1 Multithreading

- The point of using threads was to execute multiple “threads of control” in parallel – but how do they actually run in parallel?
- Since most machines are uniprocessor, yet still support multithreading, the operating system has to somehow schedule threads (and processes, by extension) so that each gets some time on the CPU
- This is the problem of job scheduling
- We also must factor in the usage of I/O devices when scheduling processes

2 Metrics for scheduling

- Throughput – Average number of processes completed per unit time
- CPU Utilization – Average (percent) usage of the CPU
- Wait time – Time spent by processes waiting to use the CPU (in the ready queue)
- Turnaround time – Total time from the time a process is started to the time it is completed
- Response time – Time for the process to start responding (get some CPU time)
- We would like, ideally, to maximize throughput and cpu utilization while minimizing wait time, turnaround time, and response time
- The performance of various scheduling algorithms also depends on how we treat processes that deal with I/O

3 FIFO/FCFS (First-Come First-Serve)

- Process jobs (processes) as they arrive in the system, and finish them completely before moving on to the next job
- Essentially a batch system – DOS batch files provided this capability to a certain extent
- Seems ok if all jobs are the same, we don’t care when they finish, and the order in which they enter is arbitrary, but otherwise?
- Long wait time – since we don’t take into account short jobs before long ones, we can have all jobs (some of which may be short) get stuck behind long ones, increasing the overall average wait time
4 Shortest Job First (SJF)

- Place jobs in a priority queue as they arrive, with total job run-time as the key – choose minimum, and finish that job completely before moving on to the next job
- Requires knowledge of the future (how do we know how long a particular job will run?)
- Works better than FCFS because we get short jobs out of the way quickly so that the overall average waiting time is decreased
- Is it fair to give short jobs priority over long jobs?

5 Time-slicing and Preemption

- Instead of allowing one process to stay on the processor forever, instead allow a process only a certain slice of time at which point the scheduler may decide to run a different process instead
- Each unit of time given to a process is a time-slice or quantum
- If a job finishes before its time-slice is over, simply allow the scheduler to run another job immediately (rather than waiting until the end of the now-unused time-slice)
- We consider this behavior preemption – the scheduler preempts (interrupts) processes while they are running by taking the CPU away from one process and giving it to another

6 Round Robin

- Place jobs as they arrive in a standard FIFO queue – remove the first job from the queue, run it for at most one time slice
- If it doesn’t finish within the time slice, place it at the end of the queue, and choose the next item from the queue
- Provides some sort of fairness (since all jobs are treated equally), but takes a long time to finish everything because each job gets very little time on the CPU (so the average turnaround time is high)

7 Shortest Remaining Time First (SRTF)

- Similar to shortest job first, except we use “time to completion” (remaining time) instead of “total job length” and we use time-slicing
- Place jobs in a priority queue as they arrive, with remaining time as the key – choose minimum, and run that job for at most one time-slice
- If the job finishes, remove it from the queue, otherwise, update its key value (decrease-key) and choose the minimum from the queue once again
- SRTF also requires knowledge of future information – it turns out that SRTF is optimal (it minimizes average wait time over all processes)
8 Shortest Elapsed Time (SET)

- Since SRTF and SJF require knowledge of the future, we can’t actually use them – instead, we can approximate SRTF by using “elapsed time” as the key for processes in the priority queue, where elapsed time is the amount of time the process has been running.
- Shortest Elapsed Time approximates SRTF because it is fairly safe to assume that a job that has been running for a long time will continue to run for a long time and that a job that has been running for little time will complete soon.

9 Fairness

- The above scheduling algorithms somehow assume that all users have an interest in decreasing the average wait time – in reality, users only care to decrease their own jobs’ wait times.
- How can a scheduling algorithm be fair?
- Round robin was fair in a sense (it gave all processes equal time), but was inefficient – is it possible to provide decent performance and be somewhat fair?
- Need a way of distinguishing between processes to somehow decide which processes deserve more CPU time or more prompt scheduling.

10 I/O Bound vs. CPU Bound Processes

- Processes that do I/O (input/output) with I/O devices (such as the disk, network, terminal, ...) often spend a large portion of their time waiting on the devices, meaning they only need a little time on the CPU to request the I/O operation and then can sit and wait for the I/O device to respond (which does not require the CPU).
- Processes that do CPU intensive tasks (computation) often do not use I/O devices significantly in proportion to how much they use the CPU – these tasks need to spend time using CPU to do their work.

11 Priority Scheduling

- There are various ways of handling priorities for processes – we’ll look at one way.
- Have multiple queues – a ready queue (of processes ready to run), a wait queue (of processes waiting on I/O or blocked on locks), and a running queue (of processes currently running on the CPU).
- Each process has a priority, and we choose based on priority which process to run.
- When processes make I/O requests, causing them to go to the wait queue before using up their time quantum, they gain priority (because they probably are I/O bound).
- When processes finish their time quantum uninterrupted, they lose priority (because they probably are CPU bound).
- When processes wait on locks, the process that is holding the lock being waited on gains priority based on the waiting process’s priority (because we’d like the lock to be released as soon as possible) – this is called priority donation.
• Various unix systems use a type of priority scheduling
• Can lead to certain processes getting very little or no CPU time on busy systems
• Processes can pretend to be I/O bound by doing spurious I/O to /dev/null

12 Bin Packing and Knapsack

• We would like to schedule k jobs, each of which have running times \( t_i \), \( 0 \leq i \leq k - 1 \) – these running times cannot be broken into pieces (each job must be run to completion at once)
• We have n machines, each of which can be run for \( h_j \), \( 0 \leq j \leq n - 1 \) hours a day
• How do we schedule the jobs across the machines?
• We can treat each machine as a “bin” in which we can place “items”
• Each bin has a capacity (in this case, the number of hours a machine can be run on a given day) and each item has a size (in this case, the number of hours needed to run a job)
• We want to maximally pack the bins
• The greedy approach to solving this won’t give us the optimal solution (why not?)
• We can either use heuristic based approaches to solving it (such as putting items in the first bin they fit in) or we can try all possible bin packings and choose the optimal one
• The knapsack problem has a similar formulation – we want maximize our profit when choosing among a set of items of different sizes (and values) with the constraint of a fixed size bag
• Again, the greedy approach will not give us an optimal solution