Queueing Theory and Scheduling

Scheduling in an operating system or elsewhere[1] is a very interesting mix of complex engineering and theory. The purpose of this document is to describe the basics of queueing theory, which is a key approach to scheduling from a theoretical perspective, and to describe some interesting results and scheduling disciplines beyond the ones noted in the textbook. The workload characterization document should be read before or at least in tandem with this one---scheduling is one area in which different workloads can lead to radically different results, even at the theoretical level.

Queuing and scheduling theory is used in two ways. The first is to analyze an existing system design and determine, ideally analytically, its performance characteristics. The second is to help to design a new system, providing input in terms of its structure, the necessary capacity for the parts of its structure, and how to schedule these elements.

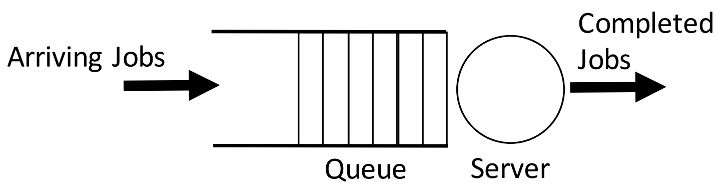
Queueing and scheduling theory is an active area of research, both in terms of applying it to solve systems, networking, and other problems, and in terms of making the theory itself more capable of solving problems, in particular given workloads that better approximate reality.

Other fields, for example industrial engineering and operations research, are also important users and contributors to queueing and scheduling theory. Indeed, these two fields are actually quite older than computer science or engineering in their application and development of these theories. In a lot of ways, a computer system has scheduling and structural problems like a "job shop", assembly line, or factory, in these fields, but operates at warp speed.

We will draw on the treatment from Jain[2], and Harchol-Balter[3], both of which are great resources (for different reasons), if you would like to learn more.

A Simple Queueing System

Below is perhaps simplest queueing system:

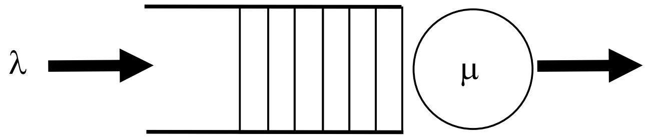


[1] Scheduling is choosing what goes where when in order to optimize some objective function while obeying a set of constraints. Since it is such a general problem, it occurs in all layers of system design, from microops within a processor core to wide-area distributed systems. The focus in this document is on *online* scheduling, in which we do not know about all the "whats" ahead of time.
[2] Raj Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, 1991.
[3] Mor Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, Cambridge, 2013.

Jobs (as described in the workload characterization document) are placed into the *queue* when they *arrive*. The queue has infinite size. A *scheduling policy* or *service discipline* (not shown), selects the next job from the queue that will advance to the *server*, and how long the job will run on the server. Once the job is completed, it departs from the system. Here, "server" does not mean a server computer, but rather any schedulable resource. In the context of a typical kernel, a good example of a server is a CPU or hardware thread. Also, job is very broadly defined. In a typical kernel, it might be a software thread's burst of computation between I/O operations.

M/M/1: Solving An Even Simpler Version of The Simple Queueing System

We are now going to sketch the analytic solution[4] of the simple queueing system, under very limiting assumptions, known as M/M/1. The "1" just means there is only one queue. We will introduce general terminology as we go. Let's start by putting some more notation on the simple queueing system:



The idea here is that jobs arrive at the queue at an *arrival rate* λ - that is, there are λ jobs per second. The jobs arrive at random points in time, with the time between job arrivals being a random variable selected from an exponential distribution. This is also called a Poisson arrival process. Because the exponential distribution is memoryless, this forms part of a "Markovian assumption"---it's also the first "M" in M/M/1. Each job will require that the server do work that will take some time. This is the job's *service time*, T_s. Service times will also be drawn from an exponential distribution---that's the second "M" in M/M/1. The mean of this distribution is the *mean service time.* The reciprocal of the mean service time is the *service rate*, μ. We assume here that the two rates are *stationary,* meaning they do not depend on time.

