COMP31212: Concurrency
Topics 4.1: Concurrency Patterns - Monitors
Outline

**Topic 4.1: Concurrency Patterns: Monitors**

- Monitors
- FSP Models-to-Java Monitors
- Producers/Consumers
Monitors

Monitors are data structures with an interface which allows controlled access for multiple processes.

Edsger Dijkstra

Per Brinch Hansen

Sir Tony Hoare
Papers

Monitors: access control

For **sequential systems** access is controlled through an interface of function names/procedures/methods and their types.

For **concurrent systems** we need sequential access control, plus:

- Restrictions on the numbers and kinds of processes that can have simultaneous access to the data, so called “mutual exclusion”, and
- Job scheduling - determining which processes have access and when, using queues, activations, and priorities.

Monitors are both an implementation technique and also (in some languages) a syntactic construct which allows secure multiprogramming with simple proof rules.
Monitor scheme

Active elements

Access control: Mutual exclusion and process scheduling

Passive element

Data structure
Variety of monitor schemes

There is a variety of monitor schemes, depending on (1) how processes requesting access are organised, (2) how ‘conditional waits’ and process suspension are handled, (3) how processes are reactivated after suspension, and (4) how process access is prioritised.

- Some schemes have a **single queue** of processes requesting access to a data structure, combining those that request access with those that have suspended processing.
- Some schemes have **two queues** - new requests, and another for all suspended processes, and then new accesses have to choose between them.
- Some schemes allow explicit “wait on a condition”. In this case all those waiting on a given (named) condition may be put in the same queue, so there are **many queues of suspended processes**, one for each condition.

How are processes **reactivated**? Several schemes are available.
Monitors and Java implementation

Java does *not* have explicit monitors (in this sense), but does allow us to implement monitors (this is seen, by some, as a failure in language design).

- Passive shared objects, methods invoked by (active) threads
- synchronized methods to exclusively access private variables.
- *Condition Synchronization*: guarded execution:
  - looped guard-test containing `wait()`
  - `notify()`/`notifyAll()` after guard-state change
class Monitor
{
    private int x = 0;

    synchronized
    void addOne(int n, int delay) throws InterruptedException
    {
        System.out.println("Thread "+n+": x is "+x);
        int t = x + 1;
        System.out.println("Thread "+n+": t is "+t);
        Thread.sleep(delay);
        x = t;
        System.out.println("Thread "+n+": x is "+x);
    }
}
Synchronisation - II

class T extends Thread
{
    Monitor obj;

    private int name, threadDelay;

    T(Monitor o, int n, int delay){
        obj = o;
        name = n;
        threadDelay = delay; }

    public void run(){
        try {obj.addOne(name, threadDelay); } } catch (InterruptedException e) {} }
public class MonitorExample {

    public static void main(String[] args) throws InterruptedException {

        int delay = Integer.parseInt(args[0]);
        Monitor m = new Monitor();
        T t1 = new T(m, 1, delay);
        T t2 = new T(m, 2, delay/2);
        T t3 = new T(m, 3, delay/3);

        t1.start(); t2.start(); t3.start();

        Thread.sleep(10); m.addOne(0, 0);
    }
}
And what happens ... without synchronisation

```
$ java MonitorExample 20
Thread 1: x is 0
Thread 1: t is 1
Thread 2: x is 0
Thread 2: t is 1
Thread 3: x is 0
Thread 3: t is 1
Thread 3: x is 1
Thread 0: x is 1
Thread 0: t is 2
Thread 0: x is 2
Thread 2: x is 1
Thread 1: x is 1

$ java MonitorExample 30
Thread 1: x is 0
Thread 1: t is 1
Thread 2: x is 0
Thread 2: t is 1
Thread 3: x is 0
Thread 3: t is 1
Thread 3: x is 0
Thread 3: t is 1
Thread 0: x is 0
Thread 0: t is 1
Thread 0: x is 1
Thread 0: t is 2
Thread 3: x is 1
Thread 2: x is 1
Thread 1: x is 1

$ $```
And what happens ... with synchronisation

$ java MonitorExample 20
Thread 1: x is 0
Thread 1: t is 1
Thread 1: x is 1
Thread 0: x is 1
Thread 0: t is 2
Thread 0: x is 2
Thread 3: x is 2
Thread 3: t is 3
Thread 3: x is 3
Thread 2: x is 3
Thread 2: t is 4
Thread 2: x is 4

Thread 1 runs to completion
Thread 0 runs to completion
Thread 3 runs to completion
Thread 2 runs to completion
no matter what delay

$
Now with Conditional synchronisation ..

class Monitor
{
    private int x = 0;

    synchronized
    void addOne(int n, int delay) throws InterruptedException
    {
        while (!(x==n)) wait();
        System.out.println("Thread "+n+": x is "+x);
        int t = x + 1;
        System.out.println("Thread "+n+": t is "+t);
        Thread.sleep(delay);
        x = t;
        notifyAll();
        System.out.println("Thread "+n+": x is "+x);
    }
}
And this is what happens ... 

