Monitors and Semaphores

Jens Chr. Godskesen

IT University of Copenhagen
Overview

Monitors and semaphores may help to obtain mutual exclusion.

- Monitors
- Waiting queues
- Semaphores
- Bounded Buffers (Producer/Consumer)
- Nested Monitors
- Exercises
class Counter {

    private int value=0;
    private boolean updated = false;

    public synchronized void update() {
        updated = true;
        value++;
    }

    public synchronized int getValue() {
        return value;
    }
}

Monitors

Monitors and Semaphores

Jens Chr. Godskesen, IT University of Copenhagen
Monitor Definition

A *monitor* is a piece of code (a class) characterized by the facts that it

- allows for encapsulation of data
- provides access procedures (or methods) to the data
- the methods (all of them) are executed under mutual exclusion
- supports *condition synchronization*

Condition synchronization allows the methods of a monitor to block until a particular condition holds.

```java
public synchronized int getValue() {
    while (!updated) wait();
    return value;
}
```

but we should be careful to wake up blocked processes
A car park with three parking slots can be modeled as a parallel composition of three processes Arr, Dep, and Con by

agent Arr(arr) = 'arr.Arr<arr>
agent Dep(dep) = 'dep.Dep<dep>

(* Con(a,d) controls a park lot with initially 3 vacant slots *)

agent Con(a,d) = Con3(a,d)
agent Con0(a,d) = d.Con1<a,d>
agent Con1(a,d) = d.Con2<a,d> + a.Con0<a,d>
agent Con2(a,d) = d.Con3<a,d> + a.Con1<a,d>
agent Con3(a,d) = a.Con2<a,d>

agent CarPark(arr,dep) = (^ arr, dep)(Arr<arr> | Con<arr,dep> | Dep<dep>)
Compact representation of controller

The specification of the controller could have been more compactly defined by
(letting \( i \) denote the number of vacant slots)

\[
\begin{align*}
\text{agent } & \text{Con}(a, d) = \text{Con}(a, d, 3) \\
\text{agent } & \text{Con}(a, d, i) = \text{if } (i > 0) \ a.\text{Con}<a, d, i-1> \\
& \quad + \text{if } (i < 3) \ d.\text{Con}<a, d, i+1>
\end{align*}
\]

but not allowed in MWB.
Car park implementation

public class CarPark {

    final static int Places = 3;

    public static void main(String[] args) {
        CarParkControl carparkcontrol = new CarParkControl(Places);
        Arrivals arrivals = new Arrivals(carparkcontrol);
        Departures departures = new Departures(carparkcontrol);
        arrivals.start();
        departures.start();
    }
}

Jens Chr. Godskesen, IT University of Copenhagen
Implementation of arrivals thread

class Arrivals extends Thread {

    CarParkControl carpark;

    Arrivals(CarParkControl c) {carpark = c;}

    public void run() {
        try {
            try {
                while(true) {
                    if (Math.random()<0.5) Thread.sleep(500);
                    carpark.arrive();
                }
            } catch (InterruptedException e){}
        }
    }
}
Implementation of departures thread

class Departures extends Thread {
    CarParkControl carpark;
    Departures(CarParkControl c) {carpark = c;}
    public void run() {
        try {
            while(true) {
                if (Math.random()<0.5) Thread.sleep(500);
                carpark.depart();
            }
        } catch (InterruptedException e){}
    }
}
Implementation of Control

class CarParkControl {

    private int spaces; // vacant spaces
    private int capacity;

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

    synchronized void arrive() throws InterruptedException {
        while (spaces==0) wait(); //block if full
        --spaces;
        System.out.println("car arrived; now vacant "+ spaces);
        notifyAll();
    }

    synchronized void depart() throws InterruptedException {
        while (spaces==capacity) wait(); //block if empty
        ++spaces;
        System.out.println("car departed; now vacant "+ spaces);
        notifyAll();
    }
}
Exercise

Try to run the program CarPark.java
if (b) act.P

is implemented as

synchronized void act() throws InterruptedException {
    while (!b) wait();
    // ... whatever is going on
    notifyAll();
}

I.e. each guarded action is implemented as a synchronized method with the combination of a while loop and a call to `wait()` to implement the guard.

The condition in the while loop is the negation of the condition of the guard.
Guiding the translation from model to Java

For instance,

```java
if (i > 0) a.Con<a,d,i-1>
```

is implemented by

```java
synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait(); // block if full
    --spaces;
    notifyAll();
}
```
Waiting queues

Any Java object $O$ has a waiting queue.