The system is fully parameterized by just these two exponential distributions.[5] We will also assume the service discipline is the simplest possible one: first-come-first-served (FCFS), also called first-in-first-out (FIFO). Jobs are handled in the order in which they arrive. There is no preemption, so once a job starts running on the server, it runs to completion and then leaves.

Let's draw an analogy with a grocery store checkout aisle. The jobs are customers ready to make their purchases. The server is the cashier. The queue is the line of customers waiting for checkout. The FCFS/FIFO discipline is exactly how the line and the cashier work---the customers can't cut in line, and the cashier won't start working on the next customer's purchase until they are done with the current customer's purchase. The arrival rate is the rate at which customers show up and get in line. The service time is analogous to the how quickly the cashier

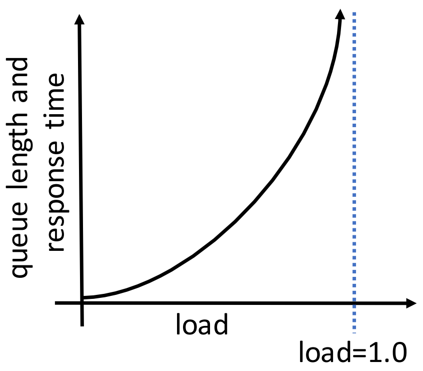
[4] "Analytic solution" means we will derive closed form equations for values that we care about.
[5] This is a major simplifying assumption about the workload that makes analytic solution of this problem very tractable. In reality, many/most workloads do NOT conform to this assumption. Workload characterization is essential to determining whether any statistical model is relevant.

can check out the customer.[6] The service rate is how many customers per second are checked out.

A newly arriving job will have some number of jobs ahead of it. These jobs will delay the new job. The amount of that delay is called the *queueing delay* of the job, T_q. The *mean queueing delay* is the expected queueing delay seen by any new job. In the analogy, this is how long a new customer can expect to wait before they are at the head of the grocery store checkout aisle.

The time from when a job arrives to when it is finished is the *response time*, T_r, which consists of the queueing delay and the service time: $T_r = T_q + T_s$. We can also characterize the whole system as having a *mean response time*, which is the expected time a new job spends in the system. In the analogy, this is how long between when a new customer gets in line and when they leave the store.

The *load* on the system is λ/μ. If the load is greater than one, the server cannot keep up with the incoming jobs, the number of jobs in the queue grows without bound, and the mean queuing delay goes to infinity. This is an *overload* situation. If the load is much less than one, the server can easily keep up, and there are few if any jobs in the queue. As the load approaches one from below, the queue length grows, in fact, it will grow to infinity:



The queue length (or queue depth) is the number of jobs ahead of a new job that arrives in the system. The more jobs ahead of the new job, the longer it will take for the new job to start running, and the longer its response time. The load described here is over the very long term. *Transient overload* is a different matter. You will often find the load measurements (which you can see using, for example, the uptime, top, or htop commands) on a computer to be above 1.0 temporarily.

From the perspective of a job (a customer in the store), we clearly want to know the mean (or expected) response time (how long the customer is likely to spend in the checkout line.) *Little's Law* tells us this:

mean response time $= n / \lambda$

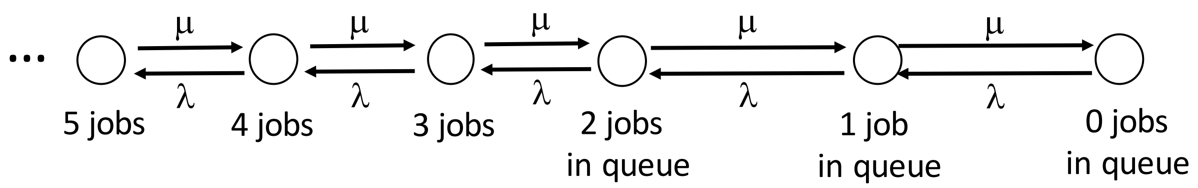
[6] You might think this is a bit strange since customers have a widely varying number of items in their cart. Under the exponential service time model, however, this is not really the case. That the exponential service time model does not fit grocery store customers well is absolutely true.

