Chapter 5

Monitors &
Condition Synchronization
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).
- wait(), notify() and notifyAll() for condition synch.
- single thread active in the monitor at a time
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

♦ Events or actions of interest?

arrive and depart

♦ Identify processes.

arrivals, departures and carpark control

♦ Define each process and interactions (structure).
carpark model

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

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

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

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?

```
ARRIVALS       CARPARK CONTROL      DEPARTURES
    arrive                       depart
```

©Magee/Kramer 2nd Edition
We have omitted DisplayThread and GraphicCanvas threads managed by ThreadPanel.
carpark program

Arrivals and Departures implement Runnable, CarParkControl provides the control (condition synchronization).

Instances of these are created by the start() method of the CarPark applet:

```java
public void start() {
    CarParkControl c =
        new DisplayCarPark(carDisplay, Places);
    arrivals.start(new Arrivals(c));
    departures.start(new Departures(c));
}
```
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().
class CarParkControl {
  protected int spaces;
  protected int capacity;

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

  synchronized void arrive() {
    --spaces; ...
  }

  synchronized void depart() {
    ++spaces; ...
  }
}

mutual exclusion by synch methods
condition synchronization?
block if full? (spaces==0)
block if empty? (spaces==N)
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.

public final void notifyAll()
    Wakes up all threads that are waiting on this object's wait set.

public final void wait()
    throws InterruptedException
    Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution.
```
**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.
Monitor lock

Thread C
Thread F
Thread E

Monitor

Thread B

monitor lock

Thread A

Thread A

Thread B

Thread F

Thread E

Monitor

wait()
condition synchronization in Java

FSP:  when cond act -> NEWSTAT

Java:  public synchronized void act() throws InterruptedException {
    while (!cond) wait();
    // modify monitor data
    notifyAll()
}

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

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

Is it safe to use notify() rather than notifyAll()?
models to monitors - summary

**Active** entities (that initiate actions) are implemented as **threads**. **Passive** entities (that respond to actions) are implemented as **monitors**.

Each guarded action in the model of a monitor is implemented as a **synchronized** method which uses a while loop and **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 **notify()** or **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.

\[
\begin{align*}
\text{down}(s): & \quad \text{if } s > 0 \text{ then} \\
& \quad \quad \text{decrement } s \\
& \quad \text{else} \\
& \quad \quad \text{block execution of the calling process} \\
\text{up}(s): & \quad \text{if processes blocked on } s \text{ then} \\
& \quad \quad \text{awaken one of them} \\
& \quad \text{else} \\
& \quad \quad \text{increment } s
\end{align*}
\]
modeling semaphores

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.

\[
\begin{align*}
\text{const Max} &= 3 \\
\text{range Int} &= 0..\text{Max} \\
\text{SEMAPHORE}(N=0) &= \text{SEMA}[N], \\
\text{SEMA}[v:\text{Int}] &= (\text{up}->\text{SEMA}[v+1] \\
&\quad | \text{when}(v>0) \text{ down}->\text{SEMA}[v-1] \\
&\quad ), \\
\text{SEMA}[\text{Max+1}] &= \text{ERROR}.
\end{align*}
\]
modeling semaphores

Action `down` is only accepted when value \( v \) of the semaphore is greater than 0.

Action `up` is not guarded.

Trace to a violation:
\[
\text{up } \rightarrow \text{ up } \rightarrow \text{ up } \rightarrow \text{ up }
\]
semaphore demo - model

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

\[
\text{LOOP} = (\text{mutex.down} \Rightarrow \text{critical} \Rightarrow \text{mutex.up} \Rightarrow \text{LOOP}).
\]

\[
\| \| \text{SEMADEMO} = (p[1..3]:\text{LOOP} \\
\| \| \{p[1..3]\}::\text{mutex}:\text{SEMAPHORE}(1)).
\]

For mutual exclusion, the semaphore initial value is 1. Why?

Is the ERROR state reachable for $\text{SEMADEMO}$?

Is a binary semaphore sufficient (i.e. $\text{Max}=1$) ?

$LTS$?
semaphore demo - model
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)
current semaphore value

thread 1 is executing critical actions.

thread 2 is blocked waiting.

thread 3 is executing non-critical actions.
What if we adjust the time that each thread spends in its critical section?

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

Hence the time a thread spends in its critical section should be kept as short as possible.
public class ThreadPanel extends Panel {

    // construct display with title and rotating arc color c
    public ThreadPanel(String title, Color c) {...}

    // hasSlider == true creates panel with slider
    public ThreadPanel
        (String title, Color c, boolean hasSlider) {...}

    // rotate display of currently running thread 6 degrees
    // return false when in initial color, return true when in second color
    public static boolean rotate()
        throws InterruptedException {...}

    // rotate display of currently running thread by degrees
    public static void rotate(int degrees)
        throws InterruptedException {...}

    // create a new thread with target r and start it running
    public void start(Runnable r) {...}

    // stop the thread using Thread.interrupt()
    public void stop() {...}
}
class MutexLoop implements Runnable {
    Semaphore mutex;
    MutexLoop (Semaphore sema) {mutex=sema;}
    public void run() {
        try {
            while(true) {
                while(!ThreadPanel.rotate());
                mutex.down(); // get mutual exclusion
                while(ThreadPanel.rotate()); //critical actions
                mutex.up(); //release mutual exclusion
            }
        } catch(InterruptedException e){}
    }
}

ThreadPanel.rotate() returns false while executing non-critical actions (dark color) and true otherwise.

Threads and semaphore are created by the applet start() method.
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

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

\[
\text{PRODUCER} = (\text{put->PRODUCER}).
\]
\[
\text{CONSUMER} = (\text{get->CONSUMER}).
\]

\[
|\text{BOUNDEDBUFFER} = (\text{PRODUCER}|\text{BUFFER}(5)|\text{CONSUMER}) .
\]
bounded buffer program - buffer monitor

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

class BufferImpl <E> implements Buffer <E> {
    ...
    public synchronized void put(E o)
        throws InterruptedException {
            while (count==size) wait();
            buf[in] = o; ++count; in=(in+1)%size;
            notifyAll();
        }

    public synchronized E get()
        throws InterruptedException {
            while (count==0) wait();
            E o =buf[out];
            buf[out]=null; --count; out=(out+1)%size;
            notifyAll();
            return (o);
        }
}
```