- A thread $T$ calling $O$'s $\text{wait()}$ method is put into $O$'s waiting queue, $T$ releases the synchronization lock of $O$, and waits to be notified by another thread. When $T$ is notified it must reacquire the lock once again before it is allowed to resume execution.

- A thread calling $O$'s $\text{notify()}$ method wakes up a single randomly chosen thread that is waiting in the queue of $O$, the thread being notified becomes runnable.

- A thread calling $O$'s $\text{notifyAll()}$ method wakes up all the threads waiting in the queue of $O$, all the threads becomes runnable.
**Exercise**  Why cannot *if* be used instead of *while* in the synchronization condition of a monitor?

It’s “cheaper” to call `notify()` instead of `notifyAll()`, since only one of the threads in a waiting queue is released when calling `notify()`. However, sometimes it may be erroneous to call `notify()` instead of `notifyAll()`.

**Exercise**  In the CarParkControl example why is it sufficient to call `notify()` instead of `notifyAll()`?
Example

Reconsider the CarPark-example.

- Suppose we have a car park with (only) two parking slots.
- Suppose we have one Arrivals thread $A$ and four Departures threads $D_1, D_2, D_3, D_4$.
- Suppose moreover that all occurrences of `notifyAll()` are replaced by `notify()` in `CarPark.java`.
- Assume the car park is initially empty and that $D_4$ is running.

An example of execution that deadlocks may be as follows:
### Example

<table>
<thead>
<tr>
<th>Runnable</th>
<th>Running</th>
<th>Vacant Spaces</th>
<th>Queued</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, D₁, D₂, D₃</td>
<td>D₄</td>
<td>2</td>
<td>D₄</td>
</tr>
<tr>
<td>A, D₁, D₂, D₃</td>
<td>D₃</td>
<td>2</td>
<td>D₄</td>
</tr>
<tr>
<td>A, D₁, D₂</td>
<td>D₃</td>
<td>2</td>
<td>D₃, D₄</td>
</tr>
<tr>
<td>A, D₁</td>
<td>D₂</td>
<td>2</td>
<td>D₃, D₄</td>
</tr>
<tr>
<td>A, D₁</td>
<td>D₂</td>
<td>2</td>
<td>D₂, D₃, D₄</td>
</tr>
<tr>
<td>A</td>
<td>D₁</td>
<td>2</td>
<td>D₂, D₃, D₄</td>
</tr>
<tr>
<td>A</td>
<td>D₁</td>
<td>2</td>
<td>D₁, D₂, D₃, D₄</td>
</tr>
<tr>
<td>A</td>
<td>D₁</td>
<td>2</td>
<td>D₂, D₃, D₄</td>
</tr>
<tr>
<td>A</td>
<td>D₁</td>
<td>2</td>
<td>D₁, D₂, D₃, D₄</td>
</tr>
<tr>
<td>A</td>
<td>D₁</td>
<td>2</td>
<td>D₂, D₃, D₄</td>
</tr>
<tr>
<td>D₁</td>
<td>A</td>
<td>1</td>
<td>D₂, D₃, D₄</td>
</tr>
<tr>
<td>D₁, D₂</td>
<td>A</td>
<td>0</td>
<td>D₃, D₄</td>
</tr>
<tr>
<td>D₁, D₂</td>
<td>A</td>
<td>0</td>
<td>A, D₃, D₄</td>
</tr>
<tr>
<td>D₁</td>
<td>D₁</td>
<td>0</td>
<td>A, D₃, D₄</td>
</tr>
<tr>
<td>D₂, D₃</td>
<td>D₁</td>
<td>1</td>
<td>A, D₄</td>
</tr>
<tr>
<td>D₂, D₃, D₄</td>
<td>D₁</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D₂, D₃, D₄</td>
<td>D₁</td>
<td>2</td>
<td>A, D₁</td>
</tr>
<tr>
<td>D₃, D₄</td>
<td>D₂</td>
<td>2</td>
<td>A, D₁</td>
</tr>
<tr>
<td>D₃, D₄</td>
<td>D₃</td>
<td>2</td>
<td>A, D₁, D₂</td>
</tr>
<tr>
<td>D₄</td>
<td>D₃</td>
<td>2</td>
<td>A, D₁, D₂</td>
</tr>
<tr>
<td>D₄</td>
<td>D₄</td>
<td>2</td>
<td>A, D₁, D₂</td>
</tr>
<tr>
<td>D₄</td>
<td>D₄</td>
<td>2</td>
<td>A, D₁, D₂</td>
</tr>
<tr>
<td>D₄</td>
<td>D₄</td>
<td>2</td>
<td>A, D₁, D₂</td>
</tr>
<tr>
<td>D₄</td>
<td>D₄</td>
<td>2</td>
<td>A, D₁, D₂</td>
</tr>
<tr>
<td>D₄</td>
<td>D₄</td>
<td>2</td>
<td>A, D₁, D₂</td>
</tr>
</tbody>
</table>
Semaphores

Another mechanism to deal with synchronization problems is a semaphore (by Dijkstra in 1968, so historically it predates monitors).

