**Review of Processes**

A process is the unit of execution. Processes are represented as PCBs in the OS which contains all the information we need about it. Processes are always in one of these states: New, Ready, Waiting, Running, or Terminated.

A Uniprocessor can only only run one process at a time. When the currently running process is changed it is called a context switch.

Processes communicate with each other by using message passing or shared memory.

**Scheduling Processes**

**Long Term Scheduling**

Long term scheduling is the **big picture.** We are answering questions like how many process do we want to run at a time. If we do too many processes at once, the system will bog down with vital memory issues. If we don’t have enough we are wasting efficiency.

**Short Term Scheduling**

Short term scheduling is more our concern. This is when we are deciding “**What to do right now**?” How do we pick which process should be moved **from ready queue to run next**. This job is done by the **Short Term Scheduler.**

There cases when we want the Short Term Scheduler to run, these cases are when a process goes from **running to waiting**, an **interrupt occurs**, or when a process is **created or terminated**.

We have two options when these cases are met. We can use a **non-preemptive system** which is the scheduler is **waiting for one of those events** to happen. The other option is a **Preemptive system** is when the scheduler can interrupt a running process to make a change.

**How do we compare scheduling algorithms**

We have a couple measurements we can use: **CPU utilization** (How much of the CPU’s time is spent being busy), **Throughput** (the number of processes that are being finished), **Turnaround time** (how long it takes for a process to go from initialization to termination), **waiting time** (the amount of time the process is in the ready queue), **response time** (the time between a process being ready to run and its next I/O request).

We pick the measurement we want based on what we are expecting the system to do. If we are doing mostly I/O bound jobs, CPU utilization would not be a good thing to focus on, response time would be.

We want our scheduler to optimize everything but thats not possible. We’re going to pick our scheduling algorithm based on what we are using the system for.

We are going to make some assumptions when measuring scheduling algorithms. These are one process per user, one thread per process, and processes are independent.

**Scheduling Policies**

**FCFS:** first come first serve. The scheduler executes jobs in the order that it gets them. The process doesn’t give up the CPU except for when it is waiting on I/O. Once it gets its I/O request back, it goes to the front of the line.

The advantage of this is it is simple and easy to implement.

The disadvantage of this is a volatile average wait time as long jobs might be holding up shorter jobs. This also leads to poor overlap performance between I/O and CPU. The I/O devices might end up sitting idle for a long time causing back up.

**Round Robin:** The basis is time sharing. We use a timer and a preemptive policy where whenever a timer goes off we issue an interrupt that shoves the running process to the back of the ready queue.

The disadvantage is with picking the time slice. If you pick a time slice that’s too large, it turns into FCFS where jobs are just finishing. If you pick a time thats too small, the throughput suffers because we are having to do too many context switches.

This is bad for average wait time since they are always being sent to the end of the line.

The advantage of round robin is that it is fair. No jobs are going to be forgotten, all of them are going to execute in a fair amount of time.

**Shortest Job First (SJF):** This is scheduling the job that has the least amount of work to do before its done first. We measure this by how long it would take until the process either terminates or has to wait on I/O.

This is provably optimal with respect to wait time. No matter what the next process has to wait the minimal amount of time because the process being currently done takes the shortest amount of time.

If we want to have this as a preemptive process, we can do **Shortest Remaining Time Left (SRTL).** In this case, we work on a job until there is another job that has less time left on it then we switch to the, now, shorter job.

This prioritizes I/O bound jobs over CPU bound jobs because time between I/O waits is typically shorter than completing a CPU job.

The disadvantages are that it is really hard to predict how much time a job has left. We don’t really know how long a job will take so we could be wrong sometimes. Another disadvantage is that we can starve long running jobs. If we have a big enough job, it may never actually get executed because there will always be shorter ones.

**Multilevel Feedback Queues (MLFQ):** This is what is most comely used now. It uses past behavior to predict the future and make job assignments. If a job has been I/O bound in the past, it is likely that it will be I/O bound in the future. Using this we can better predict which jobs have the least amount of CPU time. This is an adaptive process because it uses past behavior which changes how it acts.

We are going to make multiple queues with processes in each of them. We are then going to give each of these queues its own priority level. We run round robin on the highest priority queue. Once one queue is finished, it moves to the next. I/O jobs are typically going to occupy the higher priorities while CPU bound jobs are going to be lower. Because of this, we actually change the time slice we are using on lower priority queues so that they get more time.

Jobs all start in the highest level queue, if the time slice expires and it is not done, it moves down one queue. If the time slice does not expire, meaning it either finished or got to an I/O wait, it is moved up in priority. This is how we get I/O jobs a higher priority and CPU ones to a lower one. But since lower priority queues have bigger time slices, when they are run, more can be done.