Monitors & Condition Synchronization

Concepts: monitors:
- encapsulated data + access procedures
- mutual exclusion + condition synchronization
- nested monitors

Models: guarded actions

Practice: private data and synchronized methods (exclusion).
- \texttt{wait()}, \texttt{notify()} and \texttt{notifyAll()} for condition synch.
- single thread active in the monitor at a time

5.1 Condition synchronization

A controller is required for a carpark, which only permits cars to enter when the carpark is not full and permits cars to leave when there it is not empty. Car arrival and departure are simulated by separate threads.
carpark model

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

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

\[
||\text{CARPARK} =
\text{(ARRIVALS}||\text{CARPARK}\text{CONTROL}(4)||\text{DEPARTURES}).
\]

Guarded actions are used to control arrive and depart.

LTS?

carpark program

♦ Model - all entities are processes interacting by actions
♦ Program - need to identify threads and monitors
  ♦ thread - active entity which initiates (output) actions
  ♦ monitor - passive entity which responds to (input) actions.

For the carpark?

\[
\text{ARRIVALS} \quad \text{arrive} \quad \text{CARPARK} \text{CONTROL} \quad \text{depart} \quad \text{DEPARTURES}
\]

carpark program - class diagram

Arrivals and Departures implement Runnable, CarPark\text{Control} provides the control (condition synchronization).

Instances of these are created by the \text{start()} method of the CarPark applet:

\[
\text{public void start()} \{ \\
\text{CarPark\text{Control} c = new DisplayCarPark(carDisplay, Places);} \\
\text{arrivals.start(new Arrivals(c));} \\
\text{departures.start(new Departures(c));}
\}
\]

carpark program

We have omitted DisplayThread and GraphicCanvas threads managed by ThreadPanel.
carpark program - Arrivals and Departures threads

```java
class Arrivals implements Runnable {
    CarParkControl carpark;
    Arrivals(CarParkControl c) {carpark = c;}
    public void run() {
        try {
            while(true) {
                ThreadPanel.rotate(330);
                carpark.arrive();
                ThreadPanel.rotate(30);
            }
        }
    }
    catch (InterruptedException e){}
}
```

How do we implement the control of CarParkControl?

Similarly Departures which calls carpark.depart().

condition synchronization in Java

Java provides a thread wait set per monitor (actually per object) with the following methods:

```java
public final void notify() 
    Wakes up a single thread that is waiting on this object's wait set.

class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int n) 
    {capacity = spaces = n;}
    synchronized void arrive() {
        spaces -= 1;
    }
    synchronized void depart() {
        spaces += 1;
    }
}
```

condition synchronization in Java

We refer to a thread entering a monitor when it acquires the mutual exclusion lock associated with the monitor and exiting the monitor when it releases the lock.

Wait() - causes the thread to exit the monitor, permitting other threads to enter the monitor.
**condition synchronization in Java**

FSP: when \textit{cond} \textit{act} \rightarrow \textit{NEWSTAT}

Java: \textbf{public synchronized} void \textit{act}() \textbf{throws InterruptedException} 

\begin{verbatim}
    { 
    \textbf{while} (!\textit{cond}) \textbf{wait}(); 
    \textbf{notifyAll}(); 
    \} 
\end{verbatim}

The \textbf{while} loop is necessary to retest the condition \textit{cond} to ensure that \textit{cond} is indeed satisfied when it re-enters the monitor.

\textbf{notifyAll()} is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

**models to monitors - summary**

Active entities (that initiate actions) are implemented as \textbf{threads}. Passive entities (that respond to actions) are implemented as \textbf{monitors}.

Each guarded action in the model of a monitor is implemented as a \textbf{synchronized} method which uses a while loop and \textbf{wait()} to implement the guard. The while loop condition is the negation of the model guard condition.

Changes in the state of the monitor are signaled to waiting threads using \textbf{notify()} or \textbf{notifyAll()}.
5.2 Semaphores

Semaphores are a low-level, primitive construct widely used for dealing with inter-process synchronization in operating systems. Semaphore $s$ is an integer variable that can take only non-negative values.

The only operations permitted on $s$ are $up(s)$ and $down(s)$. Blocked processes are held in a FIFO queue.

$down(s)$: if $s > 0$ then decrement $s$
else block execution of the calling process

$up(s)$: if processes blocked on $s$ then awaken one of them
else increment $s$

To ensure analyzability, we only model semaphores that take a finite range of values. If this range is exceeded then we regard this as an ERROR. $N$ is the initial value.

const Max = 3
range Int = 0..Max


LTS?

