Monitors

With semaphores, shared variables are global to all processes. To understand a concurrent program, one needs to examine the entire program. If a new process is added, the programmer needs to examine that the process uses the shared variables and semaphores correctly.

Monitors are program modules that provide a data abstraction mechanism: they encapsulate the shared data and provide operations to manipulate the shared data. These operations are the **only** means to access the data. A process can only access the data via these operations or procedures.

Mutual exclusion is provided automatically. Synchronization is provided through the use of condition variables, which are used in a manner similar to semaphores.

The general form of a monitor is

```
monitor mname {
   Shared monitor variables
   condition variables
   initialization code
   monitor_functions
}
```

The idea behind this construct is that the variables declared in the monitor are only available through the defined functions; therefore access is controlled, making it easier to ensure mutual exclusion for processes or threads executing one of the monitor functions.

Condition variables are special variables that are used to block a thread or process that needs to wait for some event or condition. A process that waits on a condition variable release the exclusive access it had while it was executing inside the monitor. Some other thread can then enter the monitor and perhaps change the state of the monitor. If conditions are right, the 2nd thread can now *signal* the first one that it can at some point continue its execution. So essentially, a condition variable is a list of waiting processes.

Typically, condition variables are declared in a form similar to

```
condition c;
```

Then the operations defined on the condition variable are

```
wait(c)
```

which places the process at the end of `c`'s delay queue. To allow other processes to enter the monitor, wait also releases the exclusive access to the monitor.

The waiting process is awakened when some other process enters the monitor and executes

```
signal(c)
```

If `c`'s queue is not empty, this will release the process at the front of the delay queue. That process continues its execution at some future time in the monitor.

The big question becomes: when a thread signals another, who continues ? The process that signaled, or the process that was waiting ? They can't *both* continue at the same time, because then we would no longer have mutual exclusion in the monitor. Different (languages | monitors | environments supporting monitors) offer

different possibilities:

- Java and Mesa say that there is no reason to block the process that signals, so it should continue. This is called Signal and Continue (SC).
- Concurrent Euclid and Modula say that the process that was waiting on a condition variable is the one that gets access to the monitor. This is called Signal and Wait (SW).
- Pascal Plus guarantees that not only should the waiting process continue, but the process that signals should continue as soon as the awakened process either exits the monitor or waits again. This is called Signal and Urgent Wait (SU).
- Finally, Concurrent Pascal decides that the process that signals must exit the monitor. This is called Signal and eXit (SX).

The wait and signal operations on condition variables are similar to the `P` and `V` operations on semaphores: `wait`, like `P`, delays a process, and `signal`, like `V`, awakens a process. However, there are important differences:

1. `signal` has no effect if there is no process delayed on the condition variable.
2. `wait` always delays, even if there were prior signals.

An example of a monitor, implementing a bounded buffer for the Consumers/Producers problem in pseudo-code, could look like this:

```
monitor bounded_buffer {
   int buf[N] = new int[N];        // our messages
   int front=0, rear=0, count=0;   // ptrs for Cons, Prod, msgs
   condition not_full;             // signaled when count < N
   condition not_empty;            // signaled when count > 0

   // called by Producers
   void deposit(int data) {
      while (count == N) wait(not_full);
      buf[rear]= data;
      rear = ++rear % N;
      count++;
      signal(not_empty);
   }

   // Called by Consumers
   int fetch() {
      while(count == 0) wait(not_empty);
      int o = buf[front];   // return this data
      front = ++front % N;
      count--;
      signal(not_full);
      return o;
   }
}
```

Monitors in JR

JR uses a preprocessor to translate code written in "monitor format" into JR. The processor, called `m2jr` is somewhat primitive, but does have some nice features, such as the support for Signal and Continue, Signal and Wait, Signal and Urgent wait, and Signal and eXit (The command `man $JR_HOME/man/m2jr.1` will give a quick reference on how to use).

Using the `m2jr` translator, the bounded buffer code for the Consumer/Producer problem would look something like this:

```
// this monitor uses SC
_monitor BB {
   _var int [] buf = new int [N]; // data we want shared
   _var int front = 0;  _var int rear = 0;      // pointers for Cons, Prod resp.
   _var int count = 0;                          // number of messages
   _condvar not_full;                           // signaled when count < N
   _condvar not_empty;                          // signaled when count > 0

   // called by producers
   _proc void deposit(int data) {
      while (count == N) {
         _wait(not_full);
      }
      buf[rear] = data;
      rear = (rear+1) % N;
      count++;
      _signal(not_empty);
   }

   // called by consumers
   _proc int fetch() {
      while (count == 0) {
         _wait(not_empty);
      }
      int result = buf[front];   // return this data
      front = (front+1) % N;
      count--;
      _signal(not_full);
      _return result;
   }
}
```

This text would be placed in a file called `BB.m`. To translate to JR, type `m2jr BB.m`. This produces a file called `BB.jr` which is then compiled normally by JR.

The Consumer and Producer processes are then in separate classes; an on object BB is instantiated to produce the monitor that is then used by the processes; for example

```
import edu.ucdavis.jr.JR;
public class BBMain {
   // number of fetchers and depositors (each) processes
   private static final int N = 6;
   // the monitor
   private static BB bb = new BB("bb");
   public static void main(String [] args) {
   }
   private static process prod( (int i = 1; i <= N; i++) ) {
      System.out.println(i + " prod before");
      bb.deposit(i);
      System.out.println(i + " prod after");
   }
   private static process cons( (int i = 1; i <= N; i++) ) {
      System.out.println(i + " cons before");
      int got = bb.fetch();
      System.out.println(got);
      System.out.println(i + " cons after");
   }
}
```

Note that there is automatic constructor that takes a `String` to "name" the monitor. This is done automatically by the translator. You may define your own constructors, of course.