Lecture 6 Concurrent Programming

12th September 2003

Finish up ticket and bakery algorithms from last time.

Remarks on busy-waiting protocols: in my experience, these are best used to synchronize multiple processors' access to data structures that implement higher-level synchronization primtitives such as P/V and monitors. Why? Because even on a multiprocessor it is likely you will have more threads than processors. You don't want application-level code to use spin-lock techniques because normally there will be some other task that the processor should be working on. Rather you want the blocked thread to enter a "wait" state.

Exception: if you know that your threads are the only ones running on the machine, you know that there are no more threads than processors, and you are going for ultimate parallel performance, then you most likely do want to use a spin-lock technique because you incur less overhead in acquiring locks that way.

So let's move on to some synchronization primitives that have more general use in shared-memory concurrent programs.

First stop: semaphores and the famous P and V operations. Why P and V? – they stand for the names of the operations in Dutch (see the historical notes at the end of Chapter 4). P and V can be expressed as two particular (conditional) atomic actions:

```
P (semaphore s) { -- wait
   <await (s>0) s=s-1>
}

V (semaphore s) { -- signal
   <s=s+1>
}
```

These are the only operations on variables with type semaphore. A semaphore can be created and initialized with any non-negative value. (Some implementations allow negative initial values.)

Semaphore global invariant for any semaphore, s, using my rules: $\{s \geq 0\}$.

A *binary semaphore* has a stronger global invariant: $\{0 \leq s \leq 1\}$. Notice how we don't declare a semaphore to be a binary semaphore – we just use it in a way that preserves this invariant.

Using a binary semaphore to solve the critical section problem is easy – the version using conditional atomic actions translates directly.

```
semaphore mutex = 1.
while (1) {
   {0≤mutex≤1}
   P(mutex); -- entry protocol
   {mutex==0}
   CS
   V(mutex); -- exit protocol
   {0≤mutex≤1} -- why not mutex==1 here?
}
```

This solution isn't *obviously* any better at guaranteeing eventual entry than was our version that used spin locks but with good implementations of P and V eventual entry is guaranteed.

Barriers

We skipped over barriers in chapter 3 but now with a better primitive in hand I want to at least touch on the notion.

A barrier is a point at which we require some set of threads to arrive before any can proceed. Examples:

- the host at the restaurant won't seat your party until you are all there
- implementing oc
- multiple threads compute values in one phase of their computation that are used by other threads in the next phase

Implementing barriers with semaphores. Although not clear in the above examples we require that a barrier be reusable: processes can visit the barrier repeatedly: we require that a process leave before other processes see it arriving at the next iteration.

We use a new notion: a *signalling semaphore.* Initialized to zero a signalling semaphore allows a process to wait for another process to "signal" that a desired condition has been achieved. For a two processor barriers, each arriving process signals its arrival with a V operation.

```
semaphore here1=0, here2=0;
co
   while 1 {
      beforebarrier1;
      V(here1); P(here2);
   }
//
   while 1 {
      beforebarrier2;
      V(here2); P(here1);
   }
oc
```

Another approach using a coordinator which generalizes easily to n processes:

```
semaphore here=0, go[1:2] = {0,0}
co
   while 1 {
      beforebarrier1;
      V(here); P(go[1]);
   }
//
   while 1 {
      beforebarrier2;
      V(here); P(go[2]);
   }
// -- coordinator
   while 1 {
      for [i=1,2] {
         P(here)
      };
      for [i=1,2] {
         V(go[i])
      }
   }
oc
```

We used one semaphore for here but a separate semaphore for each client process. What would happen if we tried to use a single semaphore that we repeatedly V'd to release all the clients?

Producer/Consumer

In lecture 3 we proved properties of a producer/consumer pair that synchronized using conditional atomic actions. In those programs the await statement in the producer was

<await p==c> which the global invariant told us meant “the buffer is empty”. The consumer used <await p==c+1> which the GI told us meant “the buffer’s contents are valid”. How can we mimic this solution with semaphores?

