Monitors, Java, Threads and Processes

An object-oriented view of shared memory

- A semaphore can be seen as a shared object accessible through two methods: `wait` and `signal`.

- The idea behind the concept of monitor is to generalize this to other types of objects with other methods.

- If several processes can execute methods on the same object, the interaction between these processes has to be managed:
  - An appropriate level of atomicity has to be imposed in order to guarantee that methods are executed correctly;
  - A mechanism for suspending processes while they wait for a given condition on the state of the shared object is needed.

Monitors : the concept

- A *monitor* is a class used in the context of concurrency. The instances of the class will thus be objects simultaneously used by several processes.

- Defining a monitor is done by defining the corresponding class. We will use a Java inspired syntax.

A class example: stack

```
                   public class Stack {
                     protected static int max = 300;
                     private int nbElements;
                     private Object[] content;

                   public Stack()
                     { nbElements = 0;
                       content = new Object[max];
                     }
public void push(Object e)            public Object pop()
{                                     {
  if (nbElements < max)                 if (nbElements > 0)
    content[nbElements++] = e;             return content[--nbElements];
}                                       else
                                          return null;
                                      }

}
```

Monitors : a first synchronization rule

- Even for a class as simple as `Stack`, the simultaneous execution of class methods by different processes can be problematic.

- A first synchronization rule imposed in the context of monitors is thus the following:

  > **The methods of a monitor are executed in mutual exclusion.**

Java Note. In Java, mutual exclusion of methods has to be explicitly specified withe the keyword `synchronized`.

The producer-consumer problem in the context of monitors

This is quite naturally solved with the following class.

```
public class PCbuffer
{
  private Object buffer[];     /* Shared memory    */
  private int N;               /* Buffer capacity */
  private int count, in, out;  /* nb of elements, pointers */

  public PCbuffer(int argSize)
   {                           /* creation of a buffer of size
                                  argSize */
    N = argSize;
    buffer = new Object[N];
    count = 0; in = 0; out = 0;
  }
```

```
  public synchronized void append(Object data)
  {
    buffer[in] = data;
    in = (in + 1) % N; count++;
  }

  public synchronized Object take()
  {
    Object data;

    data = buffer[out];
    out = (out + 1) % N;
    count--; return data;
  }
}
```

The required synchronization when the buffer is empty or full is obviously missing.

Monitors: synchronization with wait queues

- In order to suspend processes when they cannot execute the required operation, we will use *wait queues*.

- These queues are managed using three operations.

  - `qWait()` : suspends the process executing this operation and places it in the queue.
  - `qSignal()` : frees the first process in the wait queue (has no effect if the queue is empty).
  - `qNonempty()` : tests that the queue is not empty.

- When a process is suspended and placed in the queue with the operation `qWait()`, it relinquishes the mutual exclusion linked to the execution of a monitor method.

Wait queues viewed as a class

Wait queues are viewed as objects instantiated form the following class whose full definition will be given later.

```
public class Waitqueue
{     ....            /* data (to be defined) */
   public Waitqueue()
   {  ....            /* constructor (to be defined) */
   }
   public void qWait()
   {  ....            /* wait operation */
   }
   public void qSignal()
   {  ....            /* signal operation (allows a waiting process
                         to resume its execution) */
   }
   public boolean qNonempty()
   { ....             /* tests that the queue is not empty */
   }
}
```

The producer-consumer problem with wait queues

To synchronize the producers and consumers, two wait queues will be used: one for the processes waiting for an element to consume, one for the processes waiting for available space in the buffer.

```
public class PCbuffer
{
  private Object buffer[];     /* Shared memory   */
  private int N ;              /* Buffer capacity */
  private int count, in, out;  /* nb of elements, pointers */
  private Waitqueue notfull, notempty; /* wait queues */

  public PCbuffer(int argSize)
    { N = argSize;              /* creating a buffer of size
      buffer = new Object[N];      argSize */
      count = 0; in = 0; out = 0;
      notfull = new Waitqueue(); notempty = new Waitqueue();
    }
```

```
  public synchronized void append(Object data)
  { if (count == N) notfull.qWait();
    buffer[in] = data;
    in = (in + 1) % N; count++;
    notempty.qSignal();
  }

  public synchronized Object take()
  { Object data;

    if (count == 0) notempty.qWait();
    data = buffer[out];
    out = (out + 1) % N;
    count--; notfull.qSignal();
    return data;
  }
}
```

The operation `qSignal` has no effect if the queue is empty. Thus this solution avoids the double signaling problem encountered when using binary semaphores.

