**Review of Multilevel Feedback Queues (MLFQ)**

Using past behavior to predict the future. Processes that have been I/O bound in the past will probably be I/O bound in the future. We can prioritize processes that have been I/O bound in the past because they take the shortest amount of time. This is how we approximate shortest job first. We give the lower priority queues more time to use the CPU because those jobs are the longest. Jobs start in the highest priority queue and if it doesn’t finish in the CPU time its given, we move it down a queue. If the job does finish before the time runs out, we move it up a priority.

**Review of Lottery Scheduling**

Every job has a ticket and we pick them at random. The immediate next job is random, but we can give different amounts of tickets to more important jobs making them have more time. All jobs are going to get done because of statistics but we give the shortest jobs the most tickets so they are most likely to be called first. Remember we want to emulate shortest job first.

**Threads**

**What are threads?**

A thread is within a processes. It is a single sequential execution stream within a process. A single process can have many threads running at once but a thread is bound to just one process. Threads have their own registers and stack but use the same code, data, and files as all the other threads in the process.

**How do we manage threads?**

We have some options with how we manage threads. The first way is by using **kernel threads.**  These are threads that the operating system knows about and is managing them. As the name implies, the kernel is doing all the work and it is treating the threads in a similar way that it treats processes. It can use the same scheduling algorithms. We still have to do context switches between them but these switches are much faster than the ones processes have to do.

The second way to manage threads is **user-level threads**. The OS is unaware of these because the user level program is the one managing them. The OS just sees a process but knows nothing about the threads on the inside. The programmer uses a thread library to manage the threads. We don’t have to do context switches with these meaning user level threads are faster than kernel level threads. We also have a lot more control over how we schedule our threads so we can personalize it.

The issue with user level threads is if the CPU only has the one process executing, all of it will run on the same core. There is no parallelism available since the OS doesn’t know about the different processes.

Best case is to use both.

**Threading Models**

We have different models on how to manage threads. We can have one kernel thread for every user thread, we can have multiple user threads on one kernel thread, we can have many kernel threads mapped to many user threads, and we can have a mixture of one to one and many to many.