Transforming Monitors into Servers

Remember that a monitor is a resource manager or a process synchronizer. It encapsulates permanent variables that record the state of the resource, and it provides methods that give access to the resource. The methods execute with mutual exclusion and use condition variables for synchronization. We will show how to transform a monitor into an active server which simulates the monitor.

The permanent variables become the server's variables, and therefore the server is responsible for maintaining a consistent state of these variables exactly like the monitor was responsible for its variables. Then it is a matter of implementing the original method using messages.

The clients simulate a method invocation by "sending" the input parameters and "receiving" the results, via communication channels.

In general, though, we need to allow for multiple methods, and to allow for synchronization. We could have the server use one channel per method, but we don't know what order the methods are invoked. We could have one thread per method, and have each thread wait for an invocation on "its" method, but then we still need to provide synchronization between the threads that are modifying shared variables. In addition, we need to implement somehow the monitor's condition variables.

To illustrate all this, let's use a simple resource allocator implemented as a monitor, and see what we need to do to implement it as a server.

First, the monitor:

```
monitor ResourceAllocator {
   final int MAXUNITS = some initial value        // max number of units of this resource
   int avail = MAXUNITS;                          // the number of resource units available
   SetOfUnits units = new SetofUnits(MAXUNITS);   // a set representing one instance of each resource
   Condition free = new Condition();              // make requestor wait if there are no resource available

   // called by a client to obtain a resource
   int acquire() {
      if (avail == 0) free.await();
      else avail--;
      return units.remove();                      // returns the Id of one available unit
   }

   // called by client when finished with the resource
   void release(int id) {
      units.add(id);                              // add the unit back to the set of resources now available
      if free.empty() avail++;                    // if no one needs this, increment
      else free.signal();                         // otherwise, let the next waiting client have it
   }
}
```

Now for the server code:

```
// this class replaces the monitor allocating some resource
class ResourceAllocator {

   final int MAXUNITS = some initial value        // max number of units of this resource
   int avail = MAXUNITS;                          // the number of resource units available
   SetOfUnits units = new SetofUnits(MAXUNITS);   // a set representing one instance of each resource

   public enum REQTYPE { ACQ, REL };              // representing the 2 methods of the monitor
   RequestQueue waiting = new RequestQueue();     // This request queue takes cap int() as elements

   // This is the communication channel the client sees
   public op void request(REQTYPE,                // which "method" is the client "calling"
                          cap void (int),         // the capability to use in case of an "ACQ"
                          int);                   // the unit that was returned in case of a "REL"




   // process code is on following page
```

```
    // now implement the monitor code //
    process server {
       cap void (int) client;                        // a client requesting or returning a resource
       REQTYPE req;                                  // which of the above it is
       int id;                                       // a unit id

       while (true) {
          receive request(req, client, id);         // wait for a client to invoke a method of the monitor
          if(req == ACQ) {                          // client wants to acquire a resource
             if (avail == 0)
                waiting.enqueue(client);
             else {
                avail--;
                id = units.remove();                // get some resource to return to client
                send client(id);
             }
          else if (request == REL) {
             if (waiting.isEmpty()) {           // simply put back the resource, none is waiting
                avail++;
                units.add(id);
             }
             else {                                 // someone is waiting for this resource
                client = waiting.dequeue();         // get a waiting client
                send client(id);                    //
             }
          else { // handle bad message type
         }
       }  // while
    } // process

} // end of class
```