3C03 Concurrency: Semaphores and Monitors

Mark Handley

Goals

- Introduce concepts of
  - Semaphores
  - Monitors

- Implementation in Java
  - synchronised methods and private attributes
  - single thread active in the monitor at any time
  - wait, notify and notifyAll
Semaphores

Introduced by Dijkstra in 1968
ADT with counter and waiting list
Down and Up need to be atomic.

P/Wait/Down:
if (counter > 0)
counter--
else
add caller to waiting list

S/Signal/Up:
if (threads wait)
activate waiting thread
else
counter++

Semaphores and Mutual Exclusion

- One semaphore for each critical section
- Initialize semaphore to 1.
- Embed critical sections in wait/signal pair

Example in Java:
```
Semaphore S=new Semaphore(1);
S.down();
<critical section>
S.up();
```

Demo: Semaphores
Semaphore in Java

class Semaphore {
    private int value_;
    Semaphore (int initial) {
        value_ = initial;
    }

    synchronized public void up() {
        ++value_;
        notify();
    }

    synchronized public void down() {
        while (value_ == 0) {
            try {wait();} catch(InterruptedException e){}
        }
        --value_;
    }
}

Evaluation of Semaphores

+ Nice and simple mechanism
+ Can be efficiently implemented
+ Available in every programming language
Evaluation of Semaphores (2)

- **Too low level of abstraction**
- **Unstructured use of wait and signal leads to spaghetti synchronisation.**
- **Error prone, and errors are dangerous:**
  
  Correct:
  
  ```
  wait(S); {critical section}; signal(S);
  ```

  **Deadlock due to:**
  
  Omitting signal.
  
  ```
  wait(S); {critical section}; wait(S);
  ```

  **Safety violations due to:**
  
  Omitting wait.
  
  Reverse: signal(S); {critical section}; wait(S);

Potential Solutions

- **How do we protect ourselves from these kinds of errors?**

- **Develop language constructs that can be validated automatically by the compiler or run-time environment.**
  - **Monitors**

- **Guarantee mutual exclusion by definition**
  - **Example: Java synchronised method**
Monitors

- Hoare’s response to Dijkstra’s semaphores
  - Higher-level
  - Structured
- Monitors encapsulate data structures that are not externally accessible
  - Mutual exclusive access to data structure enforced by compiler or language run-time.

Monitors in Java

- To implement a monitor in Java:
  - All instance and class variables need to be private or protected
  - All methods need to be synchronized

- Example: semaphore implementation.
- Use of Monitors: Carpark Problem
Car Park Problem

- Only admit cars if car park is not full
- Cars can only leave if car park is not empty
- Car arrival and departure are independent

Demo: CarPark

Car Park Model

- Events or actions of interest:
  - Arrive and depart
- Processes:
  - Arrivals, departures and car park control

Process and Interaction structure:
Carpark FSP Specification

CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] =
    (when(i>0) arrive-> SPACES[i-1]
     |when(i<N) depart-> SPACES[i+1]
     )

ARRIVALS = (arrive-> ARRIVALS).

DEPARTURES = (depart-> DEPARTURES).

||CARPARK =
    (ARRIVALS|CARPARKCONTROL|DEPARTURES).

Translating Car Park into Java

- **Car Arrival** is an active process that can happen independently.
  - Implement as a Java Thread.

- **Car Departure** is an active process that can happen independently.
  - Implement as a Java Thread

- **Car Park Control** is a passive process that merely controls when arrival and departure can occur.
  - Implement as a Monitor.
Java Class Carpark

```java
public class Carpark extends Applet {
    final static int N=4;

    public void init() {
        CarParkControl cpk = new CarParkControl(N);
        Thread arrival,departures;
        arrivals = new Thread(new Arrivals(cpk));
        departures = new Thread(new Departures(cpk));
        arrivals.start();
        departures.start();
    }
}
```

Java Classes Arrivals & Departures

```java
public class Arrivals implements Runnable {
    CarParkControl carpark;
    Arrivals(CarParkControl c) {carpark = c;}
    public void run() {
        while (true) carpark.arrive();
    }
}
```
```java
class Departures implements Runnable {
    ...  
    public void run() {
        while (true) carpark.depart();
    }
}
```
Java Class CarParkControl (Monitor)

class CarParkControl {// synchronisation?
    private int spaces;
    private int N;
    CarParkControl(int capacity) {
        N = capacity;
        spaces = capacity;
    }
    synchronized public void arrive() {
        ... -- spaces; ... } {// Block if full?
    synchronized public void depart() {
        ... ++ spaces; ... } {// Block if empty?
    }
}

Problems with CarParkControl

- How do we send arrivals to sleep if car park is full?
- How do we awake it if space becomes available?
- How do we send departures to sleep if car park is empty?
- How do we awake it if car becomes available to depart?

- Solution: Condition synchronisation.
Summary

- **Semaphores**
- **Monitors**

**Next session:**
- *Java condition synchronization*
- *Relationship between FSP guarded actions and condition synchronization*
- *Fairness and Starvation*