One key observation is that it is going to require more than one semaphore. Why? A semaphore can signal only one property becoming true. In this case we need to communicate two properties “the buffer is empty” and “the buffer’s contents are valid”. So let’s introduce two signalling semaphores, empty and full: after removing the buffer content the consumer performs V(empty). After filling the buffer the producer performs V(full). The consumer waits for the buffer to be non-empty with P(full); the producer waits for it to be empty with P(empty).

```
int buf, semaphore empty=1, full=0;
-- p and c are no longer shared variables
process Producer
   const int a[n] -- assume initialized
   int p=0;
   while (p<n)
      P(empty)
      buf = a[p]
      V(full)
      p = p+1

process Consumer
   int b[n]
   int c=0;
   while (c<n)
      P(full)
      b[c] = buf
      V(empty)
      c = c+1
```

Observe now that full and empty constitute a *split binary semaphore.* The sum of their values at any time is either 0 or 1. So out of our attempt to achieve signalling we have also achieved mutual exclusion.

General bounded buffer

The one-slot buffer works, but entails a great deal of *task switching* between the producer and the consumer. Since task switching is often an expensive operation (10s to 100s of times the cost of a procedure call) we would like to reduce the number of task switch operations. We’ll try using a buffer with more slots. *The bounded buffer is a very common pattern in concurrent programs and you should become familiar with its implementation using a number of different synchronization mechanisms.*

The key observation about the bounded buffer is that the producers and consumers wait essentially for the same things that they did in the 1-slot buffer case. For the consumer to proceed there must be something in the buffer. For the producer to proceed there must be empty space in the buffer. I'll depart from the book and name my semaphores fullSlots and emptySlots. It is immediately apparent that if the buffer has n slots total we should have an invariant that fullSlots+emptySlots==n (except when a thread is actively manipulating the buffer – that is, converting a full slot to an empty slot or vice versa).

If we use an array buf[0..n-1] as our buffer, we can use arithmetic modulo n to manipulate the insertion and removal points which we'll call rear and front, respectively.

```
int buf[0:n-1], semaphore emptySlots=n, fullSlots=0;
-- p and c are no longer shared variables
process Producer
   const int a[n] -- assume initialized
   int p=0;
   int rear=0;
   while (p<n)
      P(emptySlots)
      P(producerSem) -- for multiple producers
      buf[rear] = a[p]
      rear = (rear+1) mod n
      V(producerSem) -- for multiple producers
      V(fullSlots)
      p = p+1

process Consumer
   int b[n]
   int c=0;
   int front=0;
   while (c<n)
      P(full)
      P(consumerSem) -- for multiple consumers
      b[c] = buf[front]
      front = (front+1) mod n
      V(consumerSem) -- for multiple consumers
      V(emptySlots)
      c = c+1
```

For a single producer and single consumer we are done, but if we wanted to handle multiple producers (and consumers) front and rear would be shared variables and their use and updates would have to be protected. In the single slot buffer case we could have many producer processes relying on the split binary semaphore nature of empty and full to provide mutual exclusion. Now we are not so fortunate: if we want to support multiple producers and consumers we'll have to protect the uses of front and rear using mutex semaphores (which can be separate for the producers and the consumers – why?)

Reading for next time

Sections 4.3 through 4.5 and Chapter 4 Historical Notes, Chapter 5 intro and section 5.1.

Errata for book

Sec. 4.2.1 The second paragraph second sentence currently says:

Figure 3.2 presented a solution using lock variables in which variable lock is true when no process is in its critical section and lock is false otherwise...

This is exactly backwards. In figure 3.2 lock is true when a process *is* in its critical section. After correcting "no" to "a" in the existing sentence you also have to map true to 0 and false to 1 to make the translation to a semaphore. That’s ugly but I don’t see how to fix it any more simply.

Page 164, third line where it says Section 4.3 it should say 4.4.