Here λ is the arrival rate, as before, while *n* is the average number of jobs in the system (or customers in the checkout line in our analogy). Surprisingly, Little's Law is true under a wide range of assumptions, not just the very restrictive set of assumptions we are considering.

So, to find the mean response time, we are left needing to find *n*, the expected queue depth. To determine this, we need to know the probability distribution over queue depths. If we knew this, we could compute *n* something like this:

$$n \ = \ \sum_{i=0}^{\infty} i \times P[queue\ depth\ is\ i]$$

Now we need to determine the probability distribution. Here is where the Markovian stuff really comes to play. Suppose the current queue depth is 5. It could next become 6 because of a job arrival, or it could next become 4 because of a job completion. The next queue depth depends only on the current queue depth. Furthermore, because of this direct dependence, we can think of this problem is a *Markov chain*. Consider the following picture, ignoring the μs and λs for the moment.



This a probabilistic state transition diagram. In the middle, we have the "2 jobs in queue" state. Suppose we are in that state, and a brief interval of time passes. What could happen? With some probability, we will transition to the "3 jobs" state. With some other probability, we will transition into the "1 job in queue" state. With the remaining probability, we will stay in the "2 jobs in queue" state. These probabilities depend on μ and λ, the service and arrival rates.

We are now going to do some hand-waving to argue that solving this is tractable (you can easily find detailed descriptions of how this is solved in the referenced books), and skip to the solution:

$$P[queue\ depth\ is\ 0] \ = \ 1 - \frac{\lambda}{\mu}$$
$$P[queue\ depth\ is\ i] \ = \left(\frac{\lambda}{\mu}\right) \times P[queue\ depth\ is\ i-1]$$
$$P[queue\ depth\ is\ i] \ = \left(\frac{\lambda}{\mu}\right)^{i} \times P[queue\ depth\ is\ 0] \ = \ \left(\frac{\lambda}{\mu}\right)^{i}\left(1-\frac{\lambda}{\mu}\right)$$

The point here is that we have computed an *analytic* probability distribution over queue depths.

We can now use this distribution to determine *n*, the expected queue depth:

$$n \ = \ \frac{\lambda}{\mu-\lambda}$$

Consider what this is saying: as the service and arrival rates get closer together, the expected queue depth will get larger. As they get incrementally closer and closer, the expected queue depth grows faster and faster. This gives the asymptote of the graph shown previously.

We also now get the mean response time, from Little's Law:

$$mean\ response\ time\ = \frac{\lambda}{\mu-\lambda} \times \frac{1}{\lambda} = \frac{1}{\mu-\lambda}$$

As you might expect, as the expected queue depth grows, so does the mean response time. The closer we get to the load limit of 1.0, the more we see the mean response time explode exponentially. This is one reason why systems are run with load kept to be a bit less than 1.0

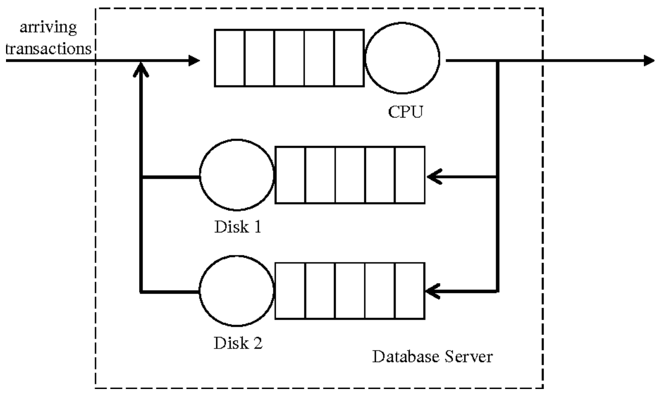
It is important to remember that these results are true only in the very constrained M/M/1 case we are considering. Finding analytic solutions/results in less constrained situations is a hot topic of research.

