cR); }
    else { notify(cW); }
}
```
The "more efficient solution" for the Readers and Writers problem that we presented in this lecture is not starvation-free, because it always gives priority to the readers. Modify the solution so to ensure that neither the writers nor the readers will starve.

About the first solution we presented for the Readers and Writers problem: it that one starvation-free? Justify your answer.
A case study: Ariane