CIS520 – Operating Systems
Handout 6
CPU Scheduling

- What is CPU scheduling? Determining which processes run when there are multiple runnable processes. Why is it important? Because it can can have a big effect on resource utilization and the overall performance of the system.
- By the way, the most popular operating systems in the world (DOS, Mac) have NO sophisticated CPU scheduling algorithms. They are single threaded and run one process at a time until the user directs them to run another process. Why is this true, and will future operating systems have this property?
- Basic assumptions behind most scheduling algorithms:
  - There is a pool of runnable processes contending for the CPU.
  - The processes are independent and compete for resources.
  - The job of the scheduler is to distribute the scarce resource of the CPU to the different processes "fairly" (according to some definition of fairness) and in a way that optimizes some performance criteria.

  In general, these assumptions are starting to break down. First of all, CPUs are not really that scarce - almost everybody has at least one, and pretty soon people will be able to afford lots. Second, many applications are starting to be structured as multiple cooperating processes. So, a view of the scheduler as mediating between competing entities may be partially obsolete.
- How do processes behave? First, CPU/IO burst cycle (OSC Sec. 5.1.1). A process will run for a while, perform some IO, then run for a while more. How long between IO operations? Depends on the process.
  - IO Bound processes: processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO, then more IO happens.
  - CPU bound processes: processes that perform lots of computation and do little IO. Tend to have a few long CPU bursts.

  One of the things a scheduler will typically do is switch the CPU to another process when one process does IO. Why? The IO will take a long time, and don't want to leave the CPU idle while wait for the IO to finish.
- When look at CPU burst times across the whole system, have the exponential or hyperexponential distribution in Fig. 5.2.
- What are possible process states?
  - Running - process is running on CPU.
  - Ready - ready to run, but not actually running on the CPU.
  - Waiting - waiting for some event like IO to happen.
- When do scheduling decisions take place? When does CPU choose which process to run? Are a variety of possibilities:

  - When process switches from running to waiting. Could be because of IO request, because wait for child to terminate, or wait for synchronization operation (like lock acquisition) to complete.
  - When process switches from running to ready - on completion of interrupt handler, for example. Common example of interrupt handler - timer interrupt in interactive systems. If scheduler switches processes in this case, it has preempted the running process. Another common case interrupt handler is the IO completion handler.
  - When process switches from waiting to ready state (on completion of IO or acquisition of a lock, for example).
  - When a process terminates.

- How to evaluate scheduling algorithm? There are many possible criteria:
  - CPU Utilization: Keep CPU utilization as high as possible. (What is utilization, by the way?).
  - Throughput: number of processes completed per unit time.
  - Turnaround Time: mean time from submission to completion of process.
  - Waiting Time: Amount of time spent ready to run but not running.
  - Response Time: Time between submission of requests and first response to the request.
  - Scheduler Efficiency: The scheduler doesn't perform any useful work, so any time it takes is pure overhead. So, need to make the scheduler very efficient.

- Big difference: Batch and Interactive systems. In batch systems, typically want good throughput or turnaround time. In interactive systems, both of these are still usually important (after all, want some computation to happen), but response time is usually a primary consideration. And, for some systems, throughput or turnaround time is not really relevant - some processes conceptually run forever.

- Difference between long and short term scheduling. Long term scheduler is given a set of processes and decides which ones should start to run. Once they start running, they may suspend because of IO or because of preemption. Short term scheduler decides which of the available jobs that long term scheduler has decided are runnable to actually run.

- Let's start looking at several vanilla scheduling algorithms.

- First-Come, First-Served. One ready queue, OS runs the process at head of queue, new processes come in at the end of the queue. A process does not give up CPU until it either terminates or performs IO.

- Consider performance of FCFS algorithm for three compute-bound processes. What if have 4 processes P1 (takes 24 seconds), P2 (takes 3 seconds) and P3 (takes 3 seconds). If arrive in order P1, P2, P3, what is
  - Waiting Time? $(24 + 27)/3 = 17$
  - Turnaround Time? $(24 + 27 + 30) = 27$.
  - Throughput? $30/3 = 10$.

  What about if processes come in order P2, P3, P1? What is
  - Waiting Time? $(3 + 3)/2 = 6$
  - Turnaround Time? $(3 + 6 + 30) = 13$.
  - Throughput? $30/3 = 10$.

