Lecture 21
Concurrent Programming

13th November 2003

Leaders and followers pattern

Problem: Clients periodically present service requests which requires a significant amount of work that along the way involves waiting for something else to happen. Example - on-line transaction processing.

Characteristics: application should not block on any one source of requests

First, some examples of other patterns for implementing such servers.

Reactor (Dispatcher or Notifier)

We saw this pattern before in Lecture 12 in the form of what I then called an event loop framework. For each different kind of event (request) to be handled an event handler is registered with the framework. The framework demultiplexes the arriving events delivering each one to the appropriate handler.

As previously noted, reactor is fast and uses few resources, but can lead to rather complicated programs if the individual requests involve interactions with external entities that can cause delay.

Illustrate: single thread with registration mechanism and dispatching loop.

Forking server

An approach you may have seen before:

```
serversocket = socket(...)
rc = bind(serversocket)
rc = listen(serversocket)
while true {
   newsocket = accept(serversocket)
```

```
    if (!fork()) {
        close(serversocket);
        serviceoneclient(newsocket);
    }
}
```

This is the approach used in many Unix servers – web, ftp, telnet, ssh, etc. The problem is that the fork() operation is quite costly both in time and space. It takes a lot of work to construct a new address space, and the new process occupies a significant amount of memory (even if the code and much of the data is implemented using shared memory techniques).

The forking server approach was the only one used in the Apache web server until quite recently.

Threaded server

Using threads instead of processes can reduce the cost. In many threading systems, the memory cost is reduced to a separate, pre-allocated stack per thread, and the time cost is lessened because less address space manipulation is needed to create the thread. What if threads were really cheap? Notice that these threads have very little interaction with one another: what measures of thread performance are most important in this setting?

Examples: recent Apache web servers, the AOLServer web server, many services written in Java.

Leaders and followers

Instead of a thread per client, the LF pattern allocates a fixed number of threads, just a sufficient number to keep the server busy. The threads take turns demultiplexing client requests. At any time, one thread is designated as the leader and only the leader thread is allowed to read a client request. Since it doesn't know which client will make the request it uses a selecting operator to wait for any existing client or a new client. When a request arrives, the leader removes that client from the selectable client list, lets one of the follower threads become the new leader and goes off to process the request it received.

```
/* initialization */
serversocket = socket(...)
rc = bind(serversocket)
rc = listen(serversocket)
selectset = emptyset;
setadd(selectset, serversocket);
... create n server threads ...
```

```
/* each server thread */
while true {
   if iAmLeader() {
      rc = select(selectset)
      readyfd = findreadyfd()
      if readyfd==serversocket {
         newfd = accept(serversocket)
         selectset = union(selectset, {newfd})
      } else {
         selectset = setremove(selectset,readyfd) /* why? */
         leader = null;
         retireAsLeader();
         readAndServiceOneRequest(newfd);
      }
   else {
      waitToBecomeLeader();
   }
}

synchronized boolean iAmLeader () {
   return leader==myId;
}

synchronized waitToBecomeLeader() {
   while (leader!= null) {wait()}
   leader = myId;
}

synchronized retireAsLeader() {
   leader = null;
   notify();
}
```

How many threads should there be in a leaders/followers thread pool?

Obvious: fewer than one per connected client – otherwise we would just use the threaded server pattern which is simpler to program. Can we come up with a more precise characterization of the desired number of threads? Consider the role of this server: it serves client requests by making use of other resources. We want to use threads because these other resources impose delays, but there are many of them so we wish to overlap those delays for different clients.

Gaining more control over the set of waiting requests

In the example code above we let the OS manage the queue of waiting requests: they are either connect requests or request messages from clients waiting to be read from file descriptors. What if we needed more control over which of the requests would be processed first? Idea: impose an intermediary.

Bag of Tasks

Leaders and followers is similar to the bag-of-tasks pattern discussed in the textbook (p. 132). Each task in the bag-of-tasks corresponds to what we have called a client request. The multiple threads of the leaders and followers pattern are the worker processes of bag-of-tasks. Note that LF explicitly addresses something that is left implicit in the BoT paradigm: which process/thread is to be the next to remove an element from the set of waiting tasks/requests.

Example1: Akamai

Akamai is a company that sells the service of web caching. Akamai has thousands of servers worldwide on thousands of network. The goal is to have servers that are close in network distance to all internet users. Akamai uses DNS manipulation to direct clients to nearby servers. Once a client request reaches an Akamai server, satisfying the request may involve interaction with Akamai's customer's servers. (Akamai's customer in this case is the company that owns the web site; the client might be that company's customer).

An Akamai-like server might have thousands of connected clients simultaneously, only some of which have an outstanding request. The ones with outstanding requests will have interactions with the customer servers which need to be managed over several interactions – a thread is a natural way to manage the state evolution of these. But we can't afford to tie up a thread for each connected client.

Example 2: NFS server

Before the Solaris OS had a multithreaded kernel, and still today in Linux, the leaders and followers pattern was used to implement the NFS server. All of the work of an NFS server can happen in the kernel – request packets arrive first in the kernel, file reads and writes can only get to the disk by being handled by the kernel. Thus, although NFS can be implemented in a user-level process, doing so adds overhead that could be avoided by a kernel implementation.

The problem is that the OS kernel in these systems is an asynchronous system, and as we've seen, handling complex state evolution in an asynchronous system over long periods adds a lot of complexity. What the NFS implementors did was in essence create a fixed-size pool of user processes that do nothing but make a single system call

that waits in the kernel for NFS requests. As each request arrives it is assigned to one of these processes which goes through the sequence of steps necessary to satisfy that request. Once the request is satisfied the process is ready to accept the next request.

In this system the NFS server can be working on a fixed maximum number of requests at a time.

Once the kernel evolved to include kernel-level threads subterfuge of user-level processes was not needed any more, and threads could be created as needed to handle the load. If I were designing the system, I would certainly still use a thread pool so that I didn't have to create a thread for each request. But with kernel level threads available I could make the size of the thread pool change dynamically.

Example 3: CORBA ORB

Leaders and followers might be one of the options offered by a CORBA ORB for implementing servant objects. (CORBA ORBs typically let the system implementor choose between several of the patterns discussed above – sometimes on an object-by-object basis.