Chapter 5

Monitors &
Condition Synchronization
monitors & condition synchronization

**Concepts:** monitors:
- encapsulated data + access procedures
- mutual exclusion + condition synchronization
- single access procedure active in the monitor
- 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
5.1 Condition synchronization

A controller is required for a carpark, which only permits cars to enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark. 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).

![Diagram of carpark model]

Concurrency: monitors & condition synchronization
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] \\
\quad | \text{when}(i<N) \ \text{depart} \rightarrow \text{SPACES}[i+1] \\
\).
\]

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

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

**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 arrive CARPARK
- CARPARK CONTROL
- depart DEPARTURES
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().

Concurrence: monitors & condition synchronization
class CarParkControl {
    protected int spaces;
    protected int capacity;

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

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

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

Concurrency: monitors & condition synchronization
condition synchronization in Java

Java provides a thread **wait queue** 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 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. 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.
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.
CarParkControl - condition synchronization

class CarParkControl {
  protected int spaces;
  protected int capacity;

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

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

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

Why is it safe to use notify() here 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 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 $\text{up}(s)$ and $\text{down}(s)$. Blocked processes are held in a FIFO queue.

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

$\text{up}(s)$: if processes blocked on $s$ then
awaken one of them
else
increment $s$
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.

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

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

For mutual exclusion, the semaphore initial value is 1. \textbf{Why?}

Is the \textbf{ERROR} state reachable for \textbf{SEMADEMO}?

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

\textbf{LTS?}
Concurrency: monitors & condition synchronization

semaphore demo - model

0 ➔ p.1.mutex.down ➔ 1 ➔ p.2.mutex.down ➔ 2 ➔ p.3.mutex.down ➔ 3 ➔ p.3.mutex.up ➔ 4 ➔ p.2.mutex.up ➔ 5 ➔ p.1.mutex.up ➔ 6 ➔ p.1.mutex.up ➔ 0

©Magee/Kramer
semaphores in Java

Semaphores are passive objects, therefore implemented as monitors.

(In practice, semaphores are a low-level mechanism often used in implementing the higher-level monitor construct.)

```
public class Semaphore {
    private int value;

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

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

    synchronized public void down() throws InterruptedException {
        while (value== 0) wait();
        --value;
    }
}
```
**Concurrency: monitors & condition synchronization**

The image shows a demonstration of semaphore operations in a multi-threaded environment. The display includes:

- **Mutex**: The current semaphore value is 0.
- **Thread 1**: Executing critical actions.
- **Thread 2**: Blocked waiting.
- **Thread 3**: Executing non-critical actions.

The demonstration illustrates how threads are managed and synchronized using semaphores.
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.
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() {...}
}

©Magee/Kramer
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:

Concurrency: monitors & condition synchronization
bounded buffer - a data-independent model

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

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

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

```java
public interface Buffer {
...
}

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

    public synchronized Object get() throws InterruptedException {
        while (count==0) wait();
        Object o =buf[out];
        buf[out]=null; --count; out=(out+1)%size;
        notify();
        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(new Character(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 *count* variable and condition synchronization directly, we instead use two semaphores *full* and *empty* to reflect the state of the buffer.

```java
class SemaBuffer implements Buffer {
    ...
    Semaphore full;    //counts number of items
    Semaphore empty;   //counts number of spaces

    SemaBuffer(int size) {
        this.size = size; buf = new Object[size];
        full = new Semaphore(0);
        empty = new Semaphore(size);
    }
    ...
}
```
nested monitors - bounded buffer program

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

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

Does this behave as desired?

*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.

Concurrency: monitors & condition synchronization
nested monitors - bounded buffer model

```plaintext
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**:

<table>
<thead>
<tr>
<th>Composing potential DEADLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>States Composed: 28 Transitions: 32 in 60ms</td>
</tr>
<tr>
<td>Trace to DEADLOCK:</td>
</tr>
<tr>
<td>get</td>
</tr>
</tbody>
</table>

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**.
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(Object 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: \( 0 \leq \text{count} \leq \text{size} \)
\[ \text{and } 0 \leq \text{in} < \text{size} \]
\[ \text{and } 0 \leq \text{out} < \text{size} \]
\[ \text{and } \text{in} = (\text{out} + \text{count}) \text{ modulo size} \]

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.
Summary

Concepts

monitors: encapsulated data + access procedures

mutual exclusion + condition synchronization

nested monitors

Model

guarded actions

Practice

private data and synchronized methods in Java

wait(), notify() and notifyAll() for condition synchronization

single thread active in the monitor at a time