Lecture 11
Concurrent Programming

8th October 2002

About the project

After working on the project a great deal more I've concluded that it is too much work to make the final step that allows the ER, EL, WR, WL lights to operate independently. It is quite difficult to come up with an approach that behaves reasonably in the first place. Implementing that approach requires a great deal of bookkeeping code that doesn't particularly ehance understanding of concurrent programming. So unless you really, really want to do it, I will relieve you of the obligation to make the final split of the signals: you may provide a controller that locks the ER and WR signals together and locks the EL and WL signals together.

Using monitors to implement synchronization schemes

So far we have talked about using monitors' mutual exclusion to protect the shared data, and condition variables for providing condition synchronization between processes that use the shared data. The discussion of implementing P and V with monitors has hinted at another style of using monitors. Recall the general approach to using P and V:

```
entry protocol using P/V;
manipulate shared data
exit protocol using P/V;
```

When we implemented semaphores using monitors we were really showing examples of how monitors could be used to protect shared data that was accessed outside the monitor: with what we have so far, you would use P/V implemented using monitors, just as you would use P/V if they were implemented as primitive operations. But we haven't looked at any problems solved using this style.

Therefore let us turn to the readers and writers example and examine its implementation using first semaphores and then monitors. In the readers and writers problem we want to allow multiple readers to access shared data simultaneously, but only allow a single writer at a time.

Recall during this discussion the issues of memory consistency on multiprocessors and in Java – data accessed outside the monitor will require careful attention to ensuring that different processes see a serially consistent view.

Our first example is Fig. 4.12 which uses binary semaphores and passing the baton to implement readers and writers.

At each point in the figure where SIGNAL appears we need to insert a specialization (optimization) of the following code which is generated by the following the passing the baton pattern.

```
if (nw == 0 && dr > 0) {
   dr = dr - 1; V(r); -- awaken a reader
} else if (nr == 0 && nw == 0 && dw > 0) {
   dw = dw - 1; V(w); -- awaken a writer
} else {
   V(e);
}
```

The different instances of SIGNAL optimize differently: in the first one, we know that nr>0, and nw==0 so the second case goes away and we're left with

```
if (dr > 0) {
   dr = dr - 1; V(r); -- awaken a reader
} else {
   V(e);
}
```

In the second case nw==0 and dr==0 (why?) so SIGNAL becomes

```
if (nr==0 && dw > 0) {
   dw = dw - 1; V(w); -- awaken a writer
} else {
   V(e);
}
```

In the third case (the first SIGNAL of the writer), nw>0 and nr==0 so SIGNAL here becomes

```
V(e);
```

And in the last case, nw==0 and nr==0 so SIGNAL becomes

```
if (dr > 0) {
   dr = dr - 1; V(r); -- awaken a reader
} else if (dw > 0) {
   dw = dw - 1; V(w); -- awaken a writer
} else {
   V(e);
}
```

The result of making all these optimizations appears in the book as Fig. 4.13. Notice how the solution gives *priority* to readers: if there are both readers and writers waiting the readers will go first. If readers arrive while there is a writer waiting, the readers will go first (potentially forever.)

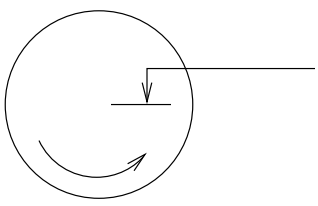
Let's turn now to using monitors to implement a solution to the readers and writers problem. We could, of course, use a monitor implementation of P/V and substitute it directly into the previous solution. That would require 3 monitors, one for each semaphore). Can we go more directly to a monitor-based solution and get something better?

Refer to Fig. 5.5 and compare with Fig. 4.12. Notice how the monitor lock plays the role of the e semaphore, oktoread is similar to the r semaphore and oktowrite is similar to the w semaphore. Note also the differences: with semaphores the signaller only signals in the case that there is a waiter because the signal is remembered. In the monitor, the signaller notifies every time and it is up to the process receiving the signal to determine whether or not the condition it is waiting for truly holds.

I'll point out again what seems to me to be the relative fragility of the semaphore solution: the basic safety invariant requires considerable attention across the span of the program. The monitor approach is somewhat less fragile, it seems to me, because each process locally checks for the necessary condition before proceeding to access the shared data. However, with this style, involving surrounding shared data access with calls to the right monitor procedures at the right time is, I believe, less robust than including the shared data directly in the monitor.

Another example: three approaches to disk-arm scheduling.

Disk fundamentals:



Head moves across a concentric *tracks*. Time to access any particular data is combination of *seek time*, to move the head to the right track, *rotational latency*, for data to come under the head, and *transfer time*.

Moving from track to track is a non-linear, but increasing, function of the distance moved. A complete traversal might take only twice as long as moving one track.

Common scheduling policy for disk arms is to service requests in increasing track number order until the highest track is reached, then start over again at track 0. This is commonly known as the (circular) elevator algorithm.

With multiple processes using the disk how can we schedule the arm optimally? Why doesn't simple mutual exclusion work? Sub-optimal because the disk cannot "see" pending requests and schedule accordingly. We have to split read into *request* and *do*. *Request* can be called at any time and provides info about what is to be done. *Do* takes different forms in different approaches:

- client calls *request*; request returns when it is this client's turn; client then calls *do* and *release*. Problem: we rely on the client to perform the operations in the right order. Also, *do* is typically a complex set of commands that we don't really want to leave to the client to get right. The protocol can, however, be encapsulated in a disk object that clients use to access the disk.
- disk driver process (coordinator pattern/pipeline pattern): adding a process to serve as the disk driver simplifies the code considerably at the cost of additional process switches. The monitor used here provides operations for clients to make requests and the driver process to pick up the next request and return its results. (As suggested in the book, the right primitive for the driver might be *returnResultsAndGetNextRequest*. Why?) I like this approach a lot because using a separate process allows the required order of disk operations to show up naturally in the structure of the driver process instead of being distributed between the client processes (and therefore having to use synchronization to achieve the correct order.)



- The third approach discussed in the book (section 5.3.3) using nested monitors I do not consider viable because it requires so-called *open calling* between monitors which is implemented by no systems that I am aware of. Without open calling it reduces to essentially the first approach, using a disk object to encapsulate the conventions.