3C03 Concurrency: Condition Synchronisation

Mark Handley

Goals

- **Introduce concepts of**
  - Condition synchronisation
  - Fairness
  - Starvation

- **Modelling:**
  - Relationship between guarded actions and condition synchronisation?

- **Implementation:**
  - Condition Monitors in Java,
  - Semaphores as Java Monitors
Thread Waiting Queues in Java

- `public final void notify()`
  
  *Wakes up a single thread that is waiting on this object’s queue*

- `public final void notifyAll()`
  
  *Wakes up all threads that are waiting on this object’s queue*

- `public final void wait()`
  
  throws InterruptedException
  
  *Waits to be notified by another thread. When notified must reacquire monitor.*

Condition synchronisation in Java

- *Thread enters monitor when it acquires mutual exclusion lock of monitor*
- *Thread exits monitor when releasing lock*
- *Wait causes thread to exit monitor*
Semaphore as a Java Monitor

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

    public synchronized down() {
        while (value_ == 0) wait();
        --value;
    }

    public synchronized up() {
        ++value;
        notify();
    }
}

Condition Synchronisation in Java

- **FSP Model:** when cond act -> NEWSTATE

- **Java:**
  public synchronized void act() throws InterruptedException {
      while (! cond) wait();
      act
      notifyAll();
  }

- **Loop re-tests cond to make sure that it is valid when it re-enters the monitor**
CarParkControl revisited

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).

CarParkControl revisited

class CarParkControl {
    private int spaces;
    private int N;

    synchronized public void arrive() {
        while (spaces<=0) {
            try {
                wait();
            } catch(InterruptedException e) {} 
        }
        --spaces;
        notify();
    }
}
FSP and Condition Synchronisation

For each guarded action in the FSP model of a monitor:
- Implement action as a synchronised method
  - That invokes `wait()` in a while loop before it begins
  - While condition is negation of guard condition
- Every change in the monitor is signalled to waiting threads using `notify()` or `notifyAll()`

Example: Producer/Consumer

![Diagram of Producer/Consumer model](image)

```
Buffer
producer
consumer
```

Demo
Producer Consumer in FSP

PRODUCER = (put -> PRODUCER).

CONSUMER = (get -> CONSUMER).

BUFFER(SIZE=5) = BUFFER[0],
BUFFER[count:0..SIZE] = (when (count<SIZE) put->BUFFER[count+1] |when (count>0) get -> BUFFER[count-1]).

||PC=(PRODUCER||BUFFER||CONSUMER).

Bounded Buffer - Outline

class Buffer {
    private Object[] buf;
    private int in = 0;  //index for put
    private int out = 0;  //index for get
    private int count = 0; //no of items
    private int size;
    Buffer(int size) {
        this.size = size;
        buf = new Object[size];
    }
    synchronized public void put(Object o) {...}
    synchronized public Object get() {...}
}
Bounded Buffer - put

synchronized public void put(Object o) {
    while (count>=size) {
        try {
            wait();
        } catch (InterruptedException e) {} 
    }
    buf[in] = o;
    ++count;
    in=(in+1) % size;
    notifyAll();
}

Bounded Buffer - get

synchronized public Object get() {
    while (count==0) {
        try {
            wait();
        } catch (InterruptedException e) {} 
    }
    Object o =buf[out];
    buf[out]=null; // for display purposes
    --count;
    out=(out+1) % size;
    notifyAll(); // [count < size]
    return (o);
}
Monitor Invariants

Monitor invariant is assertion concerning attributes encapsulated by a monitor.

- **Assertion must hold when no thread is in monitor.**

Examples:
- **CarParkControl:** $0 \leq \text{spaces} \leq N$
- **Semaphore:** $0 \leq \text{value}$
- **BoundedBuffer:**
  - $0 \leq \text{count} \leq \text{size}$
  - $0 \leq \text{in} \leq \text{size}$
  - $0 \leq \text{out} \leq \text{size}$
  - $\text{in} = (\text{out} + \text{count}) \mod \text{size}$

- **Used to reason about correctness of monitors.**

Nested Monitor Problem

SEMAPHORE (I=0) = SEMA[I],
SEMA[v:0..3] = ( up -> SEMA[v+1] 
| when (v>0) down -> SEMA[v-1]).

BUFFER = ( put -> down -> BUFFER 
| get -> up -> BUFFER ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER || CONSUMER 
|| BUFFER || SEMAPHORE(3) 
)@{put,get}. 
Nested Monitors

Normal sequence of events seems fine:
{put {down}} {get {up}} {put {down}} {get {up}} ...

But what happens when:
{put {down}} {put {down}} {put {down}} {put {down..zzz}

PRODUCER will now block, waiting for an up.
- CONSUMER must do a get to cause an up to happen.
- But the PRODUCER thread is still in the BUFFER monitor, so CONSUMER cannot run.
- DEADLOCK!

This is known as the nested monitors problem.

A Thread that waits in a monitor releases only its lock, not the lock of any monitor from which it was called.

- Nested monitors calls need to be used with great care.
- The only way to avoid nested monitors in Java is by careful design.

Modelling and analysis can aid in careful design.
Non-nested Monitors.

SEMAPHORE(1=0) = SEMA[1],
SEMA[v:0..3] = ( up->SEMA[v+1] | when (v>0) down -> SEMA[v-1]).

BUFFER = ( put -> BUFFER | get -> BUFFER ).

PRODUCER = (data.down -> put -> spaces.up -> PRODUCER).
CONSUMER = (spaces.down -> get -> data.up -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER || CONSUMER || BUFFER
|| spaces:SEMAPHORE(3)
|| data:SEMAPHORE(0))@{put, get}.

Summary

- Condition synchronization.
- In Java using `wait()`, `notify()` and `notifyAll()`
- Used to implement Semaphores in Java.
- Relationship between an FSP model and its implementation as a Java monitor.
- Monitor invariants.
- Nested Monitor Problem.