Semaphores are far less structured compared to the more recent monitors.

A semaphore $s$ is an integer variable that

- can take only non-negative values

and has exactly two atomic operations defined on it:

- $\text{down}(s)$: if $s > 0$ then $s--$ else suspend the execution of the process
- $\text{up}(s)$: if there are processes suspended on $s$ then wake one of them else $s++$

A binary semaphore, in contrast to a general semaphore, can only take the values 0 and 1.
Example

Semaphore s = 1;

Process P1 is
while (true) {
    non_critical_1
    down(s);
    critical_1;
    up(s);
}

Process P2 is
while (true) {
    non_critical_2
    down(s);
    critical_2;
    up(s);
}

**Exercise**  Draw what happens when P1 and P2 are running concurrently.

Using semaphores is a low level style of programming. Require strict use of up's and down's.

**Exercise**  What happens if several up's were issued after the critical section?
Monitors and Semaphores

Modeling Semaphores

The system above consisting of the two processes that are supposed to work under mutual exclusion can be modeled using a binary semaphore with initial value one.

agent Sema(u,d) = Semaphore1(u,d)
agent Semaphore0(u,d) = u.Semaphore1<u,d>
agent Semaphore1(u,d) = d.Semaphore0<u,d>

agent Proc(critical,down,up) = 'down.critical.'up.Proc<critical,down,up>

agent Sys(c1,c2,c3) = (^ d,u) ( Proc<c1,d,u> | Proc<c2,d,u> | Sema<u,d> )

**Exercise**  Why is a binary semaphore sufficient in the model above?

**Exercise**  What happens if the initial value of the semaphore is distinct from 1 in the CCS model with the two processes above?
public class Semaphore {

    private int s = 0;

    public Semaphore(int i) {
        if (i >= 0) s = i;
    }

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

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

Semaphores in Java
Semaphores may be used for other things than mutual exclusion! Here is an allegory:

Students café  Suppose that at most \( n \) students are allowed to be in the students café at the same time. The café is a critical region! Let a semaphore \( s \) be a tray with glasses. Initially the tray contains \( n \) glasses. A student must be in possession of one of the glasses before entering the café. Each time a student enters the café she must acquire a glass (\texttt{down}(s)). Upon leaving the café she gives the glass to a student waiting for entering the café or if no one is waiting she puts the glass on the tray (\texttt{up}(s)).
Monitors revisited

The following story is an analogy for monitors.

**The Sheep Fold**  The sheeps want to enter the fold to eat grass. Some sheeps like long grass and some like short grass. Only one sheep can be in the fold at any one time (mutual exclusion). When a sheep enters the fold it detects whether the grass is long or short. If it doesn’t like the grass it enters a stable (a queue) where it goes to sleep. If it likes the grass it starts eating and leaves the fold when it is finished. But, just before exiting the fold it wakes (one or all) sheeps in the stable (notify or notify all). The awakened sheeps are now ready to compete for entering the fold again.
Producer-Consumer

Consider the following example of a producer and a consumer using a bounded buffer.

```java
public class ProducerConsumer {

    final static int Places = 4;

    public static void main(String[] args) {
        BoundedBuffer buffer = new BoundedBuffer(Places);
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);
        producer.start();
        consumer.start();
    }
}
```
class Producer extends Thread {

    Buffer buf;
    String alphabet = "abcdefghijklmnopqrstuvwxyz";

    Producer(Buffer b) { buf = b; }

    public void run() {
        try {
            int i = 0;
            while(true) {
                buf.put(new Character(alphabet.charAt(i)));
                i = (i+1) % alphabet.length();
            }
        } catch (InterruptedException e) {} 
    }
}
class Consumer extends Thread {

    Buffer buf;
    Character c;

