



C340 Concurrency: Semaphores and Monitors

Wolfgang Emmerich



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*

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



Evaluation of Semaphores

- + *Nice and simple mechanism*
- + *Can be efficiently implemented*
- + *Available in every programming language*
- *Too low level of abstraction*
- *Unstructured use of signal and wait leads to spaghetti synchronisation*
- *Error prone and errors are dangerous*
 - *Omitting signal leads to deadlocks*
 - *Omitting wait leads to safety violations*



© Wolfgang Emmerich, 1999

5



```
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_;
    }
}
```

© Wolfgang Emmerich, 1999

6



Critical Regions

- *Guarantee mutual exclusion by definition*
- *Note subtle difference to critical sections*
- *language features implement critical regions*
- *Example: Java synchronised method*

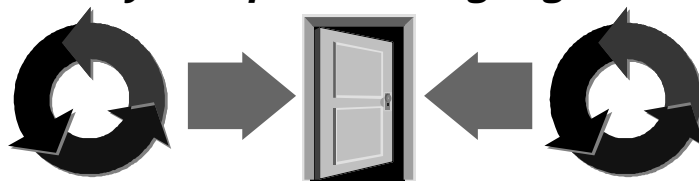
© Wolfgang Emmerich, 1999

7



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*



© Wolfgang Emmerich, 1999

8



Monitors in Java

- ***All instance and class variables need to be private or protected***
- ***All methods need to be synchronised***
- ***Example: semaphore implementation***
- ***Use of Monitors: Carpark Problem***



Carpark Problem

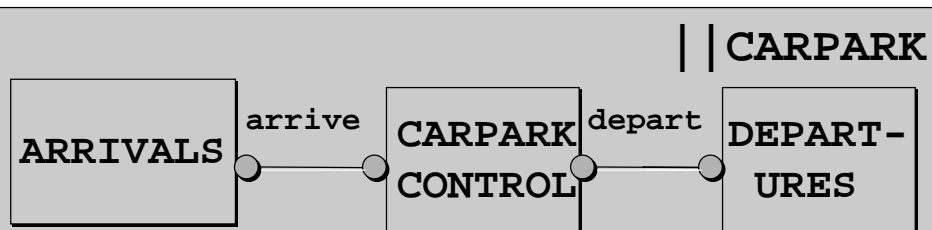
- ***Only admit cars if carpark is not full***
- ***Cars can only leave if carpark is not empty***
- ***Car arrival and departure are independent threads***

Demo: CarPark



Carpark Model

- **Events or actions of interest:**
 - Arrive and depart
- **Processes:**
 - Arrivals, departures and carpark control
- **Process and Interaction structure:**



© Wolfgang Emmerich, 1999

11



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).
  
```

LTSA

© Wolfgang Emmerich, 1999

12



Java Class Carpark

```
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();
    }
}
```

© Wolfgang Emmerich, 1999

13



Java Classes Arrivals & Departures

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

© Wolfgang Emmerich, 1999

14



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?
    }
}
```

© Wolfgang Emmerich, 1999

15



Problems with CarParkControl

- *How do we send arrivals to sleep if car park is full?*
- *How do we awake it if space becomes available?*
- **Solution: Condition synchronisation**

© Wolfgang Emmerich, 1999

16



Summary

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