It is also important to understand that this very constrained case, particularly the "M/M" part, is much rarer in practice than might meet the eye. For example, you probably have the intuition that a supermarket checkout line does not work as we have described. And you would be right. In computer systems, simple measurements can be deceiving, suggesting that you have a workload that might comply with this analysis, when it may not.

Finally, it is important to note that other objective functions than mean response time can be considered when evaluating a system. For example, the *slowdown* of a job, or ratio of response time to service time (T_r/T_s), and the *mean slowdown,* turn out to be quite important when more naturally occurring workloads are considered. We also often care about the second moment (variance) of response time and slowdown---a system with high variance is generally worse than one with low variance. We occasionally also care about various metrics of *fairness*.

A Computer System as a Queueing Network

Beyond a single queue system, we can model a complete computer system, network, distributed system, architecture, and many other things as a *queueing network*, which you can think of as being analogous to an electrical circuit, where the components consist of queues, servers, and links between them. Here is an example of a queueing network for jobs called transactions running on a database computer that has one CPU and two disks:



Here, the idea is that when a job (transaction) arrives, it first goes to the CPU queue. After it has run for enough time on the CPU, it has 1/3 probability of finishing (exiting the system), 1/3 probability of moving to Disk 1's queue, and 1/3 probability of moving to Disk 2's queue. If it moves to a disk queue, it will eventually be serviced by the disk, and then move back to the CPU queue.

Similar to electrical circuit analysis, there are governing constraints (almost like Kirchoff's laws) that let us derive a system of equations from the queueing network. We can then solve this system of equations to produce analytical expressions for questions like "what is the expected time for a job (transaction) to complete in this system?" The equations and the answers are stochastic---this is a form of deriving and reasoning about the *probabilistic properties* of the system.

As noted earlier, analytic solvability depends on the state of the art in analyzing probabilistic systems and on the nature of the workload offered to the system. In many cases, although we can write the system of equations, we cannot solve it analytically. There is still plenty of value in thinking of the system using this kind of model, however. Even when it is not possible to solve the system of equations analytically, we can try to solve them *numerically*. Even when this is not feasible, we can simply use the queueing network as the basis for a *simulation* of the system that can give us numbers.[7]

Service / Scheduling Disciplines

In the above, we used the FCFS (or FIFO) service or scheduling discipline. There are a wide range of others to consider. What is being described here are theoretical models, which can be used for analysis, numeric solution, or simulation. Actual scheduler implementations in a kernel or elsewhere may approximate a theoretical model, but they are just that, approximations. There are a lot of devils in the details. Also, many commonly used schedulers, such as those in many production kernels, are not aligned with or inspired by a theoretical model at all.

An important thing to understand is that service or scheduling discipline generally has little effect when the load is low. It is when the load approaches 1.0 (including transient overload)

[7] Simulation is exactly what you will be doing in the Queueing Lab.

that you typically see serious differentiation in the performance, in terms such as mean response time, mean slowdown, fairness, and variation in these measures.

Shortest Job First (SJF) is the optimal nonpreemptive scheduling discipline for minimizing response time. SJF requires you know the job size (service time), which is often not possible.

Shortest Remaining Processing Time (SRPT) is the preemptive variant of SJF and has similar properties. In the "M/M" scheme above, SRPT and SJF can lead to starvation (job never runs), but under the workload characteristics more commonly seen in nature, they do not. Similar to SJF, SRPT requires knowing the job size, which is often not possible.

Fair Sojourn Protocol (FSP) arguably provides the optimality benefits of SRPT while having better fairness characteristics.

Processor Sharing (PS) is the ultimate in fairness (well, depending on the definition of fairness). The idea is that for any interval of time, not matter how small, the server is evenly split between all the jobs in the queue. If there are n jobs in the queue, each one makes continuous steady progress at a rate 1/n. This is what the practical "round-robin" scheduler approximates.

Generalized Processor Sharing (GPS) is Processor Sharing with weights or priorities. The idea here is each job *i* is assigned a weight w_i, and makes progress at a rate $\frac{w_i}{\sum_j w_j}$. This is what a practical scheduler with priorities, such as the Linux scheduler or a Lottery Scheduler typically attempts to approximate.

