CS 547 Lecture 17: Scheduling

Daniel Myers

Thus far, we've only considered first-come-first-serve scheduling in our queueing models. Though FCFS is a popular choice for many real applications, other scheduling disciplines are possible.

Classifiying Scheduling Disciplines

For this lecture, we'll classify scheduling strategies based on two qualities

- preemptive vs. non-preemptive
- size-based vs. non-size-based

In *preemptive* scheduling, it's possible to stop a currently executing job, run something else, then return the original job to service at a later time. We almost always assume that preempted jobs return to service at the point the left off. In a *non-preemptive* scheduler, a job only runs to completion once started.

In a *size-based* algorithm, we know the service requirement of each job and can take these requirements into account when making scheduling decisions. This allows us to give priority to shorter jobs, which generally improves performance.

Several metrics have been proposed to evaluate the quality of scheduling disciplines. In this lecture, we'll use average residence time, $\overline{R}$, as our metric of choice.

Non-Preemptive, Non-Size-Based

This category includes all disciplines that run jobs to completion and do not consider job size.

FCFS falls into this category. Two other popular disciplines with these characteristics are last-come-first-serve (LCFS) and RANDOM.

- LCFS: when the server becomes available, run the job that arrived most recently
- RANDOM: when the server becomes available, choose randomly from among all waiting jobs

Somewhat surprisingly, all of these disciplines have the same average residence time. In fact, *any* non-preemptive, non-size-based scheduling strategy has the same average residence time as FCFS. By extension, we can use the M/G/1 residence time equation to calculate $\overline{R}$ in any of these queues.

A full proof of this theorem requires reasoning about the full behavior of the queue length distribution using transform theory, but we can easily analyze the LCFS case to show why the theorem is reasonable.

Consider the waiting time in an M/G/1-LCFS queue. There are two components to the waiting time: the residual of the customer currently in service (because all jobs run to completion), and the waiting time due to jobs that arrive *after* the tagged job arrives, but before it enters service.

On average, we expect $\lambda\overline{W}$ customers to arrive during the tagged customer's waiting period, and each requires an average service time of $\overline{s}$.

$$\begin{aligned}\overline{W} &= U\frac{\overline{s}}{2}(1+c_s^2) + \lambda\overline{W}\,\overline{s} \\ &= \frac{\overline{s}U(1+c_s^2)}{2(1-U)}\end{aligned}$$

This result is exactly the waiting time in an M/G/1-FCFS queue.

Preemptive, Non-Size-Based

This category includes disciplines that can start and stop jobs at will, but do not take into account job size when making decisions. There are two main disciplines in this category: *last-come-first-serve-preemptive-resume* (LCFSPR) and *processor sharing* (PS).

LCFSPR

LCFSPR is the preemptive version of LCFS scheduling.

- LCFSPR: when a new job arrives to the queue, preempt the currently running job and put the new job into service. When a job departs, return the most recent remaining job back to service. Jobs that return to service after being preempted resume at the point they left off.

A customer arriving to an LCFSPR queue does not have to wait for the customer currently in service, but it may be preempted by other jobs that arrive while it is running. Those jobs, in turn, may be preempted by other arriving jobs. If the customer's average residence time is $\overline{R}$, then we expect $\lambda\overline{R}$ jobs to arrive during the residence period.

$$\begin{aligned}\overline{R} &= \lambda\overline{R}\,\overline{s} + \overline{s} \\ &= \frac{\overline{s}}{1-U}\end{aligned}$$

This is an unexpected result. The average residence time in a M/G/1-LCFSPR queue is the same as an M/M/1 queue!

PS

Many computer systems use *round-robin* (RR) scheduling for the processor. In RR scheduling, each job is allowed to run for a certain *quanta* of time. When a job's quanta expires, the operating system stops the job and performs a context switch to allow a new job to run. At some time in the future, the OS will return the stopped job back to service, where it will be allowed to run for another quanta.

Processor sharing scheduling is the idealized form of round robin with infinitely small quanta. If there are k jobs in the queue at a given moment, each job is simultaneously receiving exactly $\frac{1}{k}$ of the server's total processing power at that instant. Analyzing PS is non-trivial, but it's possible to show that

$$\overline{R} = \frac{\overline{s}}{1-U}$$

Discussion

We've now found three different cases with the same average residence time: M/G/1-LCFSPR, M/G/1-PS, and M/M/1-FCFS. Is there an intuitive explanation for this similarity?

In all three cases, a newly arriving customer does not suffer a great penalty for arriving behind a currently running long job. In LCFSPR, the new customer gets to actually preempt the long job. In PS, the two jobs must share the processor, but the new job is able to begin receiving service immediately. In M/M/1-FCFS, the service times are memoryless, so the expected residual service time of the current job should not be worse than an average service time.

Further, note that PS and LCFSPR queues are invariant to the job size variability – there is no c_s^2 term in the $\overline{R}$ equations. Therefore, if $c_s^2 > 1$, using PS or LCFSPR scheduling will result in a performance improvement without making any other change to the system.

Size-Based

Knowing the distribution of job sizes allows us to give preference to smaller jobs. It's possible to show that favoring smaller jobs will improve average residence time.

Shortest-job-first (SJF) scheduling is the main non-preemptive size-based policy.

- SJF: when the processor becomes available, run the waiting job with the smallest service requirement

It's possible to show that $\overline{R}_{SJF} \leq \overline{R}_{FCFS}$.

SJF is still non-preemptive, so there is a possibility that arriving jobs will still have to wait for a long residual service time.

Shortest-remaining-processing-time (SRPT) is the preemptive version of SJF. For this strategy, we keep track of each job's remaining service requirement, taking into account how much service it may have already accumulated.

- SRPT: when the processor becomes available, run the waiting job with the smallest remaining service time requirement. If a new job arrives and has a smaller service time requirement than the currently running job, preempt the current job and put the new job into service.

SRPT is the best scheduling algorithm, in that it minimizes residence time. It has two very desirable properties. First, no job ever waits for a longer job to finish. Second, jobs with large initial service requirements will be able to run with high priority once they accumulate enough service time; this prevents long jobs from being indefinitely "starved" of service.

Heuristics

Running short jobs improves average residence time, but it has one major drawback: we need to actually know the service requirement of each job!

To combat this problem, many systems use heuristic strategies that do not rely on job size, but still attempt to favor short jobs. The basic unifying idea of these strategies is run new jobs with high priority, then steadily decrease a job's priority the longer it remains in the system. Thus, newly arriving short jobs can go immediately into service and finish quickly.

Most modern operating systems use some form of *multi-level feedback queue* (MLFQ). In an MLFQ scheduler, each job is assigned a priority level. Jobs at lower priority levels are only allowed to run if there are no higher-priority jobs currently in the system. When a new job arrives, it is automatically assigned the highest priority level and begins running immediately. Typically, a job's priority is reduced once it's accumulated a certain amount of service time at its current level. If multiple jobs at the same priority level are present, they share the processor in round-robin fashion.

There are many variations to the basic MLFQ idea. For example,

- the system may have dozens of distinct priority levels
- the very highest priorities may be reserved for the OS
- the running quanta of a job may depend on its priority – giving lower priorities a longer quanta allows CPU-bound jobs to make more progress when they finally run
- I/O bound jobs may be allowed to stay at high priority to improve responsiveness
- periodically restoring a low-priority job to high-priority keeps long jobs from starving