Lecture 15
Concurrent Programming

16th October 2003

Example: Disk Scheduler and Driver revisited

When we talked about the disk driver and scheduler before we suggested that a structure in which User processes placed their requests in an interface monitor whence they were pulled by a disk driver process was simple and elegant.

When structuring a solution with explicit communication we could, of course, implement the monitor using a process – as we discussed last time. But we can do better by making the disk driver process *self-scheduling.* The driver process and the scheduler can be merged into one: refer to figure 7.9.

The basic structure of a single iteration of the outer loop:

```
chan requests
...
while no requests on hand or additional requests pending{
   receive requests
}
select best request
move to requested cylinder
perform disk operation
send result to client
```

When a disk operation completes, wouldn't it be better to issue the next disk operation before going into a loop of unknown duration? Maybe. We'd want to assess how long that loop really could be in practice and what the practical result of that delay would be on disk throughput. We'd also want to take into account that having more requests on hand gives us more opportunity to select the best request.

It seems to me that if we're going to change the structure we should try to acquire more operations to choose among while the disk operation is happening – and that means either changing to a separate driver/scheduler architecture or using an asynchronous I/O operation:

```
chan requests, diskIOcomplete
...
while 1 {
   if !empty(diskIOcomplete) {
      recv diskIOcomplete
      send result to client
   }
   if !diskbusy {
      select best request
      start disk operation
   }
   if (disk busy && !empty(requests)) || !(diskbusy) {
      recv requests -- receive one request; could program maximum number
   }
}
```

The above program gives priority to handling disk completion, yet acquires client requests while the disk is busy. How about with select()?

```
chan requests, diskIOcomplete
...
while 1 {
   completedEvent = select([diskIOcomplete, requests])
   if completedEvent is a request {
      enqueue it
   } else
      return result to client
   }
   if !diskbusy && !workqueue empty {
      start diskIO
   }
}
```

The code is simpler but we've given up control of which messages will be received first. The book talks later about adding priorities to selective communication in order to allow control in these situations. So another dimension of variability: when selective communication is provided, does it permit the programmer to specify which communications have highest priority.

A note about project2: the specification for project 2 does not allow for priorities in selective communication. Indeed, if a SelectionList is used repeatedly, starvation may ensue. When you get stage 3 working, think about what it would take to add priorities.

Example: File servers – stateless and stateful client interactions

The example in section 7.3.3 of the book points out two rather different approaches to implementing client/server interactions. The code of figure 7.10 is based on what is called the *stateful server paradigm.* In stateful server style, when a client first contacts a server it receives from the server a token that it uses to identify itself in further interactions. The server retains data representing the state of the interaction that it uses to affect its behavior for that client.

An example: a client opens a file and performs repeated readNext operations to sequentially scan the file. The server retains the read pointer associated with the clients operation. Windows NT file servers are stateful. (The state example in 7.10 is really only that the file is open and that a particular client is using a particular server process). Examples of state: read and write pointers, locks, client cache contents, ...

When Sun designed the Sun Network File Service protocol they wanted to make the server stateless – generally a good starting point in a distributed system. Sun NFS doesn't have an open() operation. The server never knows from one operation to the next what the client is doing – the client always has to be explicit about what it wants in each request. There is no opportunity for client and server to get confused about what each other is doing. This comes at a cost though: each client operation must pass data to completely describe the desired operation. In a stateful protocol if client state and server state don't correspond, it must be detected and a recovery protocol must be run. Thus stateless protocols may be easier to implement but also may have higher run-time costs: more data is transferred with each operation.

HTTP is also essentially a stateless protocol, though through the use of *cookies* it has been extended with statefulness. One useful concept is the notion of soft state: the server is allowed to forget its state associated with a client at any time. Soft-state protocols provide operations allowing the client to recreate the necessary server state.

Example: Exchanging messages

As we saw last time with the Fibonacci dataflow network networks of communicating processes can have rich structure. In this example we look at 3 different structures solving the same problem. The programs are depicted in Figures 7.11 through 7.13 and their general structures are pictured in Figure 7.14.

The example problem: suppose each of n processes hold a private value v. Our goal is for all n processes to learn the minimum and maximum values of v in the system. A great many different communication structures can be built to solve this problem. Some of the most basic and obvious are described in the book:

- central server – one process collects all the values from the others and distributes the answer. The solution is fast and requires only a small number of messages. It is susceptible to failure of the central server. In case of failure some participants may not get answers, but all who do will get the same answers.

- fully distributed – each process learns the value of v in all the other processes and computes the minimum and maximum for itself. In case of failure different processes may compute different answers. Uses lots of communication $O(n^2)$.
- ring – processes pass values to their neighbors who, as values pass around the ring, compute successively better approximations until the last participant is reached at which point the final answer is known. Failure of any process may lead to no process obtaining an answer. Processes receive good answers if they receive any answer.

Study of these kinds of mechanisms and many more is a large part of 464/564