

Monitors and Semaphores

Version 1.1

Jens Chr. Godskesen
IT University of Copenhagen

©2004, 2005 Jens Chr. Godskesen

February 22, 2005

1 Introduction

So far we talked about synchronization through shared objects and the interference problem. Mutual exclusion solves the problem! *Monitors* and *semaphores* may help to obtain mutual exclusion. In this note we explain the concepts of semaphores and monitors.

The examples of models to be run in the *Mobility Workbench* (MWB) in this note are presented with emphasis on readability and may thus not compile due to extra line breaks.

Part of this note and some of the examples in this note are inspired by [1].

The exercises affiliated with this lecture are to be found in Section 6

2 Monitors

Locks and synchronized methods are programming language specific means for implementing *monitors*. For instance, the Java code below is an example of a monitor.

```
class Counter {  
    private int value=0;  
    public synchronized void update() {value++;}  
    public synchronized int getValue() {return value;}  
}
```

where data is declared to be private and where the methods can only be accessed one at the time since they are declared to be synchronized.

Definition 1 A monitor is a piece of code (a class) characterized by the facts that it

- allows for encapsulation of data
- provides access procedures (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. We will study it in detail in the next section.

2.1 Condition Synchronization

Let's consider an example¹ with a monitor using condition synchronization.

2.1.1 The Carpark example

Assume a carpark with one entrance and one exit; clearly the carpark has a fixed number of parking slots.

We simulate arrival of cars by a process (a thread) and likewise we simulate departure of cars by a thread.

The carpark needs a *controller* which allows cars to enter only when the carpark is not full and dually it permits cars to leave only when the carpark is not empty.

A carpark with three parking slots can be modeled as a parallel composition of three processes `Arr`, `Dep`, and `Con` as

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

(* Con(a,d) controls a carpark 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>)
```

where `arr` and `dep` signals arrival and departure respectively of cars. `Con0` is the state where there are no cars in the carpark, `Con1` indicates one car in carpark, etc.

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

¹Based on a similar example from [1].

```

agent Con(a,d) = Con(a,d,3)
agent Con(a,d,i) = if (i > 0) a.Con<a,d,i-1> + if (i < 3) d.Con<a,d,i+1>

```

however this notion of guarded commands and integer arithmetic is not allowed in MWB.

2.1.2 Implementing the carpark example

How to implement the model above? Which model entities should be *active* (threads) and which should be *passive* (monitors)? It is not always an obvious task to decide upon these matters.

The CarPark may be implemented by

```

public class CarPark {

    final static int Places = 3;

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

```

where Arrivals and Departures are chosen to be active objects (i.e. threads) and CarParkControl is a static monitor.

The two threads are implemented following the model closely by

```

class Arrivals extends Thread {

    CarParkControl carpark;

    Arrivals(CarParkControl c) {carpark = c;}

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

```

and by

```

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){}
    }
}

```

and finally the monitor CarParkControl is implemented by

```

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();
    }
}

```

The reason why arrive() and depart() throws InterruptedException is due to the method wait(), see Section 2.2.

It is essential that CarParkControl uses condition synchronization since it must block calls to arrive() if the carpark is full and also it must block calls to depart() if the carpark is empty. The way condition synchronization is managed is through the use of *waiting queues* that is explained in details in Section 2.2.

Notice, that Con(a, d) from the CCS model in Section 2.1.1 indeed is an abstract model of the implementation of the monitor, CarParkControl.

Exercise Try to run the program CarPark.java

2.1.3 General Rules for Guiding the Translation from Model to Java

The general schema for deriving condition synchronization conditions from guarded actions in the model of a monitor is that

```
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.

For instance, referring to the compact specification of a model of a carpark in Section 2.1.1,

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

is implemented by

```
synchronized void arrive() throws InterruptedException {
    while (i==0) wait();
    --i;
    notifyAll();
}
```

2.2 Waiting queues

Condition synchronization is realized through the use of a *waiting queue*. Any Java object has a waiting queue. Actually it is a waiting bag because due to the Java specification there is no ordering of the elements in the queue.

Any Java object has a waiting queue that is operated through the methods `wait()`, `notify()`, and `notifyAll()`:

- A thread calling the `wait()` method of an object is put into the waiting queue of the object and waits to be notified by another thread (calling one of the notify methods of that object), the thread becomes non-runnable.
- A thread calling an objects `notify()` method wakes up a single randomly chosen thread that is waiting in the queue of that object, the thread being notified becomes runnable.

- A thread calling an objects `notifyAll()` method wakes up all the threads waiting in the queue of that object, all the threads becomes runnable.

Upon calling `wait()` in a monitor the calling thread releases the synchronization lock of the monitor (and becomes non-runnable). When the thread is notified (it becomes runnable) it must reacquire the lock once again before it is allowed to resume execution (before it becomes running).²

A thread *enters* a monitor when it acquires the monitors lock; it *exits* the monitor when it releases the lock.

