Lecture 20
Concurrent Programming

7th November 2002

Rendezvous

The rendezvous mechanism was proposed in the late 1970s during the initial competition to develop the Ada programming language. It is reasonable to suppose that the design of rendezvous was influenced by the CSP design, but I'm not sure about that. Like RPC, rendezvous provides a mechanism for a client to invoke an operation in a different context, but unlike RPC the client's operation is serviced by synchronizing with an already-existing process.

Thus, on the client side, rendezvous is very similar to RPC – in both cases the invocation of the remote operation behaves like a procedure call. On the server side, a client's call is handled by an *in* statement (*accept* or *select* statement in Ada). In MPD:

```
in <opname>(<formal parameters>) -> S; ni
```

In Ada:

```
accept <opname> (<formal parameters>) do
   S
end
```

In both MPD and Ada, parameters may be either value parameters or result parameters: value parameters are copied to the in or accept statement at the start of the rendezvous; result parameters are copied to the caller at the end.

To complete the picture we need a way for a server process to offer a choice of operations. In MPD the notation (p. 375) is derived from guarded commands, while Ada uses select (p.400). The MPD implementation of choice also has additional capabilities for prioritizing which branch of the *in* will be chosen and for enabling and disabling branches based on parameter values as well as local state.

Notice that rendezvous does *not* provide selective communication in the generality that we are implementing in project 2: a client can only attempt to perform a single operation at a time.

Example:

A server implementing a bounded buffer:

```
module BoundedBuffer
   op deposit(T), fetch(result T);
body
   process Buffer{
      T buf[n];
      int front=0, rear=0, count=0;
      while (true) {
         in
            deposit(item) && count<n ->
               buf(rear)=item; rear=(rear+1) mod n; count += 1;
         []
            fetch(item) && count>0 ->
               item=buf(front); front=(front+1) mod n; count=count-1;
         ni
      }
end BoundedBuffer
```

A clients of this bounded buffer invoke operations just as if they were normal procedure calls:

```
process Producer {
   T foo;
   while ... {
      ...
      call BoundedBuffer.deposit(foo)
      ...
   }
}

process Consumer {
   T bar;
   while ... {
      ...
      call BoundedBuffer.fetch(bar)
      ...
   }
}
```

Implementation of rendezvous

Suppose that you were asked to implement a run-time mechanism to support rendezvous. You would have to provide mechanisms to marshall and unmarshall arguments and results, of course. If you had mechanisms to turn argument lists and result lists into message (and vice versa) how would you implement rendezvous?

On the client side, we might tell the compiler writer to translate

```
call M.op(args)
```

to

```
channel M;
...
M.send(marshall(value args))
M.recv(unmarshall(result args))
```

On the server side, let's translate the in statement in

```
module M;
body
process server {
   ...
   in
      op1(args1) and B1 -> S1;
   []
      op2(args2) and B2 -> S2;
   [] ...
      opn(argsn) and Bn -> Sn;
   ni
   ...
}
end M
```

into

```
sl = new SelectionList();
re1 = new RecvEvent(); re2 = new RecvEvent(); ... ren = new RecvEvent()
if B1 then sl.add(re1);
if B2 then sl.add(re2);
...
if Bn then sl.add(ren);
```

```
res = sl.select()
if res=re1 then S1(unmarshall(re1.getObject())
elif res=re2 then S2(unmarshall(re2.getObject())
...
elif res=ren then Sn(unmarshall(ren.getObject())
fi
```

Optimizations to the selective communications implementation suggested by this example: we'd like to not allocate a new SelectionList and collection of events for each operation invocation. Suggestions in the project already make the lists and events reusable, but here membership for any particular call of select depends on the boolean guards; perhaps we'd like to allow events to be *enabled* or *disabled* while remaining in a list.

In the MPD version of rendezvous the boolean guards can depend on values in the incoming messages. How could our synchronous communication framework be extended to accomodate that?

Project notes

Can sync be virtual in CommEvent? yes.

Can't I just call c.Recv() and c.Send() from the event sync method? yes, but implementing the functions as suggested in the implementation notes will move you farther along toward what is needed for select. Go over how poll and block are just pieces of the existing Send and Recv, pulled out and made accessible. The Send and Recv operations of Channel are no longer needed after this is done and can be removed from the program.

Ryan's correction to the barrier solution from last time

Ryan pointed out that the barrier code in the lecture notes from last lecture doesn't work correctly. The offerred solution was

```
Init:
OUT ("Barrier", 0)

1 Process (Arbitrary):
IN("Barrier", ?ct)
OUT("Barrier", ct+1)
IN("Barrier", N)
OUT("Barrier2", N-1)
IN("Barrier2", 0) // Waits for count down on second barrier
OUT("Barrier", 0) // Restarts the first Barrier
```

```
Other Processes:
IN("Barrier", ?ct)
OUT("Barrier", ct+1)
IN("Barrier2", ct)
OUT("Barrier2", ct-1)
```

The problem is that the "other processes" code might read a value of 0 in its first IN statement. If that happens there is a race between its second IN statement and the IN("Barrier2",0) of the special process. Regardless of which one wins, deadlock follows.

Ryan's correction which he provided before class but which I didn't notice at the time is:

```
Init:
OUT ("Barrier", 1)

1 Process (Arbitrary):
IN("Barrier", N) //Waits for all of the processes to reach the barrier
OUT("Barrier2", N-1) // Starts the count down (which releases all of the
processes
IN("Barrier2", 0) // Waits for count down on second barrier
OUT("Barrier", 1) // Restarts the first Barrier

Other Processes:
IN("Barrier", ?ct)
OUT("Barrier", ct+1)
IN("Barrier2", ct)
OUT("Barrier2", ct-1)
```