Lecture 7 - Scheduling part 2

Gidon Rosalki

2025-05-11

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`
Be warned, this lecture will involve more mathematics than usual. Likely the most mathematical in the course. Gird thy loins, and prepare thyself.

1 Using accounting data

We ideally want to carry out, or approximate, SRPT (shortest remaining processing time), but we don't have the required information, only a very coarse grained estimation. To solve this, we will hedge will Round Robin, where we let every job run, and have insurance against the errors in the estimation.

The heavy tail distribution introduced last week has the following properties:

- The tail is heavier than any exponential $\left(e^{-kx}\right)$
- $\text{CV} = \dfrac{\sigma}{\mu}$ (coefficient of variation). Heavy tail usually has $\text{CV} > 1$
- $E\left(x - x_0 : x \geq x_0\right)$: The average remaining time for a job that survived x_0 seconds. A heavy tail implies that this is monotonously increasing in x_0

1.1 Distribution examples

1.1.1 Dirac

We have a generalised function δ, such that

$$\delta\left(x - a\right) = \begin{cases} 0, & \text{if } x \neq a \\ \infty, & \text{if } x = a \end{cases}$$

This has some interesting properties. Firstly, there is no tail, $E\left[\delta\right] = a$, and $\sigma = 0$. Additionally $\text{CV} = 0 < 1$. Finally

$$E\left[x - x_0 : x \geq x_0\right] = a - x_0$$

This is decreasing in x_0.
However, we know that this doesn't happen in reality.

1.1.2 Uniform distribution

Here we have the function

$$f\left(x\right) = \frac{1}{a} : 0 \leq x \leq a$$

This has no tail, the expectation of $\dfrac{a}{2}$, the standard deviation of $\dfrac{a}{\sqrt{12}}$, and the CV $\dfrac{1}{\sqrt{3}} < 1$. Additionally

$$E\left[x - x_0 : x \geq x_0\right] = \frac{a - x_0}{2}$$

This is also decreasing in x_0

1.1.3 Normal distribution

This has a light tail, and $E\left[x - x_0 : x \geq x_0\right]$ is also decreasing in x_0

1.1.4 Exponential distribution

This has the function

$$f(x) = \lambda e^{-\lambda x} : 0 \leq x < \infty$$

Where $\lambda > 0$ is the rate parameter, the number of events per unit of time. This has a "transition" tail, an expected value and standard deviation of $\frac{1}{\lambda}$, and a CV of 1. We additionally get that

$$E[x - x_0 : x \geq x_0] = \frac{1}{\lambda}$$

No matter how long you have waited, your expected additional wait time is $\frac{1}{\lambda}$.

1.1.5 Power law (Pareto) distribution

$$f(x) = ax^{-(a+1)} : a > 1 \wedge 1 \leq x < \infty$$

This has a heavy tail, CV $= \infty > 1$ (for $a \leq 2$), and

$$E[x - x_0 : x \geq x_0] = \frac{x_0}{a - 1}$$

1.2 Multi level feedback queues

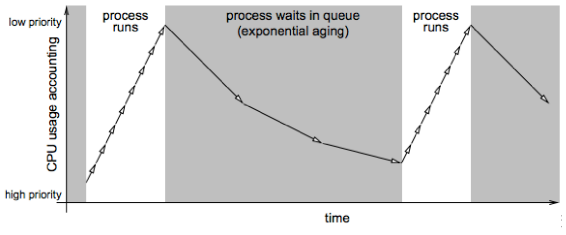
These distributions give us that if a job runs for a long time, it will most likely keep running for a long time, so if a job is preëmpted at the end of a quantum, then we learn something about it (assuming a heavy tail), in that its remaining time expectancy just got **longer**. We should thus give it a lower priority than new jobs, which are shorter on average. To help us implement, this let us consider multi-level feedback queues: When a job completes a quantum, do **not** return it to the run queue, but rather place it in a separate queue with other long jobs. We then schedule long jobs **only** if the original queue is empty. We can have multiple such queues, that can have different time quanta, and different scheduling algorithms. There are some rules for implementing a multi-level feedback queue:

1. New jobs enter the highest priority queue
2. Always schedule jobs from the highest priority non empty queue (jobs in lower priority queues do not run)
3. If a job completes its quantum, then it is demoted to a lower priority queue
4. Jobs in the same queue are scheduled with Round Robin.

To summarise, they manage to prioritise short jobs (like SRPT), without knowing in advance when additional jobs will arrive, or how new jobs will run, by using preëmption, and lightweight accumulated knowledge about jobs. Variants of this are still used by all major OSs.