Exercise Draw a picture illustrating a monitor, its waiting queue, and threads being either running, runnable, or non-runnable. Explain the distinction between `notify()` and `notifyAll()`.

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()`?

Reconsider the `CarPark`-example. Suppose we have a carpark 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 carpark is initially empty and that D_4 is running.

An example of execution may be as follows:

²If the thread is interrupted by another thread while it is waiting, then an `InterruptedException` is thrown but first when the lock of the object is reobtained. Thread interruptions is not dealt with further in this lecture.

Runnable	Running	Vacant Spaces	Queued
A, D ₁ , D ₂ , D ₃	D ₄	2	
A, D ₁ , D ₂ , D ₃		2	D ₄
A, D ₁ , D ₂	D ₃	2	D ₄
A, D ₁ , D ₂		2	D ₃ , D ₄
A, D ₁	D ₂	2	D ₃ , D ₄
A, D ₁		2	D ₂ , D ₃ , D ₄
A	D ₁	2	D ₂ , D ₃ , D ₄
A		2	D ₁ , D ₂ , D ₃ , D ₄
D ₁	A	2	D ₁ , D ₂ , D ₃ , D ₄
D ₁ , D ₂	A	1	D ₂ , D ₃ , D ₄
D ₁ , D ₂		0	D ₃ , D ₄
D ₂	D ₁	0	A, D ₃ , D ₄
D ₂ , D ₃	D ₁	0	A, D ₃ , D ₄
D ₂ , D ₃ , D ₄	D ₁	1	A, D ₄
D ₂ , D ₃ , D ₄		2	A
D ₃ , D ₄	D ₂	2	A, D ₁
D ₃ , D ₄		2	A, D ₁
D ₄	D ₃	2	A, D ₁ , D ₂
D ₄		2	A, D ₁ , D ₂
	D ₄	2	A, D ₁ , D ₂ , D ₃
		2	A, D ₁ , D ₂ , D ₃
		2	A, D ₁ , D ₂ , D ₃ , D ₄

Now all threads are queued and the system is *deadlocked*. Notice, that it was assumed that an awakened thread is not guaranteed to be running.

3 Semaphores

Another mechanism to deal with synchronization problems is a *semaphore* (by Dijkstra in 1968, so historically it predates monitors). One of the major motivations for the development of semaphores was to avoid busy waiting.³ However, as you will discover semaphores are far less structured compared to the more recent monitors.

Semaphores have traditionally been used when implementing operating systems and may be used to implement the programming language mechanism used in monitors.

³In busy waiting the thread would itself continuously check for a condition to become true instead of being notified by someone else.

Definition 2 A semaphore s is an integer variable that

- can take only non-negative values

and has exactly the following 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, hence for a binary semaphore $\text{up}(s)$ is defined by

$\text{up}(s)$: if there are processes suspended on s then wake one of them else $s = 1$

As an example, the lock on Java objects could be implemented using binary semaphores. Notice that a semaphore must be given a non-negative initial value. Also, notice that the operations $\text{down}(s)$ and $\text{up}(s)$ are atomic, e.g. for $\text{down}(s)$ no instructions can be interleaved between the check $s > 0$ and the decrement of s or the suspension of the process that called s . Finally, notice that it is not specified which of the suspended processes $\text{up}(s)$ must wake.

3.1 Example

Consider the following example with two processes that uses a common semaphore to ensure mutual exclusion from their respective critical sections.

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.

It may not always be an easy task to program using semaphores because it is a low level programming style that requires a very strict use of up 's and down 's in order to establishing proper condition synchronization.

Exercise What happens if several up 's were issued after the critical section? Why does it depend as to whether the semaphore is binary or not?

3.2 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?

Using syntactic sugar (not part of MWB) we may define a general semaphore by:

```
agent Sema(u,d) = Semaphore<u,d,1>
agent Semaphore(u,d,i) = if (i > 0) d.Semaphore<u,d,i-1>
                        + u.Semaphore<u,d,i+1>
```

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

3.3 Semaphores in Java

Semaphores may be implemented in Java using a monitor, for instance

```
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();
    }
}
```

Notice, the asymmetry between the two methods. Notification only happens in `up()`. The reason why is that processes never waits for being allowed to increment a semaphore, they only wait to decrement it.

Also, notice that the implementation of `up` doesn't precisely correspond to its definition. According to the definition `up` must wake up a suspended process and only otherwise increment `s`. However, the implementation of `up` starts incrementing `s` and then always wakes up a suspended process if some are suspended. But since any suspended process that becomes running as their first task decrements `s` before releasing the monitor lock the implementation of `up` is equivalent to its definition.

3.4 Semaphores revisited

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 (`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 (`up(s)`).

3.5 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.

4 Bounded Buffers

A *bounded buffer* is a buffer with finite capacity.

Below we give an example of the so called *consumer-producer problem*. The example is used later to explain a common monitor problem, the problem of *nested monitors*.

