Lecture 7
Concurrent Programming

16th September 2003

Pass the baton

Pass the baton is worthwhile learning because it gives a general, efficient solution to implementing coarse-grain conditional atomic actions using semaphores. The transformation proceeds as follows:

1. introduce a binary semaphore e (e for "entry"), initialized to 1, controlling entry to the implementation of the atomic actions
2. associate one binary semaphore and one counter, both initialized to 0, with each different boolean condition that occurs as the guard of an *await* statement. I tend to name these sx and dx where x is mnemonic for the the condition they are associated with (s for "semaphore", d for "delayed". In the counter we will maintain the number of processes delayed waiting for that condition. Processes delay on the semaphore waiting for the condition and the semaphore is signalled when the condition becomes true.
3. Replace each CAA <await B S> with

```
P(e); if !B {dB = dB+1; V(e); P(sB); S; SIGNAL}
```

where SIGNAL is code that tests each of the boolean guards appearing in the program and whether there are any processes waiting for that condition. If it finds any true conditions being waited for it signals the associated semaphore of one of them, otherwise it performs V(e).

We'll take as our example the homework exercise requiring simulation of a general semaphore by binary semaphores. What do we mean by a simulation of a general semaphore? Clearly it must be able to represent states achieved by initializing to a value greater than one and by executing multiple VG operations (counting) and it must not allow a process to pass a PG operation if

```
init+nVG-nPG <=0
```

where init is the initial value, nVG is the number of completed general semaphore V operations and nPG is the number of completed general semaphore P operations. Another way of putting it: init+nVG-nPG >=0 is a global invariant.

So our coarse grain solution is :

```
nVG = 0; nPG=0;
VG () {
   <nVG = nVG+1>
}

PG () {
   <await ((init+nVG-nPG)>0) nPG = nPG+1)>
}
```

To use pass the baton, following the recipe, we first use an entry semaphore initialized to 1 to control entry to all of the atomic statements and we use a signalling semaphore and a counter corresponding to each different guard, all initialized to 0. In our example we have two guards, one being true and the other representing that the value of the general semaphore is > 0. So our simulation begins:

```
semaphore e=1, st=0, sz=0;
int dt=0, dz=0;
nVG = 0; nPG=0;

PG () {
   # <await ((init+nVG-nPG)>0) nPG = nPG+1)>
   P(e);
   if ((init+nVG-nPG)<=0) {dz=dz+1; V(e); P(sz)}
   nPG = nPG+1;
   SIGNAL
}

VG () {
   # <nVG = nVG+1>
   P(e);
   nVG = nVG+1;
   SIGNAL
}
```

Notice that we don't need st and dt. Now what about SIGNAL? Very simple:

```
if (dz>0 && ((init+nVG-nPG)>0)) {dz=dz-1; V(sz)}
else V(e)
```

A number of simplifying transformations can be made. Let's let g represent the count in the general semaphore. g ≡init+nVG-nPG

```
int g = init;
semaphore e=1, sz=0;
int dz=0;
PG () {
   # <await (g>0) g=g-1)>
   P(e);
   if (g<=0) {dz=dz+1; V(e); P(sz)}
   g = g-1;
   if (dz>0 && (g>0)) {dz=dz-1; V(sz)}
   else V(e)
}

VG () {
   # <g=g+1>
   P(e);
   g=g+1
   if (dz>0 && g>0) {dz=dz-1; V(sz)}
   else V(e)
}
```

Finally, notice that the test g>0 in the VG signal code is superfluous – g satisfies the semaphore invariant (why?) so after g=g+1, g is known positive. Is `if (dz>0 && (g>0)) {dz=dz-1; V(sz)}` in the PG code superfluous? That is, could the SIGNAL code for PG be just V(e)? We'd certainly hope so, because aquiring a resource doesn't produce anything for another process to consume. (But we would need to show that (dz>0 && (g>0)) is provably false at this point in the program to justify such an optimization.)

Drawbacks of pass the baton: the code for SIGNAL is replicated and optimized in each setting, which may lead to bugs as programs are modified. Notice how the PG code relies on the correctness of the test in the VG code prior to the call to V(sz).

Still, this is a very powerful technique, giving as it does a straightforward translation of coarse-grained code to fine-grained, but high level binary semaphore code.

Homework discussion

3.3 CS using swap instruction

```
swap(x,y) {
   <(x,y) = (y,x)>
}
```

If we don't worry about eventual entry, we can easily use swap to simulate test and set in the basic CS entry protocol.

```
lock = 0; -- corrected here but not in the F'02 hdcpy
Process i:
   int locali = 1;
   while (locali==1) swap(lock, locali);
   ...CS...
   lock = 0;
   ...NCS...
```

For ensuring eventual entry, swap is well-suited to atomically extending a linked list containing the waiting processes. Here's what I came up with. We'll build a linked list whose head is the last process to enter.

```
int fin[0..n] = ([n+1] 0);
int lastProcess = 0; /* id and fin value of previous process to enter */

process CS[i=1 to n] {
   int waitingFor;
   while (true) {
      -- CS entry
      waitingFor = i*2 + ((fin[i]+1) mod 2)
         /* this process is finished when fin[i] changes value */
      swap(waitingFor, lastProcess);
         /* learn the previous process id and fin value,
            store our processid and fin */
      waitingId = waitingFor/2;
      waitingFin = waitingFor mod 2;
      -- if (waitingId != i) {
      while (fin[waitingId] != waitingFin) skip;
      -- }
      CS
      fin[i] = (fin[i]+1) mod 2
      NCS
   }
}
```

Assignment for next class

Read the programming project and come prepared to discuss any questions you may have.

Assignment for Tuesday, Sept. 23

Try to implement signal-and-continue monitors using binary semaphores.