Condition Synchronisation III & Life in Deadlock I

Lecture 11

Alma Rahat

CS-210: Concurrency

02 March 2021

What did we do in the last session?

- Car park model and code.
- `wait()`, `notify()`, and `notifyAll()`.
- A library example with `wait()-notify()`.

In this lecture...

Learning outcomes.

1. To apply Semaphores for condition synchronisation.
2. To describe deadlocks between processes.
3. To explain the necessary and sufficient conditions for deadlock.
4. To understand and apply Coffman conditions in order to analyse deadlocks in given scenario.

Outline.

1. A library example with `Semaphore`.
2. Deadlock:
   - Dining Philosophers' Problem.
   - Necessary and sufficient conditions for deadlock.

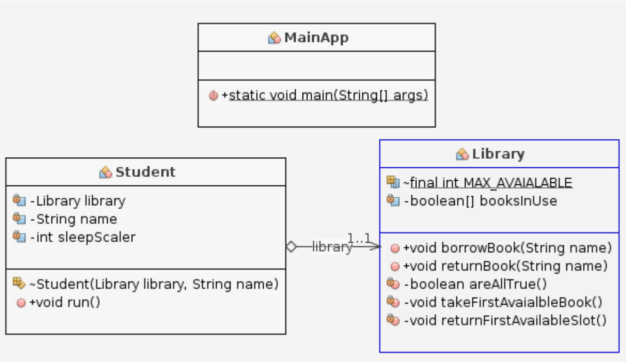
A Library Example: Quick Recap

. . .


Our library has *n* copies of the concurrency book for the course. A student can borrow a copy, read it, and then return it. A record is kept of which slot in the shelf has been emptied. Condition is that no one can take a book when there are none, and hence must wait for one to be returned.

We can use `wait()`→`notifyAll()` structure, so that we are allowing non-blocking access to a number of individuals.

A Library Example: Quick Recap



Library is the passive process here, and therefore the monitor class. Student is implementing the *Runnable* interface as the active process.

A Library Example: Quick Recap

```
public class Library{
    static final int MAX_AVAIALABLE = 2;
    private boolean[] booksInUse = new boolean[MAX_AVAIALABLE];
    public synchronized void borrowBook(String name)
            throws InterruptedException{
        while(areAllTrue()){
            System.out.println(name + " is waiting!");
            wait();
        }
        System.out.println(name + " is trying to get a book!");
        takeFirstAvaialbleBook();
        notifyAll();
    }
    private boolean areAllTrue(){
        for(boolean b : booksInUse) if(!b) return false;
        return true;
    }
    private void takeFirstAvaialbleBook() {
        for(int i=0; i<MAX_AVAIALABLE; i++)
            if (booksInUse[i] == false){
                booksInUse[i] = true;
                break;
            }
    }
```

A Library Example: Quick Recap

```java
    public synchronized void returnBook(String name)
            throws InterruptedException{
        System.out.println(name + " is trying to return a book!");
        returnFirstAvailableSlot();
        notifyAll();
    }
    private void returnFirstAvailableSlot() {
        for(int i=0; i<MAX_AVAIALABLE; i++)
            if (booksInUse[i] == true){
                booksInUse[i] = false;
                break;
            }
    }
}
```

A Library Example: Quick Recap

```java
public class Student implements Runnable{
    private Library library;
    private String name;
    private int sleepScaler = 10000;
    Student(Library library, String name){
        this.library = library;
        this.name = name;
    }
    @Override
    public void run() {
        try {
            library.borrowBook(name);
            double random = Math.random();
            System.out.println(name + " is starting to read.");
            Thread.sleep((long) (random*sleepScaler));
            System.out.println(name + " has finisehd reading.");
            library.returnBook(name);
        } catch (InterruptedException ex) {
            System.out.println("Interrupted Arrival Thread");
            return;
        }
    }
}
```

A Library Example: Quick Recap

```
public class MainApp {
    public static void main(String[] args)
            throws InterruptedException {
        Library library = new Library();
        Student s1 = new Student(library, "Tom");
        Student s2 = new Student(library, "Jenny");
        Student s3 = new Student(library, "Plato");
        Thread t1 = new Thread(s1);
        Thread t2 = new Thread(s2);
        Thread t3 = new Thread(s3);
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();
    }
}
```

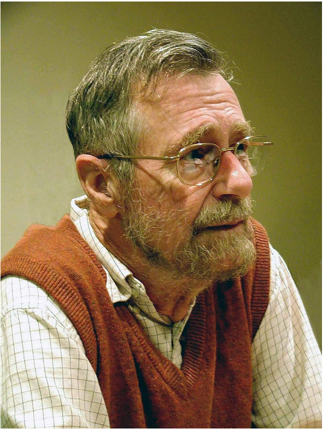
A Library Example: Quick Recap

```
run:
Tom is trying to get a book!
Plato is trying to get a book!
Jenny is waiting!
Plato is starting to read.
Tom is starting to read.
Plato has finisehd reading.
Plato is trying to return a book!
Jenny is trying to get a book!
Jenny is starting to read.
Tom has finisehd reading.
Tom is trying to return a book!
Jenny has finisehd reading.
Jenny is trying to return a book!
BUILD SUCCESSFUL (total time: 12 seconds)
```

Tom and Plato gets a book from the shelf, but Jenny keep waiting until Plato returns his book.

