CS 33

Final Words
Thread Scheduling

- The OS multiplexes threads on the available processors/cores
  - share the processors equally
    » time slicing: each thread gets a fixed amount of time before it's forced to yield the processor to another thread (if there is one)
  - some threads are more important than others
    » priorities: higher-priority threads get the processor in preference to lower-priority threads
A Scheduling Issue

• You and four friends each contribute $1000 towards a server
  – you, rightfully, feel you own 20% of it
• Your friends are into threads, you’re not
  – they run 5-threaded programs
  – you run a 1-threaded program
• The scheduler treats all threads equally
• Their programs each get 5/21 of the processor
• Your programs get 1/21 of the processor
  – (you should have paid more attention to the fractal threads lab)
Lottery Scheduling

- 25 lottery tickets are distributed equally to you and your four friends
  - you give 5 tickets to your one thread
  - they give one ticket each to their threads
- A lottery is held for every scheduling decision
  - your thread is 5 times more likely to win than the others
To measure the usage of a processor, let’s assume the existence of a meter.
Assuming all threads are equal, all started at the same time, and all run forever, the intent is to share the processor equitably. Note that as the time between clock ticks approaches zero, each thread gets $1/n$ of total processor time, where $n$ is the number of threads.
Issue

• Some threads may be more important than others
Let’s now assume that meters can be “fixed” so that they run more slowly than they should. Thus a thread with a fixed meter gets charged for less processor time than it has actually used.
Details …

- Each thread pays a bribe
  - the greater the bribe, the slower the meter runs

- to simplify bribing, you buy “tickets”
  » one ticket is required to get a fair meter
  » two tickets get a meter running at half speed
  » three tickets get a meter running at 1/3 speed
  » etc.
New Algorithm

• Each thread has a *(possibly crooked)* meter, which runs only when the thread is running on the processor

• At every clock tick
  – give processor to thread that’s had the least processor time as shown on its meter
  – in case of tie, thread with lowest ID wins
The slide illustrates the execution of three threads using stride scheduling. Thread 1 (labeled with a triangle) has paid a bribe of three tickets. Thread 2 (labeled with a circle) has paid a bribe of two tickets, and thread three (labeled with a square) had paid only one ticket. The thicker lines indicate when a thread is running. Their slopes are proportional to the meter rates (and inversely proportional to the bribe). Note that meter values on the y axis are twice as far apart as ticks on the x axis.

In this example, a total bribe of six tickets has been paid. After six clock ticks, each thread’s meter has been increased by 1.

In general, if the clock ticks once per second and the total bribe is \( B \), then after \( B \) seconds, each thread’s meter has increased by exactly 1. To see this, assume that each thread \( t_i \) starts with a meter reading of the reciprocal of its bribe \( b_i \). To make this easier, let’s assume that each thread has paid a different bribe. Suppose thread \( t_1 \) paid the largest bribe, \( b_1 \). After some period of time its meter will have increased by 1, requiring \( b_1 \) seconds of actual execution. Since it’s the thread that paid the largest bribe, its meter will be increased by 1 before that of any other thread. It of course won’t run again until its meter has the lowest value. Thread \( t_2 \), which paid the second largest bribe, will be the second thread to have its meter increased by 1, requiring \( b_2 \) seconds of actual execution. It also won’t run again until its meter has the lowest value. Similar arguments can be made for the remaining threads, through \( t_n \). Once \( t_n \)’s meter has been increased by 1, \( t_1 \) again has the lowest meter value and the cycle starts again. The total amount of time required to get to this point is \( b_1 + b_2 + \ldots + b_n \), i.e., the total bribe.
File systems are certainly important parts of general-purpose computer systems. They are responsible for the storage of data (organized as files) and for providing a means for applications to store, access, and modify data. Local file systems handle these chores on individual computers; distributed file systems handle these chores on collections of computers. In the typical design, distributed file systems provide a means for getting at the facilities of local file systems.
What we’re accustomed to with local file systems is that, in the event of a crash, everything goes down. This is simple to deal with.
In a distributed system, if the server crashes, there is no inherent reason for clients to crash as well. Assuming there was no damage done to the on-disk file system, client processes might experience a delay while the server is down, but should be able to continue execution once the server comes back up, as if nothing had happened. The crash of a client computer is bad news for the processes running on that computer, but should have no adverse effect on the server or on other client computers. We'd like the effect to be as if the client processes on the crashed computer had suddenly closed all their files and terminated.
Note that the NFS server does not maintain open-file state – for that reason it’s said to be *stateless*. 
The thread opens a file, reads the first 100 bytes, then writes 100 bytes at the beginning of the file, replacing the previous data there. The kernel data structures representing the open file are just as they would be if the file were local. The only difference from the client operating system’s point of view is that the actual operations on the file are handled by the NFSv2 client code rather than by a local file system such as FFS.