Priorities when executing a `qSignal` operation

- When a `qSignal` operation is executed, the process at the head of the wait queue is freed, but *a priori* this does not mean that it will immediately resume its execution.

- This is potentially problematic since the situation that triggered the `qSignal` operation (for instance a nonempty buffer) might no longer be true when the freed process actually resumes its execution (for instance because another process has taken the only remaining element in a nonempty buffer).

- To avoid this, the following rule is imposed on monitor synchronization:

  > **Immediate resumption: the process freed by a `qSignal` operation has priority over all other processes attempting to execute and operation on the object for which the wait queue is used.**

- How about the process executing the `qSignal` operation ?
  - The process executing the `qSignal` operation has priority to finish executing the method it is currently in, as soon as the freed process has completed its own monitor method.

Semaphores viewed a monitors

With the wait queue mechanism, semaphores can easily be implemented as monitors. This also allows defining other operations on semaphores, in particular testing the number of processes waiting on the semaphore, which we will make use of later.

```
public class Semaphore
{ int value;                /* The value of the semaphore */
  int nbWait = 0;           /* The number of waiting processes */
  Waitqueue queue;

  public Semaphore(int inival)
  {  value = inival;
     queue = new Waitqueue();
  }
```

```
  public synchronized void semWait()
  {  if (value <= 0)
     {  nbWait++;
        queue.qWait();
        nbWait--;
     }
     value--;
  }

  public synchronized void semSignal()
  {  value++; queue.qSignal();
  }

public synchronized int semNbWait()
  {  return nbWait;
  }
```

Note that immediate resumption is essential for this implementation to be correct.

Implementing wait queues

- Obviously, implementing wait queues requires a mechanism for suspending processes that has to be provided by the program execution environment.

- We will temporarily ignore this problem, assuming that we have access to semaphores with a direct implementation that does not use queues as above.

- Immediate resumption requires special attention since it establishes a link between the wait queues and mutual exclusion of the monitor methods. We will first examine an implementation that does not guarantee immediate resumption, taking this constraint into account at a later stage.

Implementing wait queues without immediate resumption

A queue is implemented with a semaphore (assumed to be FIFO) whose value is always 0.

```
public class Waitqueue
{  private int qcount = 0;  /* The number of waiting processes */
   private Semaphore Qsem;

   public Waitqueue()
   {  Qsem = new Semaphore(0);
                          /* creating the semaphore initialized to 0 */
   }

   public void synchronized qWait()  /* wait operation */
   {  qcount++; Qsem.semWait(); qcount--;
   }
```

```
   public void synchronized qSignal()  /* signal operation  */
   {  if (qcount > 0) Qsem.semSignal();
   }

public boolean synchronized qNonempty()
   {  return qcount == 0;     /* testing if the queue is non empty */
   }
}
```

Independently of the problem linked to immediate resumption, this solution is prone to potential deadlocks linked to the multiple use of the *synchronized* attribute.

Note on Java : `synchronized and locks`

- A method declared as `synchronized` is executed in mutual exclusion with respect to the other methods operating on the same object.
- This can be understood by considering that there is a *lock* (possibly implemented by a semaphore) linked to the object.
- When a `synchronized` method calls a `synchronized` method of another object, locking on this new object also occurs.
- A problem appears when a process is suspended and enters a wait state. When this happens, locks must be freed, but this is hard to do if a sequence of lock operations have been performed.
- In Java, the basic wait operation (see further down) only frees the lock on the current object.

Implementing queues with immediate resumption

- To solve both the problem of immediate resumption and that of multiple locks, we will manage mutual exclusion between monitor methods explicitly with a semaphore.
- When a process will be suspended, the operation `qWait` will explicitly free the mutual exclusion associated with the method calling the operation `qWait`.
- In order to do this, an argument giving the mutual exclusion semaphore to be freed will be given to `qWait`, as well as to `qSignal` in order to be able to handle immediate resumption.