Action $down$ is only accepted when value $v$ of the semaphore is greater than 0.
Action $up$ is not guarded.
Trace to a violation:
$up \rightarrow up \rightarrow up \rightarrow up$

semaphore demo - model

Three processes $p[1..3]$ use a shared semaphore mutex to ensure mutually exclusive access (action $critical$) to some resource.

$LOOP = (mutex.down->critical->mutex.up->LOOP)$. $||SEMADemo = (p[1..3]:LOOP$. $||p[1..3]:mutex:SEMAPHORE(1))$.

For mutual exclusion, the semaphore initial value is 1. Why?
Is the ERROR state reachable for SEMADemo?
Is a binary semaphore sufficient (i.e. Max=1)?

LTS?
semaphores in Java

Semaphores are passive objects, therefore implemented as monitors.

(Note: In practice, semaphores are a low-level mechanism often used for implementing the higher-level monitor construct.

Java SE5 provides general counting semaphores)

```java
public class Semaphore {
    private int value;

    public Semaphore (int initial)
    {value = initial;}

    synchronized public void up() {
        ++value;
        notifyAll();
    }

    synchronized public void down()
    throws InterruptedException {
        while (value== 0) wait();
        --value;
    }
}
```

**SEMADEMO**

What if we adjust the time that each thread spends in its critical section?

- Large resource requirement - more conflict?
  (eg. more than 67% of a rotation)?
- Small resource requirement - no conflict?
  (eg. less than 33% of a rotation)?

Hence the time a thread spends in its critical section should be kept as short as possible.
5.3 Bounded Buffer

A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a producer process and removed by a consumer process. It can be used to smooth out transfer rates between the producer and consumer.

(see car park example)

bounded buffer - a data-independent model

The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner.

LTS:
bounded buffer - a data-independent model

\[
\text{BUFFER}(N=5) = \text{COUNT}[0], \\
\text{COUNT}[i:0..N] = \begin{cases} 
\text{put}\rightarrow\text{COUNT}[i+1] & \text{when } (i<N) \\
\text{get}\rightarrow\text{COUNT}[i-1] & \text{when } (i>0)
\end{cases}
\]

\[
\text{PRODUCER} = (\text{put}\rightarrow\text{PRODUCER}). \\
\text{CONSUMER} = (\text{get}\rightarrow\text{CONSUMER}). \\
\text{BOUNDEDBUFFER} = (\text{PRODUCER}|\text{BUFFER}(5)|\text{CONSUMER}).
\]

bounded buffer program - buffer monitor

```java
public interface Buffer <E> {...

class BufferImpl <E> implements Buffer <E> { ...

    public synchronized void put(E o) 
        throws InterruptedException {
            //other implementation...
        }

    public synchronized E get() 
        throws InterruptedException {
            //other implementation...
        }
}
```

We separate the interface to permit an alternative implementation later.

bounded buffer program - producer process

```java
class Producer implements Runnable {
    Buffer buf;
    String alphabet= "abcdefghijklmnopqrstuvwxyz";
    Producer(Buffer b) {buf = b;}

    public void run() {
        try {
            int ai = 0;
            while(true) {
                ThreadPanel.rotate(12);
                buf.put(alphabet.charAt(ai));
                ai=(ai+1) % alphabet.length();
                ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e){}
    }
}
```

Similarly Consumer which calls buf.get().

5.4 Nested Monitors

Suppose that, in place of using the count variable and condition synchronization directly, we instead use two semaphores full and empty to reflect the state of the buffer.

```java
class SemaBuffer <E> implements Buffer <E> {
    //other implementation...

class sBuffer { ...
    Semaphore full; //counts number of items
    Semaphore empty; //counts number of spaces

    SemaBuffer(int size) {
        this.size = size; 
        buf = (E[])new Object[size];
        full = new Semaphore(0);
        empty= new Semaphore(size);
    }
}
```

...
nested monitors - bounded buffer program

```java
synchronized public void put(E o) throws InterruptedException {
   empty.down();
   buf[in] = o;
   ++count; in=(in+1)%size;
   full.up();
}

synchronized public E get() throws InterruptedException{
   full.down();
   E o =buf[out]; buf[out]=null;
   --count; out=(out+1)%size;
   empty.up ();
   return (o);
}
```

`empty` is decremented during a `put` operation, which is blocked if `empty` is zero; `full` is decremented by a `get` operation, which is blocked if `full` is zero.