    Consumer(Buffer b) {buf = b;}

    public void run() {
        try {
            while(true) c = (Character)buf.get();
        } catch(InterruptedException e ){}
    }
}
public interface Buffer {
    public void put(Object o) throws InterruptedException; // put object into buffer
    public Object get() throws InterruptedException; // get an object from buffer
}
public class BoundedBuffer implements Buffer {

    protected Object[] buf;
    protected int in = 0;  //next index to insert
    protected int out = 0; //next index to take from
    protected int count = 0; //amount of objects in buffer
    protected int size;   //size of the buffer

    public BoundedBuffer(int size) {
        if (size > 0) this.size = size;
        buf = new Object[size];
    }

    public synchronized void put(Object o) throws InterruptedException {
        while (count == size) wait();
        buf[in] = o;
        in = (in + 1) % size;
        System.out.println("put: "+o);
        if (count++ == 0) notify();  //consumer may be waiting
    }

    public synchronized Object get() throws InterruptedException {
        while (count == 0) wait();
        Object o = buf[out];
        buf[out] = null;
        out = (out + 1) % size;
        System.out.println("get:"+o);
        if (count-- == size) notify();  //producer may be waiting
        return (o);
    }
}
Producer-Consumer

Exercise  Make a drawing of the circular indexing principle using in and out. Be aware that the “difference” between in and out is count, i.e.

\[ \text{in} == (\text{out} + \text{count}) \mod \text{size} \]

Notice the notification principle in the code for the bounded buffer. Only one producer (or consumer) needs to be awakened (hence notify and not notifyAll) and notifications are not performed always, only in case someone may have been waiting.

Exercise  Try to run the program ProducerConsumer.java.
Suppose we use semaphores instead of condition synchronization in the implementation of the buffer.

The semaphores replaces the variable count and the use of condition synchronization.

Two semaphores, items and spaces, shared between the producer and the consumer are needed.

The role of the semaphore items is to count the number of items, hence it cannot be decremented when there is no items in the buffer.

The role of spaces is to count the number of vacant spaces in the buffer, hence it cannot be decremented when there is no spaces left in the buffer.
protected Object[] buf;
protected int in = 0, int out = 0, int size;

Semaphore items;  // counts number of items
Semaphore spaces;  // counts number of spaces

public SemaBoundedBuffer(int size) {
    this.size = size;
    buf = new Object[size];
    items = new Semaphore(0);
    spaces = new Semaphore(size);
}

public synchronized void put(Object o) throws InterruptedException {
    spaces.down();
    buf[in] = o;
    in=(in+1) % size;
    System.out.println("put");
    items.up();
}

public synchronized Object get() throws InterruptedException {
    items.down();
    Object o = buf[out];
    buf[out]=null;
    out=(out+1) % size;
    System.out.println("get");
    spaces.up();
    return (o);
}
Monitors and Semaphores

Model of the bounded buffer

agent Semaphore0(u,d) = u.Semaphore1<u,d>
agent Semaphore1(u,d) = u.Semaphore2<u,d> + d.Semaphore0<u,d>
agent Semaphore2(u,d) = u.Semaphore3<u,d> + d.Semaphore1<u,d>
agent Semaphore3(u,d) = d.Semaphore2<u,d>

agent Spaces(u,d) = Semaphore3(u,d)
agent Items(u,d) = Semaphore0(u,d)

agent Buf(p,g,s_d,s_u,i_d,i_u) =
   p.'s_d.'i_u.Buf<p,g,s_d,s_u,i_d,i_u>
   + g.'i_d.'s_u.Buf<p,g,s_d,s_u,i_d,i_u>

agent BB(put,get) =
   (^ s_d,s_u,i_d,i_u)( Buf<put,get,s_d,s_u,i_d,i_u>
   | Spaces<s_u,s_d> | Items<i_u,i_d> )
Monitors and Semaphores

Detecting a deadlock

Notice, that Buf models that in the Java code get and put are synchronized!

**Exercise** If we start with get, then BB blocks, i.e. a deadlock may occur! Explain why there is a deadlock? Try to run the model ErrBoundBuf in MWB issuing the command `step BB<put,get>`. In the Java code there is therefore also a deadlock. When the call to get locks the monitor object the body of get is blocked on items. This blocking can only be released due to a call to put, which may never occur because the object is locked.

**Exercise** Try to run the Java code SemaProducerConsumer.java.
Fixing the nested monitor problem

```java
public FixedSemaBuffer(int size) {
    this.size = size;
    buf = new Object[size];
    items = new Semaphore(0);
    spaces = new Semaphore(size);
}

public void put(Object o) throws InterruptedException {
    spaces.down();
    synchronized(this) { // wait locking monitor until after semaphore passed
        buf[in] = o;
        in=(in+1) % size;
        System.out.println("put");
    }
    items.up();
}

public Object get() throws InterruptedException {
    items.down();
    Object o;
    synchronized(this) {
        o =buf[out];
        buf[out]=null;
        out=(out+1) % size;
        System.out.println("get");
    }
    spaces.up();
    return (o);
}
```
Running the fixed solution

Clearly we have chosen the obvious solution not to lock the monitor until the semaphore statement is passed.