4.1 Producer-Consumer

Here is an example of a producer-consumer situation:

Example: Suppose a keyboard device driver (a *producer*) that is supplying characters typed at a keyboard to an editor (a *consumer*). Some characters takes long to process (e.g. scroll the screen), hence characters must be buffered. A *type-ahead buffer* is needed.

Next, we provide a Java implementation of a similar system. It consists of two threads, the producer and the consumer, and a monitor, the bounded buffer.

```
public class ProducerConsumer {  
  
    final static int Places = 4;  
  
    public static void main(String[] args) {  
        BoundedBuffer buffer = new BoundedBuffer(Places); //Monitor  
        Producer producer = new Producer(buffer);  
        Consumer consumer = new Consumer(buffer);  
        producer.start();  
        consumer.start();  
    }  
}
```

The producer thread is implemented by

```
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){}  
    }  
}
```

and the consumer thread is implemented by

```

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){}
    }
}

```

The bounded buffer implements this interface

```

public interface Buffer {
    public void put(Object o)
        throws InterruptedException; //put object into buffer
    public Object get()
        throws InterruptedException; //get an object from buffer
}

```

The code next implements the bounded buffer, i.e. a buffer with finite capacity using the interface above.

```

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);
    }
}

```

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.

$$\text{in} == (\text{out} + \text{count}) \% \text{size}$$

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`.

5 Nested Monitors

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.

```

public class SemaProducerConsumer {

    final static int Places = 3;

    public static void main(String[] args) {
        SemaBoundedBuffer buffer = new SemaBoundedBuffer(Places);
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);
        consumer.start();
        producer.start();
    }
}

```

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.

```

public class SemaBoundedBuffer implements Buffer {

    protected Object[] buf;
    protected int in = 0;
    protected int out= 0;
    protected 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);
    }
}

```

5.1 The model of SemaBoundedBuffer

The monitor `SemaBoundedBuffer` from above can be modeled by

```
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> )
```

Notice, that `Buf` models that in the Java code `get` and `put` are synchronized because of the choice between the two actions `p` and `g`.

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>`.

This is a nice example of the usefulness of analyzing models. In case we start of with a model, instead of directly making the Java code, then we may save a lot of time (and money) finding logical errors in our design by analyzing the design model.

In the Java code there is therefore (since the model is an abstract albeit correct model of the Java code) also a deadlock. When the call to `get()` is executed it locks the monitor object because `get()` is synchronized. Then in the body of `get()` the execution is blocked on the semaphore `items` because it cannot be decremented. This blocking can only be released due to a call to `put()` in which `items` is incremented. But such a call cannot occur because the buffer object is locked.

Exercise Try to run the Java code `SemaProducerConsumer.java`.

5.2 Fixing the nested monitor problem

The problem detected above is an instance of the *nested monitor problem*. Clearly an obvious solution is not to lock the monitor until the semaphore statement is passed.

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

First, to illustrate that a new type of buffer is used

```
public class FixedSemaProducerConsumer {  
  
    final static int Places = 3;  
  
    public static void main(String[] args) {  
        FixedSemaBuffer buffer = new FixedSemaBuffer(Places);  
        Producer producer = new Producer(buffer);  
        Consumer consumer = new Consumer(buffer);  
        producer.start();  
        consumer.start();  
    }  
}
```

and here comes the actual changes to solve the problem

```
public class FixedSemaBuffer implements Buffer {  
  
    protected Object[] buf;  
    protected int in = 0;  
    protected int out= 0;  
    protected int size;  
  
    Semaphore items; // counts number of items  
    Semaphore spaces; // counts number of spaces  
  
    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) {  
            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);  
    }  
}
```

Exercise Try to run the Java code `FixedSemaProducerConsumer.java`.

5.3 A more elegant solution to the nested monitor problem

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

```
public 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();  
    }  
}
```

The monitor don't use condition synchronization, it is not needed due to the use of semaphores.

```
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);  
    }  
}
```

Notice next how the producer and the consumer synchronize internally through the use of common semaphores.

```

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){}
    }
}

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 ){}
    }
}

```

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. Hence, 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`.

5.3.1 A model of the elegant solution

Here is a model of the approach followed above where we compared to the model in Section 5.1 now also let the producer and consumer be part of the model.

```
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(p,s_d,i_u) = 's_d.'p.'i_u.Prod<put,p,s_d,i_u>
agent Cons(g,s_u,i_d) = 'i_d.'g.'s_u.Cons<get,g,s_u,i_d>

agent BB = (^ p,g,s_d,s_u,i_d,i_u)( Prod<put,p,s_d,i_u>
                                | Cons<get,g,s_u,i_d>
                                | Buf<p,g>
                                | Spaces<s_u,s_d>
                                | Items<i_u,i_d> )
```

Exercise Run the model `BoundBuf.ag` in `MWB` and convince yourself that it is okay. For instance, you may try to issue `deadlocks BB`.

6 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 values 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.

References

- [1] J. Magee and J. Kramer. *Concurrency — State Models & Java Programs*. World-wide Series in Computer Science. John Wiley & Sons, 1999.