We separate the interface to permit an alternative implementation later.
bounded buffer program - producer process

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) {} 
    }
}
5.4 Nested Monitors

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

```java
class SemaBuffer <E> implements Buffer <E> {
    ...

    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.

Does this behave as desired?
nested monitors - bounded buffer model

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?
nested monitors - bounded buffer model

*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**.
nested monitors - bounded buffer model

```java
synchronized public Object get() throws InterruptedException{
    full.down(); // if no items, block!
    ...
}
```

```
get
---
buffer
---
put

full
---
wait
---
empty
down
```
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();
}
```
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.

- CarParkControl Invariant: \[ 0 \leq \text{spaces} \leq N \]
- Semaphore Invariant: \[ 0 \leq \text{value} \]
- Buffer Invariant: \[ 0 \leq \text{count} \leq \text{size} \]
  \[ \text{and } 0 \leq \text{in} < \text{size} \]
  \[ \text{and } 0 \leq \text{out} < \text{size} \]
  \[ \text{and } \text{in} = (\text{out} + \text{count}) \mod \text{size} \]

Invariants can be helpful in reasoning about correctness of monitors using a logical 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: implicit lock associated with each object, with methods <code>wait()</code>, <code>notify()</code> and <code>notifyAll()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock interface:</strong> explicit lock objects, with methods <code>lock()</code>, <code>unlock()</code>, <code>tryLock()</code>, and <code>newCondition()</code></td>
</tr>
<tr>
<td><strong>Condition objects:</strong> 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();}
    }
}
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