This is what was going on in the code above where `synchronized` is used with argument `this` inside the `put` and `get` methods.

**Exercise**  Try to run the program `FixedSemaProducerConsumer.java`.
A more elegant solution

Although the solution above is correct it is not very elegant so we prefer this Java solution where producer and consumer have the semaphores.

```java
class NiceSemaProducerConsumer {
    final static int Places = 4;

    public static void main(String[] args) {
        Buffer buffer = new NoCondSynchBB(Places);
        Semaphore items = new Semaphore(0);
        Semaphore spaces = new Semaphore(Places);
        SemaProducer producer = new SemaProducer(buffer, items, spaces);
        SemaConsumer consumer = new SemaConsumer(buffer, items, spaces);
        producer.start();
        consumer.start();
    }
}
```
public class NoCondSynchBB implements Buffer {

    protected Object[] buf;
    protected int in = 0; //next index to insert
    protected int out= 0; //next index to take from
    protected int size; //size of the buffer

    public NoCondSynchBB(int size) {
        if (size > 0) this.size = size;
        buf = new Object[size];
    }

    public synchronized void put(Object o) throws InterruptedException {
        buf[in] = o;
        in=(in+1) % size;
        System.out.println("put: " + o);
    }

    public synchronized Object get() throws InterruptedException {
        Object o =buf[out];
        buf[out]=null;
        out=(out+1) % size;
        System.out.println("get: " + o);
        return (o);
    }
}
A more elegant solution

class SemaProducer extends Thread {

    Buffer buf;
    String alphabet= "abcdefghijklmnopqrstuvwxyz";
    Semaphore items, spaces;

    SemaProducer(Buffer b, Semaphore i, Semaphore s) {
        buf = b;
        spaces = s;
        items = i;
    }

    public void run() {
        try {
            int i = 0;
            while(true) {
                spaces.down();
                buf.put(new Character(alphabet.charAt(i)));
                items.up();
                i=(i+1) % alphabet.length();
            }
        } catch (InterruptedException e){}
    }
}
A more elegant solution

class SemaConsumer extends Thread {
    Buffer buf;
    Character c;
    Semaphore items, spaces;

    SemaConsumer(Buffer b, Semaphore i, Semaphore s) {
        buf = b;
        spaces = s;
        items = i;
    }

    public void run() {
        try {
            while(true) {
                items.down();
                c = (Character)buf.get();
                spaces.up();
            }
        } catch(InterruptedException e ){} 
    }
}
Monitors and Semaphores

Running the nice fixed solution

Intuitively, the role of the semaphores are to make sure that the consumer never tries to get from an empty buffer and the producer does not insert into a full buffer.

The role of the synchronization on the put and get methods in the monitor are solely to avoid interference.

Exercise Try to run the Java code `NiceSemaProducerConsumer.java`.
A model of the more elegant solution

Exercise  Run the model `BoundBuf.ag` below in MWB and convince yourself that is is okay.

```plaintext
agent Semaphore0(u,d) = u.Semaphore1<u,d>
agent Semaphore1(u,d) = u.Semaphore2<u,d> + d.Semaphore0<u,d>
agent Semaphore2(u,d) = u.Semaphore3<u,d> + d.Semaphore1<u,d>
agent Semaphore3(u,d) = d.Semaphore2<u,d>
agent Spaces(u,d) = Semaphore3(u,d)
agent Items(u,d) = Semaphore0(u,d)
agent Buf(p,g) = p.Buf<p,g> + g.Buf<p,g>
agent Prod(put,p,s_d,i_u) = 's_d.put.'p.'i_u.Prod<put,p,s_d,i_u>
agent Cons(get,g,s_u,i_d) = 'i_d.get.'g.'s_u.Cons<get,g,s_u,i_d>
agent BB(put,get) = (ˆ p,g,s_d,s_u,i_d,i_u)( Prod<put,p,s_d,i_u> | Buf<p,g> | Cons<get,g,s_u,i_d> | Spaces<s_u,s_d> | Items<i_u,i_d>
```

Jens Chr. Godskesen, IT University of Copenhagen
Exercises

1. Go through the exercises we didn’t make during the lecture.

2. Program a Java class for a one place buffer as a monitor.

3. Redo the one place buffer program above using semaphores instead of condition synchronization.

4. Make a CCS model of a teller machine shared by several people. The machine serves as an account where coins with currencies 25, 50 and 100 can be inserted and withdrawn. Make sure the balance of the account never becomes negative.

5. Program a monitor for the account modeled above.

6. Do the programming lab about synchronization with semaphores.