1.3 Starvation

However, this leaves the problem of new short jobs regularly arriving, thus starving out the longer jobs. A possible method to resolve this, we give each queue a relative allocation of CPU time. The higher allocations go to higher priority queues (Which contain shorter jobs), and a non zero allocation should go to lower priority queues, that contain the longer jobs. Another possible solution is to use ageing: This has a negative feedback principle: running reduces your priority to run more, and thus moves you to a lower priority queue. In addition to this, waiting increases your priority to run, and moves you to a higher priority queue. The classic Unix scheduler worked as follows: There were 128 queues for 128 priority levels. 0 - 49 were for the kernel, and 50 - 127 were for user mode processes. It always scheduled the highest priority process (i.e., the lowest priority score), and for the kernel, priority was based on the reason for sleep, for example Disk I/O was given 20, Terminal I/O was given 28. This was all as a result of heuristics, there was not mathematics underlying these decisions. A simplified overview is as follows: User priority $\approx$ cpu_usage + base. On each clock tick, about 1/100th of a second, add 1 to the CPU usage of the running process (in the PCB). If a higher priority process exists, then switch to it. At the end of a burst (quantum = 10 ticks, or block), then switch to the next process of the same priority (RR). Every second (100 clock ticks), divide cpu_usage of **all** processes by 2 (increasing their priority), and recalculate priority.



An alternative view is to view the arrival of constant short jobs bot as a bug, but rather a feature! Short jobs arriving all the time implies that the system is overloaded, and that it is impossible to run

everything. We need to decide what not to run, and sacrificing long jobs makes sense, since they are not interactive, no one is waiting for them, and one long job completing is equivalent to many short jobs completing. Alternatively, they will run once the load abates (if it does, e.g. at night).

Returning to consider multi-level feedback queues, how does it treat computational vs interactive processes? Computational processes get lower priority, and simple interactive jobs (such as editors), will get higher priority, and will thus run immediately when ready. However, complex interactive jobs (e.g. 3D games) may get a low priority. Thus, modern schedulers try to prioritise, for examples, based off the active window.

2 Performance evaluation

3 Evaluating a scheduler

We may use queueing models, use queueing theory to solve a stochastic model, and derive average performance metrics. This is based on simplifying mathematical assumptions. We may also use simulations, use a program that implements a model of a computer system, however the model is a simplification of reality. Finally we have implementation, where we put the actual algorithm in a real system for evaluation, under real operating conditions. We will focus, for now, on the first option.

Let us suppose a system model, servers with queues. There is a queueing network, queues are shared between multiple interconnected servers. Let us analyse the FCFS scheduler. Given

- Assumption on the incoming job rate: Memoryless, with a fixed average rate of λ (exponential distribution)
- Assumption on the processing time, and order: Memoryless, fixed average rate μ (and an unlimited queue), using FCFS
- Goal: Find the average response time (we'll first find the average queue length using Markov chains)

A quick note on notation: We will use Kendall's notation. M/M/1:

- Arrivals: M = Memoryless
- Processing time: M = Memoryless
- Number of servers: 1

So, let us have an M/M/1 queue, a single server, with arrivals at rate λ, and service at rate μ. The stability constraint is that $\lambda \leq \mu$.

The exponential distribution is not really heavy tail, but it is borderline, and close enough for our purposes. We will use it for simplicity, and ease of analysis. Our goal is to find, given λ and μ, what will the average response time be?

3.1 Little's law

We know λ and μ, and we want to find $\overline{r}$ (the average response time). Little's law is as follows:

$$\overline{n} = \lambda \cdot \overline{r}$$

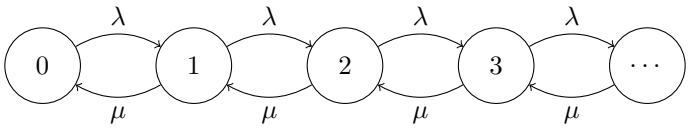
Where $\overline{n}$ is the number of processes in the system. However, we need to find $\overline{n}$ as a function of λ and μ. Consider the following example: A bar gets 6 visitors per hour (so $\lambda = 6$). Each visitor spends 3 hours at the bar, (so $\overline{r} = 3$). On average, how many chairs do you need?

$$\overline{n} = 6 \cdot 3 = 18$$

The system of M/M/1 can be in many different states:

- There may be no process at all
- There may be one process, running
- There may be 2 processes, one running, and one in the queue
- there may be 3 processes, one running, and two in the queue
- $\vdots$

The **state** is the total number of processes in the system, both running, *and* waiting. The system moves from state i to state $i+1$ at rate λ. The system moves from state $i+1$ to i at rate μ. This system is called a **Markov chain**.

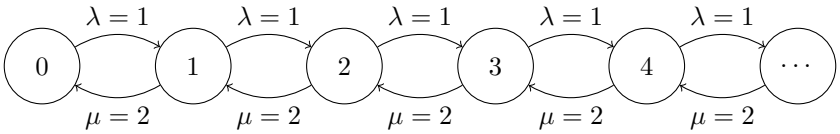


The states of Markov chains have limiting probabilities: Ergodic (connected, with no periodic cycles). We denote the probability to be in state i by π_i. This can be calculated using balanced flow (if reversible). The flow from state i to $i+1$ is the flow in the opposite direction.

$$\pi_0 \cdot \lambda = \pi_1 \cdot \mu \implies \pi_1 = \frac{\lambda}{\mu}\pi_0$$
$$\pi_1 \cdot \lambda = \pi_2 \cdot \mu \implies \pi_2 = \frac{\lambda}{\mu}\pi_1$$
$$= \left(\frac{\lambda}{\mu}\right)^2 \pi_0$$