An implementation of queues with immediate resumption (continued)

```
public class Waitqueue
{  private int qcount = 0;  /* the number of waiting processes */
   private Semaphore Qsem;

   public Waitqueue()
   {  Qsem = new Semaphore(0);
                       /* creating a semaphore initialized to 0 */
   }

   public void  qWait(Semaphore Mutsem)
                      /* wait operation */
   {  qcount++; Mutsem.semSignal(); Qsem.semWait(); qcount--;
   }
```

```
   public void  qSignal(Semaphore Mutsem)
                           /* signal operation  */
   {  if (qcount > 0) {
        Qsem.semSignal(); Mutsem.semWait();
      }
   }

public boolean  qNonempty()
   {  return qcount == 0;   /* testing if the queue is nonempty */
   }
}
```

To prevent deadlocks, mutual exclusion is not imposed on the wait queue operations. Undesired interference is nevertheless impossible since a queue is only used by a single shared object and since the methods operating on this object are already executed in mutual exclusion.

The buffer using queues implemented by semaphores

```
public class PCbuffer
{
  private Object buffer[];     /* Shared memory    */
  private int N ;              /* Buffer Capacity */
  private int count, in, out;  /* nb of elements, pointers */
  private Waitqueue notfull, notempty; /* wait queues */
  private mutex Semaphore;     /* mutual exclusion semaphore */

  public PCbuffer(int argSize)
  {  N = argSize;
     buffer = new Object[N];
     count = 0; in = 0; out = 0;
     notfull = new Waitqueue(); notempty = new Waitqueue();
     mutex = new Semaphore(1);
  }
```

```
  public  void append(Object data)
  { mutex.semWait()
    if (count == N) notfull.qWait(mutex);
    buffer[in] = data;
    in = (in + 1) % N; count++;
    notempty.qSignal(mutex);
    mutex.semSignal();
  }

  public Object take()
  { Object data;

    mutex.semWait()
    if (count == 0) notempty.qWait(mutex);
    data = buffer[out];
    out = (out + 1) % N;
    count--;
    notfull.qSignal(mutex);
    mutex.semSignal(); return data;
  }
}
```

Priority of processes having executed an operation qSignal

- In the implementation above, priority is not given to the process having executed the operation `qSignal`.
- For doing this, a second mutual exclusion semaphore `urgent` will be used.
- The operation `qWait` will be called with the mutual exclusion semaphore, or with the semaphore `urgent`, depending on whether processes are waiting on the latter of not.
- To achieve this, the number of processes waiting on `urgent` will be counted explicitly. Indeed, a call to `semNbWait(urgent)` does not count the processes that have executed an operation `qSignal`, are going to wait on `urgent`, but have not yet done so.
- At the end of a method, the operation `semSignal` is executed on `urgent` if a process is waiting on this semaphore; if not the the operation is executed on the mutual exclusion semaphore.

An implementation of wait queues with immediate resumption and priority to the "signaling" process

```
public class PCbuffer
{
  private Object buffer[];     /* Shared memory    */
  private int N ;              /* Buffer capacity  */
  private int count, in, out;  /* nb of elements, pointers */
  private Waitqueue notfull, notempty; /* wait queues */
  private Semaphore mutex, urgent;     /* semaphores */
  private int urcount;         /* nb of processes waiting on
                                  urgent */


  public PCbuffer(int argSize)
  { N = argSize;
    buffer = new Object[N];
    count = 0; in = 0; out = 0; urcount = 0;
    notfull = new Waitqueue(); notempty = new Waitqueue();
    mutex = new Semaphore(1); urgent = new Semaphore(0);
  }
```

```
public  void append(Object data)
{ mutex.semWait()
  if (count == N) {
     if (urcount > 0) notfull.qWait(urgent);
       else notfull.qWait(mutex);
    }
  buffer[in] = data;
  in = (in + 1) % N; count++;
  urcount++;
  notempty.qSignal(urgent);
  urcount--;
  if (urcount > 0) urgent.semSignal();
     else mutex.semSignal();
}
```

```
  public  Object take()
  { Object data;

    mutex.semWait()
     if (count == 0) {
       if (urcount > 0) notempty.qWait(urgent);
         else notempty.qWait(mutex);
      }
    data = buffer[out];
    out = (out + 1) % N;
    count--;
    urcount++;
    nofull.qSignal(urgent);
    urcount--;
    if (urcount > 0) urgent.semSignal();
       else mutex.semSignal();
    return data;
  }
}
```

Note on Java : Implementing semaphores

- To avoid circularity in the implementation of monitors, an implementation of semaphores that does not use the wait queues we have defined is needed.

