Concurency 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-2003/
Definition of Semaphore (from previous lecture)

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, \( s = \max \). Then
\[
\texttt{\{ \cdots \texttt{acquire}(s); C_1; \texttt{release}(s); \cdots \} || \texttt{\{ \cdots \texttt{acquire}(s); C_2; \texttt{release}(s); \cdots \}}\}
\]

Use *sus* to indicate the number of suspended threads on the semaphore

```
Semaphore

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

However, this is not efficient (why?) and it is not in the typical “Java style”
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.

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() { ... }
}

The methods of Resource

acquireShared()

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

acquireExclusive()

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

releaseShared()

    {
        notify();
    }

releaseExclusive()

    {
        writers = 0;
        notifyAll();
    }

However, this solution is not efficient. (Why?)
A more efficient solution

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

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

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

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

```c
releaseExclusive()
{
    writers = 0;
    if (sR > 0) { notifyAll(cR); }
    else { notify(cW); }
}
```

---

Exercises

- 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.