- Shortest-Job-First (SJF) can eliminate some of the variance in Waiting and Turnaround time. In fact, it is optimal with respect to average waiting time. Big problem: how does scheduler figure out how long will it take the process to run?

- For long term scheduler running on a batch system, user will give an estimate. Usually pretty good - if it is too short, system will cancel job before it finishes. If too long, system will hold off on running the process. So, users give pretty good estimates of overall running time.

- For short-term scheduler, must use the past to predict the future. Standard way: use a time-decayed exponentially weighted average of previous CPU bursts for each process. Let T_n be the measured burst time of the nth burst, τ_n be the predicted size of next CPU burst. Then, choose a weighting factor α, where $0 \leq \alpha \leq 1$ and compute

$$\tau_{n+1} = \alpha T_n + (1 - \alpha)\tau_n$$

  τ_0 is defined as some default constant or system average.

- α tells how to weight the past relative to future. If choose $\alpha = .5$, last observation has as much weight as entire rest of the history. If choose $\alpha = 1$, only last observation has any weight. Do a quick example.

- Preemptive vs. Non-preemptive SJF scheduler. Preemptive scheduler reruns scheduling decision when process becomes ready. If the new process has priority over running process, the CPU preempts the running process and executes the new process. Non-preemptive scheduler only does scheduling decision when running process voluntarily gives up CPU. In effect, it allows every running process to finish its CPU burst.

- Consider 4 processes P1 (burst time 8), P2 (burst time 4), P3 (burst time 9) P4 (burst time 5) that arrive one time unit apart in order P1, P2, P3, P4. Assume that after burst happens, process is not reenabled for a long time (at least 100, for example). What does a preemptive SJF scheduler do? What about a non-preemptive scheduler?

- Priority Scheduling. Each process is given a priority, then CPU executes process with highest priority. If multiple processes with same priority are runnable, use some other criteria - typically FCFS. SJF is an example of a priority-based scheduling algorithm. With the exponential decay algorithm above, the priorities of a given process change over time.

- Assume we have 5 processes P1 (burst time 10, priority 3), P2 (burst time 1, priority 1), P3 (burst time 2, priority 3), P4 (burst time 1, priority 4), P5 (burst time 5, priority 2). Lower numbers represent higher priorities. What would a standard priority scheduler do?

- Big problem with priority scheduling algorithms: starvation or blocking of low-priority processes. Can use aging to prevent this - make the priority of a process go up the longer it stays runnable but isn't run.

- What about interactive systems? Cannot just let any process run on the CPU until it gives it up - must give response to users in a reasonable time. So, use an algorithm called round-robin scheduling. Similar to FCFS but with preemption. Have a time quantum or time slice. Let the first process in the queue run until it expires its quantum (i.e. runs for as long as the time quantum), then run the next process in the queue.

- Implementing round-robin requires timer interrupts. When schedule a process, set the timer to go off after the time quantum amount of time expires. If process does IO before timer goes off, no problem - just run next process. But if process expires its quantum, do a context switch. Save the state of the running process and run the next process.

- How well does RR work? Well, it gives good response time, but can give bad waiting time. Consider the waiting times under round robin for 3 processes P1 (burst time 24), P2 (burst time 3), and P3 (burst time 4) with time quantum 4. What happens, and what is average waiting time? What gives best waiting time?

- What happens with really a really small quantum? It looks like you've got a CPU that is 1/n as powerful as the real CPU, where n is the number of processes. Problem with a small quantum - context switch overhead.

- What about having a really small quantum supported in hardware? Then, you have something called multithreading. Give the CPU a bunch of registers and heavily pipeline the execution. Feed the processes into the pipe one by one. Treat memory access like IO - suspend the thread until the data comes back from the memory. In the meantime, execute other threads. Use computation to hide the latency of accessing memory.

- What about a really big quantum? It turns into FCFS. Rule of thumb - want 80 percent of CPU bursts to be shorter than time quantum.