Any questions?

Semaphores



Dijkstra (source: wikipedia)

Semaphore, introduced by Dijkstra in 1968, is one of the first concepts proposed to deal with inter-process synchronisation.

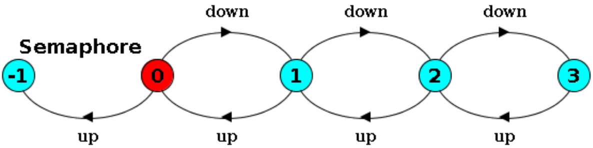
A semaphore s is a counting variable that can only take positive integer values. Once s has been given an initial value, the only allowed operations are `up(s)` and `down(s)` defined as:

- down(s): when $s > 0$ decrement s.
- up(s): increment s.

Essentially, we block processes from accessing shared resources when $s == 0$: a form of acquisition operation with each `down(s)`. To release resource, we will perform the `up(s)` action.

Modelling Semaphore

```
const N = 3
range T = 0..N
Semaphore(Capacity=N) = Semaphore[Capacity],
Semaphore[v:T] = (when v>0 down -> Semaphore[v-1]
| up -> Semaphore[v+1]).
```



Library Implementation with Semaphore

```java
private Semaphore semaphore = new Semaphore(MAX_AVAIALABLE);
public void borrowBook(String name) throws InterruptedException{
    semaphore.acquire(); ✓
    System.out.println(name + " is trying to get a book!");
    takeFirstAvaialbleBook();
}
private synchronized void takeFirstAvaialbleBook() {
    for(int i=0; i<MAX_AVAIALABLE; i++)
        if (booksInUse[i] == false){
            booksInUse[i] = true;
            break;
        }
}
public void returnBook(String name) throws InterruptedException{
    System.out.println(name + " is trying to return a book!");
    returnFirstAvailableSlot();
    semaphore.release(); ✓
}
private synchronized void returnFirstAvailableSlot() {
    for(int i=0; i<MAX_AVAIALABLE; i++)
        if (booksInUse[i] == true){
            booksInUse[i] = false;
            break;
        }
}
```

`import java.util.concurrent.Semaphore`

Library Implementation with Semaphore

```
Jenny is trying to get a book!
Tom is trying to get a book!
Jenny is starting to read.
Tom is starting to read.
Jenny has finisehd reading.
Jenny is trying to return a book!
Plato is trying to get a book!
Plato is starting to read.
Plato has finisehd reading.
Plato is trying to return a book!
Tom has finisehd reading.
Tom is trying to return a book!
BUILD SUCCESSFUL (total time: 9 seconds)
```

Here, Plato only gets access to the critical section once Jenny has incremented the counter.

Any questions?

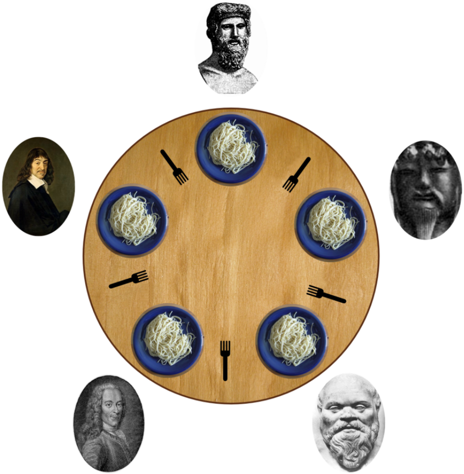
Dining Philosophers' Problem

source: wikipedia

Five philosophers are dining together, and they are sharing five forks to eat spaghetti. A philosopher arbitrarily sits down to eat, and then picks up the right fork first and then the left fork. This will lead to a deadlock situation, as in no progress can be made. How?

Please go to `www.menti.com` and use the code 45 15 57.

Dining Philosophers' Problem



source: wikipedia

Five philosophers are dining together, and they are sharing five forks to eat spaghetti. A philosopher arbitrarily sits down to eat, and then picks up the right fork first and then the left fork. This will lead to a deadlock situation, as in no progress can be made. How?

Everyone waits for the left fork to become available in the following scenario:
$P_1 \rightarrow$ right fork $\rightarrow P_2 \rightarrow$ right fork $\rightarrow P_3 \rightarrow$ right fork $\rightarrow P_4 \rightarrow$ right fork $\rightarrow P_5 \rightarrow$ right fork

Necessary and Sufficient Condition

For deadlock to occur, all of the following must occur.

1. Serially reusable resources: the processes involved shared resources which they use under **mutual exclusion**. [Philosophers share forks.]
2. Incremental acquisition: processes **hold** on to resources already allocated to them while **waiting** to acquire additional resources. [If holds right fork, waits for left to become available.]
3. **No pre-emption**: once acquired by a process resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily. [Philosopher is responsible for letting go of the fork.]
4. Wait-for-cycle: a **circular** chain (or cycle) of processes holds a resource which its successor in the cycle is waiting to acquire. [One waits for another to release fork and *vice-versa*.]

These conditions were first proposed by Coffman in 1971. This is why they are often referred to as the Coffman conditions.

Any questions?

Summary

- Wait-notify, and semaphore are Java language features that allow us to do condition synchronisation.
- Deadlock occurs when we cannot make progress, i.e. all threads are running, but no one is able to progress.