- Java has the necessary primitive to make such an implementation possible.
  - First, the `synchronized` attribute of a method guaranteed that it will be executed in mutual exclusion.
  - Furthermore, Java provides methods for suspending and reactivating processes: `wait()`, `notify()` et `notifyAll()`.

Note on Java : `wait` and `notify`

- `wait()` : Suspends the current process and frees mutual exclusion with respect to the current object.

- `notify()` : Selects a process waiting on the current object and makes it executable. The selected process must reacquire mutual exclusion (there is no immediate resumption).

- `notifyAll()` : Similar to `notify()`, but makes executable all processes that are waiting on the current object.

A first implementation of semaphores in Java

```
public class Semaphore
{ private int value;                /* the value of the semaphore */
  private int nbWait = 0;           /* nb waiting */

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

  public synchronized void semWait()
  {  while (value <= 0)
     {  nbWait++;
        wait();
     }
     value--;
  }
```

Since there is no immediate resumption, the value of the semaphore has to be tested after returning from the call to `wait()`.

```
  public synchronized void semSignal()
  {  value++;
     if (nbWait > 0)
       {  nbWait--; notify();
       }
  }

  public synchronized int semNbWait()
  {  return nbWait;
  }
}
```

The operation `nbWait--` is done in `semSignal()` just before `notify()` because, given that there is no immediate resumption, executing this operation in `semWait()` after `wait()` could lead to an incorrect value being returned by `semNbWait()`.

The semantics of `wait()` and `notify()` imply that this implementation is not fair.

A fair implementation of semaphores in Java

- To obtain a fair implementation, the wait queue has to be explicitly implemented.
- The queue will be a queue of objects on each of which a single process will be waiting.
- The fact that there is no fairness for `notify()` operations will thus not be a problem.

A fair implementation of semaphores in Java (continued)

The class `Queue` implements a classical wait queue.

```
public class SemaphoreFIFO
{ private int value;               /* the value of the semaphore */
  private int nbWait = 0;          /* nb waiting */
  private Queue theQueue;          /* The explicit wait queue */

  public SemaphoreFIFO(int inival)
  {  value = inival;
     theQueue = new Queue();
  }
```

```
public  void semWait()
{  Semaphore semElem;
   synchronized(this){
     if (value == 0)
       { semElem  = new Semaphore(0);
         theQueue.enqueue(semElem);
         nbWait++;
       }
       else
         {
           semElem  = new Semaphore(1);
           value--;
         }
   } /* synchronized */

   semElem.semWait();
}
```

The operation `semElem.semWait()` is withdrawn from the mutual exclusion to avoid double locking. Note the different initialization of `semElem` depending on whether `value` is at `0` or not.

```
  public synchronized void semSignal()
  {  Semaphore semElem;
     if (!theQueue.empty()){
        semElem = theQueue.dequeue();
        nbWait--; semElem.semSignal();
     }
       else value++;
  }

  public synchronized int semNbWait()
  { return nbWait;
  }
}
```

The operation `semSignal()` frees the process blocked on the first element in the queue, if there is one. If not `value` is incremented.

Processes and Threads

In Java one talks of *Threads* and not processes. What is the difference?

- A "thread" corresponds to an execution having its own control flow.
- Threads can be implemented by processes, but this has several drawbacks.
  - Processes operate in distinct virtual spaces. Sharing objects between threads would thus be rather cumbersome to organize.
  - Given the strong separation between the contexts of different processes, switching from one process to another is a heavy and thus slow operation.

Processes and Threads (continued)

- Several current operating systems provide support for threads or "light processes". These processes differ from traditional processes being designed to cooperate (sharing for example the same virtual space) rather than working in completely distinct contexts.
  - System threads have appeared as a tool to make use of parallel machines.
  - Java threads can be implemented by system threads.
- Java threads can also be implemented within a system process by including a simple task manager in the code to be executed.