CMPSCI 377 Operating Systems **Fall 2012**

Lecture 5: September 19

Lecturer: Prashant Shenoy *TA: Sean Barker & Demetre Lavigne*

5.1 Scheduling

5.1.1 Types of Schedulers

Operating systems often utilize two types of schedulers: a long-term scheduler (also known as an admission scheduler or high-level) and a short-term scheduler (also known as a dispatcher). The names suggest the relative frequency with which these functions are performed. In this class when we talk about scheduling, we generally mean short-term scheduling. We also make some simplifying assumptions when talking about scheduling algorithms: one process per user, one thread per process, and that processes are independent of one another. The algorithms we will look at were developed in the 1970s when these assumptions were more realistic. A research problem today is how to relax these assumptions.

In general a scheduler will want to favour I/O bound jobs over CPU bound jobs. An I/O bound job is one that does little actual computation and spends most of it's time waiting for I/O. A CPU bound job is the opposite of I/O bound: CPU bound jobs rarely do I/O and will hardly ever give up their time on the CPU before the scheduler forces them off the CPU. Wanting to maximize the use of the hardware leads to an overlap of I/O and computation (because I/O devices can be busy while other jobs are using the CPU).

5.1.1.1 Long-term Scheduler

The long-term, or admission, scheduler decides which jobs or processes are to be admitted to the ready queue and how many processes should be admitted into the ready queue. This controls the total number of jobs which can be running within the system. In practice, this limit is generally not reached, but if a process attempts to fork off a large number of processes it will eventually reach an OS defined limit where it will prevent any further processes from being created. This type of scheduling is very important for a real-time operating system, as the system's ability to meet process deadlines may be compromised by the slowdowns and contention resulting from the admission of more processes than the system can safely handle.

5.1.1.2 Short-term Scheduler

The short-term scheduler (also known as the dispatcher) decides which of the ready, in-memory processes are to be executed (allocated a CPU) next following a clock interrupt, an I/O interrupt, or an operating system call. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term schedulers - a scheduling decision will at a minimum have to be made after every time slice, which can be as often as every few milliseconds. This scheduler can be *preemptive*, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or *non-preemptive*, in which case the scheduler is unable to "force" processes off the CPU.

5.1.2 Scheduling Criteria

In this lecture we will discuss short term schedulers. Short term schedulers determine which process to run in a given time interval. Short term schedulers can be either *non-preemptive*, requiring tasks to explicitly give up the CPU when they are done, or *preemptive* where the OS scheduler can interrupt running processes and change their scheduling order.

There are several criteria to design a scheduler for. A scheduler should generally maximize CPU utilization – keeping the CPU as busy as possible generally leads to more work being completed. The throughput of a system is a measure of how many jobs are finished in a unit time. The turn around time is the amount of time which elapses from when a job arrives until it is completed. The waiting time of a process is how long a job spends sitting in the ready queue (e.g. the time spent not actively running or performing I/O). Finally, response time is a criteria important for many interactive jobs. Response time measures the time from when a process is ready to run until its next I/O activity. As an example, this corresponds to the delay between when keys are pressed on the keyboard (an I/O activity) and when characters appear on the screen. Ideally, a scheduler would optimize all of these criteria, but in practice most schedulers must optimize a few at the expense of others.

5.2 First Come First Served (FCFS)

The First-Come-First-Served, or First-In-First-Out (FIFO), algorithm is the simplest scheduling algorithm. Processes are dispatched in order according to their arrival time on the ready queue. Being a non-preemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait. FCFS is more predictable than most of other schemes since the scheduling order can be easily understood and the code for FCFS scheduling is simple to write and understand. However, the FCFS scheme is not useful in scheduling interactive processes because it cannot guarantee good response time. One of the major drawbacks of this scheme is that the average time is often quite long. Short jobs or jobs that frequently perform I/O can have very high waiting times since long jobs can monopolize the CPU. FCFS originally didn't have a job relinquish the CPU when it was doing I/O. We assume that a FCFS scheduler will run when a process is doing I/O, but it is still non-preemptive. The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other schemes.

Advantage: simple
Disadvantage: poor performance for short jobs or tasks performing frequent I/O operations

5.3 Round Robin Scheduling (RR)

Round-robin (RR) is one of the simplest preemptive scheduling algorithms for processes in an operating system. RR assigns short time slices to each process in equal portions and in order, cycling through the processes. Round-robin scheduling is both simple and easy to implement, and starvation-free (all processes will receive some portion of the CPU). Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks. However, round robin does not support prioritizing jobs, meaning that less important tasks will receive the same CPU time as more important ones. Most time sharing systems actually use variations of round robin for scheduling.