The NFSv2 server has its own directory hierarchy. It exports a number of subtrees of this hierarchy to its clients. The client’s operating system has mounted one of them on the local directory `/home/twd`. It did this by first placing a remote procedure call, using the mount protocol, to the server, obtaining a file handle for the root of this subtree. Any attempt to access the directory `/home/twd` will be interpreted as an attempt to access the remote directory represented by the file handle.

Thus to follow the path `/home/twd/dir/fileX`, our thread, executing in the client operating system, follows the path as far as `/home/twd` and discovers that a remote file system is mounted there. It places a remote procedure call to the NFSv2 server’s `lookup` routine, passing it the file handle for the root as well as the pathname `/dir/fileX`. The server returns a file handle for `fileX`.

Just as it would for local file, the client operating system records the fact that `fileX` is open. However, rather than having an inode or equivalent to represent the file, it uses the file handle together with some sort of communication handle representing the remote server.

When the thread performs the read, it, within the client operating system, places a remote procedure call passing the file handle, the offset within the file (0), and the length (100) to the server’s read routine. The server then returns the 100 bytes from that location.

The `lseek` call sets the local file offset to 0, but causes no request to be sent to the server.
server. When the thread calls write, it places a remote procedure call containing the file handle, the current offset (now 0), the data, and the length to the server’s write routine. The server returns an indication of successful completion of the operation.
Note that the fact that a file is open is known only on the client. The server is stateless (with respect to client access to file systems) and doesn’t keep track of who has what open and how.
However ...

```c
int fd = creat("/home/twd/dir/tempfile", 0600);
char buf[1024];
unlink("/home/twd/dir/tempfile");
...
write(fd, buf, 1024);
...
lseek(fd, 0, SEEK_SET);
read(fd, buf, 1024);
close(fd);
```

The file `/home/twd/dir/tempfile` is created, then its directory entry is immediately removed. If this were done on a local file system, the file itself would continue to exist, even though there no longer is a directory entry referring to it — that the file is open causes it to continue to exist. This code sequence is rather common in Unix programs, it is a standard technique for creating a temporary file — one that is guaranteed to disappear once the process terminates: as soon as the file is closed, there is no longer a reference to it and the operating system deletes it.

Since NFSv2 servers are stateless, they do not keep track of whether files are open. Thus the unlink request in the code above would remove the last link to the file and leave it with no other references — the file would then be deleted, much to the surprise of the client.

NFSv2 relies on the client to keep track of such state information and to modify its requests to the server accordingly. In our example, the client would realize that sending the server an unlink request for an open file isn’t in its best interests. So, rather than sending the server an unlink request on an open file, it sends it a rename request, changing the file’s name to a special name that does not show up in normal directory searches (in Unix, this is accomplished by having the name start with “.”). Thus the file no longer appears to exist by its original name, yet the client’s file handle still refers to it. When the client process closes the file, then an unlink request is sent to the server, removing the file’s special name from the directory and thus finally deleting the file.

This solution clearly is not perfect: it works only if the file is open on the same machine as the one doing the unlink. In practice this is the only case that matters, so it works well enough. But what happens if the client, after renaming a file, crashes before the file is closed? The result is that the renamed file is not deleted. It remains on the server. The server must periodically check for such orphaned files and delete them.
Locks on Files

• Your point of view
  – you take an exclusive lock on a file
  – you expect to have exclusive access to the file until you unlock it

• Server’s point of view
  – client A locks file
  – client A appears to have crashed
  – client B wants to lock file
  – server lets client B have the lock
How Does Server Know Client Crashed?

- **Timeout**
  - after period of unresponsiveness, server assumes client must have crashed

- **Reboot**
  - client crashes
  - reboots and notifies server that it is rebooting (and thus must have crashed)

- **Neither is perfect**
  - reboot used in CS department NFS
  - timeout used in many modern systems
You’ll Soon Finish CS 33 ...

- You might
  - celebrate

- take another systems course
  - 32
  - 138
  - 166
  - 167

- become a 33 TA
Systems Courses Next Semester

- CS 32 (Intro to Software Engineering)
  - you've mastered low-level systems programming
  - now do things at a higher level
  - learn software-engineering techniques using Java, XML, etc.

- CS 138 (Distributed Systems)
  - you now know how things work on one computer
  - what if you've got lots of computers?
  - some may have crashed, others may have been taken over by your worst (and smartest) enemy

- CS 166 (Computer Systems Security)
  - liked buffer?
  - you'll really like 166

- CS 167/169 (Operating Systems)
  - still mystified about what the OS does?
  - write your own!
The End

Well, not quite …
Database is due on 12/14.
The TAs and I will hold hours all next week.

Happy coding and happy holidays!