- Multilevel Queue Scheduling - like RR, except have multiple queues. Typically, classify processes into separate categories and give a queue to each category. So, might have system, interactive and batch processes, with the priorities in that order. Could also allocate a percentage of the CPU to each queue.
- Multilevel Feedback Queue Scheduling - Like multilevel scheduling, except processes can move between queues as their priority changes. Can be used to give IO bound and interactive processes CPU priority over CPU bound processes. Can also prevent starvation by increasing the priority of processes that have been idle for a long time.
- A simple example of a multilevel feedback queue scheduling algorithm. Have 3 queues, numbered 0, 1, 2 with corresponding priority. So, for example, execute a task in queue 2 only when queues 0 and 1 are empty.
- A process goes into queue 0 when it becomes ready. When run a process from queue 0, give it a quantum of 8 ms. If it expires its quantum, move to queue 1. When execute a process from queue 1, give it a quantum of 16. If it expires its quantum, move to queue 2. In queue 2, run a RR scheduler with a large quantum if in an interactive system or an FCFS scheduler if in a batch system. Of course, preempt queue 2 processes when a new process becomes ready.
- Another example of a multilevel feedback queue scheduling algorithm: the Unix scheduler. We will go over a simplified version that does not include kernel priorities. The point of the algorithm is to fairly allocate the CPU between processes, with processes that have not recently used a lot of CPU resources given priority over processes that have.
- Processes are given a base priority of 60, with lower numbers representing higher priorities. The system clock generates an interrupt between 50 and 100 times a second, so we will assume a value of 60 clock interrupts per second. The clock interrupt handler increments a CPU usage field in the PCB of the interrupted process every time it runs.
- The system always runs the highest priority process. If there is a tie, it runs the process that has been ready longest. Every second, it recalculates the priority and CPU usage field for every process according to the following formulas.
  - CPU usage field = CPU usage field / 2
  - Priority = CPU usage field / 2 + base priority
- So, when a process does not use much CPU recently, its priority rises. The priorities of IO bound processes and interactive processes therefore tend to be high and the priorities of CPU bound processes tend to be low (which is what you want).
- Unix also allows users to provide a "nice" value for each process. Nice values modify the priority calculation as follows:
  - Priority = CPU usage field / 2 + base priority + nice value

  So, you can reduce the priority of your process to be "nice" to other processes (which may include your own).
- In general, multilevel feedback queue schedulers are complex pieces of software that must be tuned to meet requirements.
- Anomalies and system effects associated with schedulers.
- Priority interacts with synchronization to create a really nasty effect called priority inversion. A priority inversion happens when a low-priority thread acquires a lock, then a high-priority thread tries to acquire the lock and blocks. Any middle-priority threads will prevent the low-priority thread from running and unlocking the lock. In effect, the middle-priority threads block the high-priority thread.
- How to prevent priority inversions? Use priority inheritance. Any time a thread holds a lock that other threads are waiting on, give the thread the priority of the highest-priority thread waiting to get the lock. Problem is that priority inheritance makes the scheduling algorithm less efficient and increases the overhead.

- Preemption can interact with synchronization in a multiprocessor context to create another nasty effect - the convoy effect. One thread acquires the lock, then suspends. Other threads come along, and need to acquire the lock to perform their operations. Everybody suspends until the lock that has the thread wakes up. At this point the threads are synchronized, and will convoy their way through the lock, serializing the computation. So, drives down the processor utilization.
- If have non-blocking synchronization via operations like LL/SC, don't get convoy effects caused by suspending a thread competing for access to a resource. Why not? Because threads don't hold resources and prevent other threads from accessing them.
- Similar effect when scheduling CPU and IO bound processes. Consider a FCFS algorithm with several IO bound and one CPU bound process. All of the IO bound processes execute their bursts quickly and queue up for access to the IO device. The CPU bound process then executes for a long time. During this time all of the IO bound processes have their IO requests satisfied and move back into the run queue. But they don't run - the CPU bound process is running instead - so the IO device idles. Finally, the CPU bound process gets off the CPU, and all of the IO bound processes run for a short time then queue up again for the IO devices. Result is poor utilization of IO device - it is busy for a time while it processes the IO requests, then idle while the IO bound processes wait in the run queues for their short CPU bursts. In this case an easy solution is to give IO bound processes priority over CPU bound processes.
- In general, a convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. Causes poor utilization of the other resources in the system.