Chalmers | GÖTEBORGS UNIVERSITET
David Sands, Datavetenskap, D&IT

2003-10-24
PP

Concurrent Programming TDA211/INN390/TDA380

Friday 24 October, 14.15 – 18.15, M.

(including solutions to programming problems)

Question 1. (a) Give an example (in words) of

i. a safety property

ii. a liveness property

that you would expect to hold for a correct solution to the train controller lab (lab 1). *(2p)*

(b) In Java, why do calls to `wait()` usually occur in a while loop. *(2p)*

(c) Explain the main differences and similarities between monitors and Ada's protected object construct. *(4p)*

Question 2. For the purposes of this question, assume that there is a tuplespace and the standard tuplespace operations described below.

The Linda Tuplespace Primitives Assume you have tuplespace primitives `in` and `out`. These work in the usual way. The specific syntax that you should assume in this question is as given in the following examples:

- If variable *x* has value 42, then `out(27,"semaph",x)` will place the tuple `(27,"semaph",42)` into the tuplespace.
- The `in` operation blocks until a suitable tuple is found. For example, `in(?y,99)` will block until a matching tuple is found. Assuming that `y` is of string type, an example of a tuple which matches this `in` operation is `("hello",99)`. If `("hello",99)` were present in the tuplespace, then the above `in` operation could remove it, after which the variable `y` would be bound to the value `"hello"`.

The Question Suppose we have two ops

```
op Service1(int)    returns int;
op Service2(string) returns bool;
```

which are serviced by a server process **S**. The body of S has the following simple structure:

```
while (true){
  in
    Service1(i) returns res1 ->
          res1 = PerformService1(i) (* definition not given here *)
  []
```

```
    Service2(s) returns res2 ->
          res2 = PerformService2(s) (* definition not given here *)
  ni
}
```

This question is about how this form of rendezvous could be implemented via a tuplespace. Rewrite the server code above, and show how a typical client call, e.g., `MyInteger = Service1(i)` should be rewritten so that the same synchronisation behaviour is achieved using the tuplespace *instead of* a rendezvous.

Your solution should use no other synchronisation mechanism than the tuplespace operations described above, and your solution should assume that the tuplespace is the only shared memory between the clients and the server. *(6p)*

SOLUTION

Assume each process has its own unique id, stored in a constant myid.

Then

```
   MyInteger = Service1(i)
```

is replaced by

```
   out(myid,"service" 1, i, null)
   in(myid,"reply", 1, ?MyInteger)
```

and similarly for MyBool = Service2(s):

```
   out(myid,"service" 2, null, s)
   in(myid,"reply", 2, ?MyBool)
```

The server code becomes

```
while(true){

  in(?id, "service", ?servicetype, ?int_param, ?string_param)

  if (servicetype == 1) {
    out(id, "reply", servicetype, PerformSerice1(int_param))
  }
  else if (servicetype == 2) {
    out(id, "reply", servicetype, PerformSerice2(string_param))
    } # else do nothing, or put the tuple back
}
```

Question 3. A factory is divided up into *k* work areas. Factory robots enter and leave a work area using the following code skeleton:

```
request_entry(j)
enter(j); work() ; leave(j)
notify_leave(j)
```

where $1 \leq j \leq k$. The code for `enter(j)` causes the robot to physically enter the work area – and `leave(j)` causes it to leave. This question concerns only the code for `request_entry(j)` and `notify_leave(j)`, which are purely for synchronisation purposes. The `request_entry(j)` call is a potentially blocking operation which prevents the robot from entering the work area when either

- more than 20 robots are currently in that work area, or
- if there are more than 100 robots currently working in the whole factory.

Other than that, `request_entry` should not block a robot unnecessarily.

(a) Provide an informal specification of `request_entry(j)` using an **await** statement. *(2p)*

(b) To what extent does the following semaphore-based code solve the problem? Explain your answer.

```
sem factory = 100
sem work_area[k] = [k]20 # an array of k semaphores each initialised to 20

procedure request_entry(int j){
  P(factory)
  P(work_area[j])
}

procedure notify_leave(int j){
  V(factory)
  V(work_area[j])
}
```

(3p)

(c) Provide an implementation of `request_entry` and `notify_leave` using a *monitor*. Use Java, or MPD psudocode, assuming signal-and-continue monitor semantics. *(7p)*

```
SOLUTION (c)

monitor

op request_entry(int)
op notify_leave(int)

body

condition space_available

int bots[k] = [k] 0
```

```
int factory = 0

proc request_entry(j){
  while(bots[j] > 20 and factory > 100) {
     wait(spaces_available)}
  bots[j]++; factory++
}

proc notify_leave(j){
  bots[j]--; factory--
  signal_all(space_available)
}

end
```

Question 4. A normal synchronous channel allows values to be passed between a sender and a receiver process. In this question you are to implement a synchronous *one-to-n* integer channel in which there is a single sender, and n receivers (for some constant n). The interface to your implementation should be the operations

```
op sync_send(int);
op sync_receive() returns int
```

A sender calling `sync_send(99)` is blocked until there are n processes ready to receive the value 99. A receiving process calling `sync_receive()` will be blocked at least until there are $n-1$ other receivers, as well as a sender. In the case when $n = 1$ the channel should behave like a normal synchronous channel.

Implement the above operations. Your solution should use MPD's message passing, and no other forms of synchronisation (such as semaphores, monitors etc.). *(10p)*

SOLUTION

```
process channel_server {
  op receive_waiting() returns int
  while (true) {
       in
         sync_send(i) st ?receive_waiting == n ->  # wait for n receivers
            for [r = 1 to n]{
               in receive_waiting() returns req -> # accept their call
                  req = i                      # and give them the value
               ni
            }
       []
         sync_receive() st ?receive_waiting < n ->
            forward receive_waiting() # at most n enter the waiting area
       ni
 }
}
```

Question 5. Write code which implements a *bounded* buffer for at most *M* integers. Your interface to the buffer should be provided by two procedures:

```
procedure put(int i){ ... }
procedure get() returns int i { ... }
```

You should provide an implementation of these procedures (and anything else you might need). Your buffer should allow arbitrary producers and consumers to use these operations. You do not have to write any code representing the producer or consumer processes.

Your solution should make use of MPD asynchronous message passing (i.e. the non-blocking *send* operation, together with *receive* or *in* statements) as the only form of process synchronisation. *(12p)*

SOLUTION

```
# Call-back style solution (general method):

op do_put(int, cap ())

op thechannel(int)              # the buffer itself

procedure put(int i){
  op ack_channel()              # create an acknowledgement channel
  send do_put(i, ack_channel)   # send the item and the channel
  receive ack_channel()         # wait for ack (i has been added to buf)
}

procedure get() returns int i {
  receive thechannel(i)
}

process server {
  while (true){
    in
       do_put(i, done) st ?thechannel < M # accept when there is space
       -> send thechannel(i); send done()
    ni
  }
}

-------------------------------------------------------------
Semaphore-style solution (works for this specific problem):
-------------------------------------------------------------
op empty_slot()

for [ i = 1 to M] { send empty_slot() }
```

```
# create M empty_slot tickets

procedure put(int i){
  receive empty_slot() # get an empty_slot ticket
  send buffer(i)       # put an item in the buffer
}

procedure get() returns int i {
  receive buffer(i)    # get an item
  send empty_slot()    # produce an empty_slot ticket
}
```