Example: The time slot could be 100 milliseconds. If job1 takes a total time of 250ms to complete, the

round-robin scheduler will suspend the job after 100ms and give other jobs their time on the CPU. Once the other jobs have had their equal share (100ms each), job1 will get another allocation of CPU time and the cycle will repeat. This process continues until the job finishes and needs no more time on the CPU.

A challenge with the RR scheduler is determining how to determine the time slice to allocate tasks. If the time slice is too large, then RR performs similarly to FCFS. However, if the time slice is too small, then the overhead of performing context switches so frequently can reduce overall performance. The OS must pick a time slice which balances these tradeoffs. A general rule is that the time for a context switch should be roughly 1% of the time slice.

Advantage: Fairness (each job gets an equal amount of the CPU)
Disadvantage: Average waiting time can be bad (especially when the number of processes is large)

5.4 Shortest Job First (SJF)

Shortest Job First (SJF) is a scheduling policy that selects the waiting process with the smallest (expected) amount of work (CPU time) to execute next. Once a job is selected it is run non-preemptively. Shortest job first is advantageous because it provably minimizes the average wait time. However, it is generally not possible to know exactly how much CPU time a job has remaining so it must be estimated. SJF scheduling is rarely used except in specialized environments where accurate estimations of the runtime of all processes are possible. This scheduler also has the potential for process starvation for processes which will require a long time to complete if short processes are continually added. A variant of this scheduler is Shortest Remaining Time First (SRTF) which is for jobs which may perform I/O. The SRTF scheduler brings task back to the run queue based on the estimated time remaining to run after an I/O activity. SRTF also picks the shortest job (like SJF), but it runs it for a quantum (time slice), then does a context switch to the next shortest remaining time job. This means that SRTF is unlike SJF in that it is a preemptive scheduler.

Advantage: Minimizes average waiting time. Works for preemptive and non-preemptive schedulers. Gives priority to I/O bound jobs over CPU bound jobs.
Disadvantage: Cannot know how much time a job has remaining. Long running jobs can be starved for CPU.

5.5 Multilevel Feedback Queues (MLFQ)

The Multilevel feedback queue scheduling algorithm is what is used in Linux and Unix systems. The MLFQ algorithm tries to pick jobs to run based on their observed behavior. It does this by analysing whether a job runs for its full time slice or if it relinquishes the CPU early to do I/O. This allows the scheduler to determine if a job is I/O bound or CPU bound so that they can be treated differently. This allows the scheduler to approximate the behavior of a SJF scheduler without requiring perfect information about job completion times. While over long periods of time a job may change from I/O to CPU bound and back, over a short period it will tend to behave in one manner or the other.

The MLFQ scheduler uses several different ready queues and associates a different priority with each queue. Each queue also has a time slice associated with it; the highest priority queue has the smallest time slice. When a new job arrives, it is assigned to the highest priority queue by default. At each scheduling decision, the Algorithm attempts to choose a process from the highest priority queue that is not empty. Within the highest priority non-empty queue, jobs are scheduled with a round robin scheduler.

When a job relinquishes the CPU it is because either it has used its full time quantum, or because it has

paused to perform an I/O activity. The scheduler uses this information to change the priority of the job; CPU bound jobs using their full time quantum are moved down one priority level, while I/O bound jobs that give up their time slice are moved to a one level higher priority queue. This approach gives I/O bound tasks better response times since they are given higher priority when they need the CPU. The MLFQ scheduler has a starvation problem just like SJF if there is a continual stream of small jobs. There are potential schemes to prevent starvation, for example: low priority tasks can periodically be bumped up to higher priority queues if they have not been run in a long time.

Advantage: Adaptive behavior can distinguish between CPU and I/O bound tasks and schedule them accordingly.
Disadvantage: More complicated. Uses past behavior to predict future.

5.6 Lottery Scheduling

Lottery Scheduling is a probabilistic scheduling algorithm for processes in an operating system. Processes are each assigned some number of lottery tickets, and the scheduler draws a random ticket to select the next process. The distribution of tickets need not be uniform; granting a process more tickets provides it a relative higher chance of selection. This technique can be used to approximate other scheduling algorithms, such as SJF and Fair-share scheduling. Allocating more tickets to a process gives it a higher priority since it has a greater chance of being scheduled at each scheduling decision.

Lottery scheduling solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has a non-zero probability of being selected at each scheduling operation. On average, CPU time is proportional to the number of tickets given to each job. For approximating SJF, most tickets are assigned to short running jobs and fewer to longer running jobs. To avoid starvation, every job gets at least one ticket. Implementations of lottery scheduling should take into consideration that there could be a large number of tickets distributed among a large pool of threads.