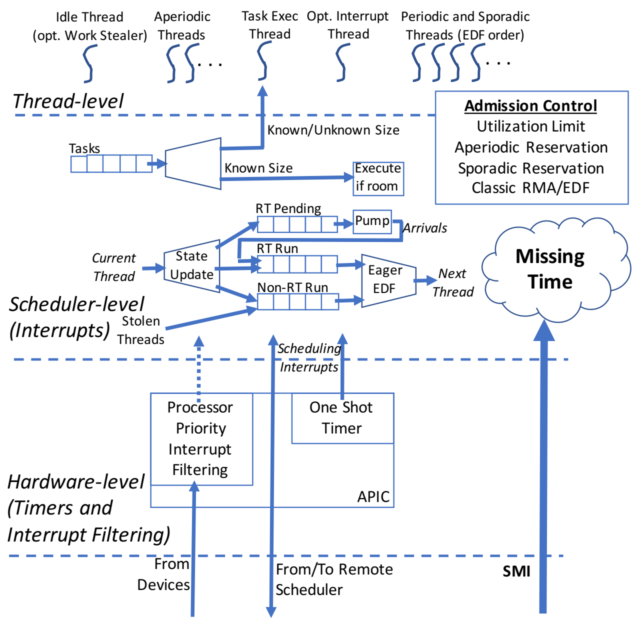
Dynamic Priority has the model that each job has a priority, and the scheduler runs the job that currently has the highest priority. The scheduler also changes job priority as it goes, based on various events. For example, if the job has been running for a long time, its priority might decline. Conversely, if a job has just arrived, or has just completed an I/O (such as a disk request completing or the user providing input), it might get a priority boost.

In real-time systems, very different scheduling models are used. In a real-time system, each job has a deadline by which it must be completed. When the job arrives at the system, it makes clear its size (service time) and its deadline. The scheduler then applies an *admission control algorithm* to determine whether it is feasible to meet the job's deadline while not missing any deadlines of already accepted jobs. If yes, the scheduler accepts the job and commits to meet the deadline. If no, the scheduler is allowed to reject the job.

To scheduler accepted jobs that include real-time jobs, a common scheduling model is the *Fixed Priority Scheduler*. Here, the idea is that each accepted job has a priority. The scheduler is committed to always be running the job that currently has the highest priority. When used in a real-time system, we commonly consider the deadline as the priority, with earlier deadlines having higher priority. When this is done, we typically refer to the scheduling discipline as *Earliest Deadline First (EDF)*, which is conveniently self-explanatory.

Scheduler Code in NK and Linux

If you'd like to look at the codebase of a relatively complex scheduler, you can take a look at include/nautilus/scheduler.h and src/nautilus/scheduler.c. The following complicated (sorry) diagram shows what the scheduler within a single CPU (hardware thread) looks like:



This scheduler schedules threads.[8] It is driven by events within threads such as calls to go to sleep or to yield to a different thread, and also by interrupts. The scheduler is invoked on every interrupt, but two particularly important sources of interrupts are local timer and companion schedulers running on other CPUs.

The basic scheduling discipline is EDF (Earliest Deadline First) --- this is a hard real-time scheduling model. The real-time threads are either sporadic (arrive once) or periodic (arrive periodically). Of all the runnable real-time threads, the one with the earliest deadline is run. If no real-time threads are runnable, the scheduler chooses a non-real-time thread (labeled "aperiodic"). This choice is made by a scheduling discipline decided when the kernel is built. Currently, the non-real-time scheduling disciplines that are available are round-robin (the default), two dynamic priority options, and lottery scheduling.

Linux's default scheduler is called the Completely Fair Scheduler (CFS). You can read more about it at https://en.wikipedia.org/wiki/Completely_Fair_Scheduler and the source code is in kernel/sched within the Linux source tree.

[8] This is a simplification. There are several other schedulable units in addition to threads, including tasks (shown on the figure), and fibers (not shown). The scheduler can also steer interrupts to particular CPUs and particular times.