Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

From monitors to servers
Active servers

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

What features exists in a monitor?

- What features exist in a monitor that we'll need to implement with a server?
  - (Global) State of the monitor
  - Mutual exclusion of the monitor
    - Only one client can be processed at time in a server?
  - Methods of the monitor
  - Conditional variables for synchronization

What is a monitor?

- Resource manger
- Process synchronizer

What is encapsulated in a mon.?

- Permanent variables that record the state of the **resource**
- Methods that give access to the **resource**

N.B.

- *Methods execute with mutual exclusion*
- *Methods use condition variables for synchronization*

Transformation of a monitor to a Server

- The permanent variables become the server's variables
  - Therefore the server is responsible for maintaining a consistent state of these variables exactly like the monitor was responsible for its variables

- Implementing the original methods using messages
  - The clients simulate a method invocation by **sending** the input parameters and **receiving** the results, via communication channels

Additional challenges for the transformation

Multiple methods / synchronization / condition variables

- We could use one channel per method
  - On the other hand we don’t know the message invocation order
- One thread per method, the thread waits for the invocation of ”its” method
  - Synchronization between threads that could modify shared variables
- Monitor’s condition variables must also be implemented somehow

Example

- Transforming Monitors into Servers

- *To illustrate all (including the challenges) 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.*

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
   }
}
```

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 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 (req == 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
```

Server code

Client code

```
process client {
   op void (int) response_ch;                    // a client requesting or returning a resource   int MY_ID;
   int resource_id;

   send request(REQTYPE.ACQ, response_ch, 0);    // call for the resource request
   receive response_ch(resource_id);             // waiting for the resource

   # use the resource <resource_id>

   # and finally release it
   send request(REQTYPE.REL, resource_id, ressource_id);
}
```

Duality between monitor and message passing

Monitor-based program	Message-passing program
Permanent variable	Local server variable
Procedure identifier	*Request* channel and operations kinds
Procedure call	**send** *request;* **receive** *reply*
Monitor entry	**receive** *request*
Procedure return	**send** *reply*
wait statement	Save pending request
signal statement	Retrieve and process pending requests
Procedure bodies	Arms of case statement on operation kinds