LTSA analysis predicts a possible DEADLOCK:

- Composing potential DEADLOCK
- States Composed: 28 Transitions: 32 in 60ms
- Trace to DEADLOCK:
  - get

The Consumer tries to `get` a character, but the buffer is empty. It blocks and releases the lock on the semaphore `full`. The Producer tries to `put` a character into the buffer, but also blocks. Why?

This situation is known as the nested monitor problem.

Does this behave as desired?

2015 Concurrency: monitors & condition synchronization

nested monitors - bounded buffer model

```java
const Max = 5
range Int = 0..Max

SEMAPHORE ...as before...

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

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).
||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
   ||empty:SEMAPHORE(5)
   ||full:SEMAPHORE(0)
 )@{put,get}.
```

Does this behave as desired?

2015 Concurrency: monitors & condition synchronization
2015 Concurrency: monitors & condition synchronization

### nested monitors - revised bounded buffer program

The only way to avoid it in Java is by careful design. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until *after* semaphores are decremented.

```java
public void put(E o) throws InterruptedException {
    empty.down();
    synchronized (this) {
        buf[in] = o; ++count; in = (in + 1) % size;
    }
    full.up();
}
```

### nested monitors - revised bounded buffer model

The semaphore actions have been moved to the producer and consumer. This is exactly as in the implementation where the semaphore actions are *outside* the monitor.

Does this behave as desired?

Minimized LTS?

### 5.5 Monitor invariants

An *invariant* for a monitor is an assertion concerning the variables it encapsulates. This assertion must hold whenever there is no thread executing inside the monitor i.e. on thread *entry* to and *exit* from a monitor.

<table>
<thead>
<tr>
<th>CarParkControl Invariant:</th>
<th>0 ≤ spaces ≤ N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore Invariant:</td>
<td>0 ≤ value</td>
</tr>
<tr>
<td>Buffer Invariant:</td>
<td>0 ≤ count ≤ size</td>
</tr>
<tr>
<td></td>
<td>and 0 ≤ in &lt; size</td>
</tr>
<tr>
<td></td>
<td>and 0 ≤ out &lt; size</td>
</tr>
<tr>
<td></td>
<td>and in = (out + count) modulo size</td>
</tr>
</tbody>
</table>

Invariants can be helpful in reasoning about correctness of monitors using a *proof-based* approach. Generally we prefer to use a *model-based* approach amenable to mechanical checking.

### 5.6 Java Concurrency Utilities Package

`java.util.concurrent` includes *semaphores*, and *explicit locks with multiple condition variables*.

<table>
<thead>
<tr>
<th>Monitors:</th>
<th>implicit lock associated with each object, with methods <code>wait()</code>, <code>notify()</code> and <code>notifyAll()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock interface:</td>
<td>explicit lock objects, with methods <code>lock()</code>, <code>unlock()</code>, <code>tryLock()</code>, and <code>newCondition()</code></td>
</tr>
<tr>
<td>Condition objects:</td>
<td>explicit lock synchronization objects, with methods <code>await()</code>, <code>signal()</code> and <code>signalAll()</code></td>
</tr>
</tbody>
</table>

Conditions gives the effect of having multiple wait-sets per object.
bounded buffer – explicit lock with separate conditions

class BufferImpl <E> implements Buffer <E> {
    final Lock buflock = new ReentrantLock();
    final Condition notFull = buflock.newCondition();
    final Condition notEmpty = buflock.newCondition();
    
    public void put(E o) throws InterruptedException {
        buflock.lock();
        try {
            while (count==size) notFull.await();
            buf[in] = o; ++count; in=(in+1)%size;
            notEmpty.signalAll();
        } finally {
            buflock.unlock();
        }
    }
    
    public E get() throws InterruptedException {
        buflock.lock();
        try {
            while (count==0) notEmpty.await();
            E o =buf[out];
            buf[out]=null; --count; out=(out+1)%size;
            notFull.signalAll();
            return (o);
        } finally {
            buflock.unlock();
        }
    }
}

notFull and notEmpty are conditions associated with buflock. Processes wait separately: producers on notFull and consumers on notEmpty.

Summary

◆ Concepts
  ● monitors: encapsulated data + access procedures
    mutual exclusion + condition synchronization
  ● nested monitors

◆ Model
  ● guarded actions

◆ Practice
  ● monitors: private data and synchronized methods in Java
  ● wait(), notify() and notifyAll() for condition synchronization
  ● single thread active in the monitor at a time