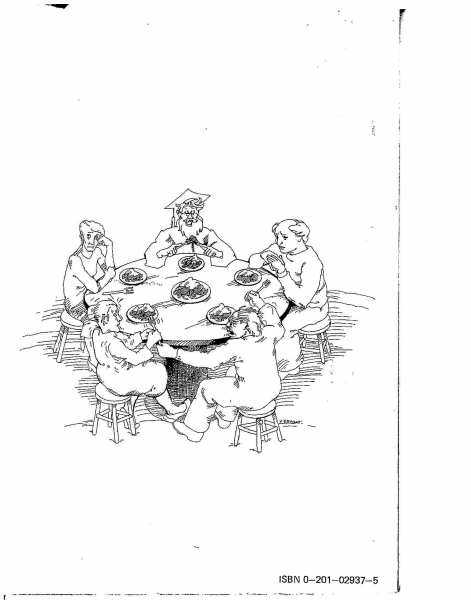
Lecture 7
Concurrent Programming

17th September 2002



Cover Illustration from
Structured Concurrent Programming with Operating Systems
Applications, R.C. Holt, E.D. Lazowska, G.S. Graham
M.A. Scott. Addison Wesley 1978

Semaphores - continued

Dining Philosophers was never one of my favorite problems but it does illustrate the potential for deadlock when using synchronization and also illustrates how to design locking so as to avoid deadlock. The key to preventing deadlock is to define a (partial) order on locks and always aquire them according to the order. If you aquire locks out of order, or you acquire locks whose ordering is not defined, you risk deadlock.

Notice that there may be more than one kind of lock in a system – semaphores, monitors, spin locks, even different processor priorities. They can all interact and lead to deadlocks, so you have to take all of them into account in designing your locking order.

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. 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. Also, is

the test leading to 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.

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 = 1;
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 arrive. We'll store the head pointer for the list in variable predecessor[0]. predecessor[j]==i should be interpreted as process j arrived after process i and will take its turn follwing process i. predecessor values will be manipulated with swap instructions to allow them to be performed atomically.

```
int predecessor[0..n] = 0;
int successor[1..n] = 0;

Process i:    -- (i in [1..n])
   predecessor[i]=i;
   swap(predecessor[0],predecessor[i]);
```

```
while(predecessor[i]!=0) skip;
... CS ...
find slot j such that predecessor[j]==i
predecessor[j] = 0
...NCS...
```

The above is not correct. The problem is in the exit code where by the time the assignment predecessor[0] = 0 is performed it may no longer be the case that predecessor[i]==0.

I have not yet managed to figure out a fix.

Assignment for next time

I will not be here for the next class. Please read the programming project and come prepared to discuss with your classmates any questions or unclarity you may have. Appoint someone to take notes and develop a list of questions and things you want me to discuss. Whoever is appointed should email it to me by Friday evening. Geir, you can send me your own list since you won't have the ability to interact with the class. I'll combine the lists and answer the questions.

Assignment for Tuesday, Sept. 24

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