```
$ java MonitorExample 20
Thread 0: x is 0
Thread 0: t is 1
Thread 0: x is 1
Thread 1: x is 1
Thread 1: t is 2
Thread 1: x is 2
Thread 2: x is 2
Thread 2: t is 3
Thread 2: x is 3
Thread 3: x is 3
Thread 3: t is 4
Thread 3: x is 4
$  
```

We will always get:

- Thread 0 runs to completion
- Thread 1 runs to completion
- Thread 2 runs to completion
- Thread 3 runs to completion

no matter what delays
notifyAll() vs. notify()

Let us demonstrate the difference between notifyAll() and notify(). Reverse the order of starting the threads, i.e. t3 then t2 then t1, and use notify() instead of notifyAll(). We get the following:

$ java MonitorExample 20
Thread 0: x is 0
Thread 0: t is 1
Thread 0: x is 1

We will always get:

Thread 0 runs to completion
Then the woken thread (3) hangs no matter what delay setting

Changing the start order back to t1, t2 and then t3, achieves the right effect with notify() because t1 will then be the woken thread.
FSP Models to Java Monitors

FSP:

\[
\text{when (guard) actionA} \rightarrow P
\]

Java:

\[
\text{synchronized type actionA (...) \{}
\]
\[
\text{while (!guard) \{ wait() ; \}}
\]
\[
\text{// Code for process P}
\]

When guard is changed, signal with notify()

Active objects initiate actions: implement as Threads
Passive (shared) objects respond to actions: implement as Monitors
Producers and Consumers: the concept

- Producers: *produce* items which are sent to consumer(s)
- Consumers: receive items and process them independently
- synchronous or buffered communication
Booth Street East Car Park — A Passive Object
FSP description

\[
\text{CARPARKCONTROL}(N=4) = \text{SPACES}[N], \\
\text{SPACES}[i:0..N] = (\text{when}(i>0) \text{ arrive}\rightarrow\text{SPACES}[i-1] \\
\text{ when}(i<N) \text{ depart}\rightarrow\text{SPACES}[i+1] \\
\text{}).
\]

\[
\text{ARRIVALS} = (\text{arrive}\rightarrow\text{ARRIVALS}). \\
\text{DEPARTURES} = (\text{depart}\rightarrow\text{DEPARTURES}).
\]

\[
||\text{CARPARK} = (\text{ARRIVALS}||\text{CARPARKCONTROL}(4)||\text{DEPARTURES}).
\]
class CarParkControl {

    protected int spaces;
    protected int capacity;

    CarParkControl(int n) {
        capacity = spaces = n;
    }

    synchronized void arrive() throws InterruptedException {
        while (spaces==0) wait();
        --spaces;
        notifyAll();
    }

    synchronized void depart() throws InterruptedException{
        while (spaces==capacity) wait();
        ++spaces;
        notifyAll();
    }

}
CountDown Timer - An Active Object

COUNTDOWN (N=3) = ( start -> COUNTDOWN[N]),

COUNTDOWN[i:0..N]= ( when (i>0) tick -> COUNTDOWN[i-1]
| when (i==0) beep -> STOP
| stop -> STOP
).
public class CountDown extends Applet implements Runnable {
    Thread counter; int i;
    AudioClip beepSound, tickSound;
    final static int N = 3;
    NumberCanvas display;
    
    public void init() {...}    // Applet methods
    public void start() {...}   // (don’t confuse with
    public void stop() {...}    //    Thread start/stop )
    public void run() {...}     // required by Runnable;
                                //    called by Thread counter
    
    private void tick(){...}   // local
    private void beep(){...}
}
public void start() { // method start for Applet
    counter = new Thread(this);
    i = N; counter.start(); // method start for Thread counter
}

public void stop() { // method stop for Applet
    counter = null; // (not using Thread stop method)
}

public void run() {
    while(true) {
        if (counter == null) return;
        if (i>0) { tick(); --i; }
        if (i==0) { beep(); return;}
    }
}

Java for CountDown Timer - II