3.1.1 Example

Let $\lambda = 1$ (1 job arrives per minute), and $\mu = 2$ (the CPU can process 2 jobs per minute). Therefore our Markov chain looks as follows:



Therefore, we get the overall expressions:

$$\pi_i \cdot \lambda = \pi_{i+1} \cdot \mu$$
$$\pi_i = 2 \cdot \pi_{i+1}$$

and we can thus derive

$$\pi_0 = \frac{1}{2}$$
$$\pi_1 = \frac{1}{4}$$
$$\pi_2 = \frac{1}{8}$$
$$\pi_3 = \frac{1}{16}$$
$$\pi_4 = \frac{1}{32}$$

3.1.2 M/M/1 state probabilities

Let us define $\rho = \left(\frac{\lambda}{\mu}\right)$ In general,

$$\pi_i = \left(\frac{\lambda}{\mu}\right)^i \pi_0 = \rho^i \pi_0$$

The sum total is

$$1 = \sum_{i=0}^{\infty} \pi_i = \pi_0 \sum_{i=0}^{\infty} \rho^i = \frac{\pi_0}{1-\rho}$$

Or

$$\pi_0 = 1 - \rho$$

So finally:

$$\pi_i = \rho^i (1-\rho)$$

Let us use the state probabilities to find the number of jobs:

$$\overline{n} = \sum_{i=0}^{\infty} i \cdot \pi_i = \sum_{i=0}^{\infty} i\,(1-\rho)\,\rho^i = \frac{\rho}{1-\rho}$$

Finally, use Little's law to find the average response time:

$$\overline{r} = \frac{\overline{n}}{\lambda} = \frac{\rho}{\lambda\,(1-\rho)} = \frac{\frac{1}{\mu}}{1-\rho}$$

The result:

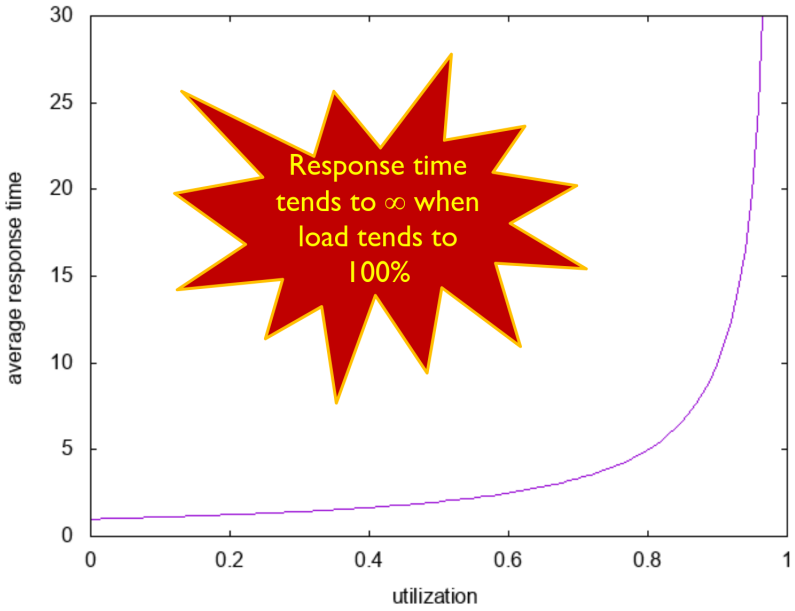


Figure 2: Average response time vs utilisation

The implications of this are that starvation is not binary, due to randomness, wait times are unbounded, and if we want a short response time, we have to accept utilisation $< 100\%$.

4 Open, closed, and combined systems

In an open system, we have no feedback from performance to jobs, where there is in closed, open has a fluctuating number of jobs, where they are constant in closed, and the load in an open system $< 100\%$, where load $= 100\%$ in closed, open has a response time metric, since this is the most important, but in closed we have a throughput metric. An example of an open system would be a web server, where for a closed system consider a controller.

In reality, models are often combined, users join, spend some time in the system, and then leave.

In a queueing network, performance is dictated by the bottleneck device. If jobs are compute bound, the CPU is the bottleneck, and the disk is mostly idle. If jobs are I/O bound, then the disk is the bottleneck, and the CPU is mostly idle. Only the scheduling of the bottleneck device is important.

5 Other scheduling contexts

5.1 Long term scheduling

Processes need memory space to run, but what if there is not enough for all of them? Then we need to swap out some of them, by storing them temporarily on the disk, and returning them alter. We must have the following considerations: Keep the interactive jobs, and create a good job mix (i.e., a complementary use of resources, minimising device bottlenecks by mixing CPU bound, and I/O bound tasks).

5.2 Fair share scheduling

It is important to note that "fair" does not necessarily mean "equal". Fair is according to allocation. We have a couple of methods to achieve this:

- Virtual time scheduling. Here we count the time ”faster” for low priority processes, which is implemented by a virtual time (”penalty”) per quantum
- Lottery scheduling: Give processes lottery tickets for various system resources, such as CPU time, and the number of lottery tickets reflects allocation. A lottery ticket is drawn at random, and the process holding the ticket gets the allocation.