

Section: Queuing Theory and File Systems

1 Announcements

- no admin this week..... good luck on your projects

Disclaimer: These notes aren't substitutes for attending lecture or reading the book. Those have been created with careful revision and review. These notes are mainly intended to be a more thorough index of the lectures and reading so that you can go off and read the full treatment on the topic and be able to ask the right questions while reading those two sources. If you have comments or suggestions please send me email and I'll fix them up.

2 Recap and Intro

Last week we mainly talked about different I/O systems and their properties that they could exhibit. This time around we are going to focus in on one very common I/O device, the disk. Instead of talking about cylinders and rotational latencies and such (which are also very important) we'll get much more into the theoretical aspects of disks such as how the data can logically be stored on it and found in an efficient manner (also known as the file system) and how long a disk actually takes to service a request. Now you may think that queuing theory is a very esoteric idea and not a practical application to our work, however, consider this. Your typical computer is now running at over 1GHz meaning at its best it can run an instruction every nanosecond (yeah, its fast). However the disks are still running in the order milliseconds to extract data. Thats millions of instructions worth of latency to hit the disk. So every tiny fraction we can save by understanding the disk translates to significant improvements in latency that the applications see and therefore its worth it. So in order to understand the usage of disks we'll need a bit of queuing theory to see what the average wait time is to service a request if multiple processes are accessing the disk at one time.

3 Queuing Theory

For the purposes of this class the main goal of the queuing theory is to understand how long the average request will wait in a queue given a relatively simple model of entry and service time. The tools that we will derive here are mainly useful for back of the envelope calculations that give you first order estimates of how a system will behave. One very important thing to note is that our queueing models the arrival rate is equal to the departure rate. If you look at the system for T seconds, then the number of tasks that enter the system is exactly equal to the number that leave the system. However this doesn't say anything about how long the queues are it just analyzes snapshots of the system.

3.1 Little's Law

Given the statement above we can prove a few things about the expected length of the queue. Little's Law is given in (1)

$$\text{mean number of tasks in the system} = \text{arrival rate} \times \text{mean response time} \quad (1)$$

We can also write this as

$$L_q = \lambda \times T_q$$

Where L_q is the Length of the queue λ is the arrival rate and T_q is the time in the queue.

Now this might seem a bit weird so let me give a couple of examples to motivate the idea. There are tons of books that can do a lot better at a formal proof the idea than I can so i will refer you to those books if you want the formal explanations. I'll just give an intuitive explanation of why its correct so you can read the proof with a bit more intuition under your belt. There two main terms on the right hand side. The first is *the arrival rate* which is defined as the number of tasks that show up in a given span of time. The second is *mean response time* which is the average time that a task spends in the queue. Remember when trying to construct examples in your head that the arrival rate must equal the departure rate otherwise these simple models start to break down.

Lets start of with a simple example. Lets say you are in line for buying movie tickets for a popular new release that takes cash only so that all you do is give the money and get the tickets. Thus the service time is fixed. Since the arrival rate is the same as the departure rate by the time you get to the front of the queue the average length of the queue will not have changed. Then you can take a look at how long you have spent in the queue (lets say 20 min) and then realize that people are arriving every thirty seconds. So during the 20 min that you spend in the queue (i.e. the mean response time) there have been 40 people that have arrived since they arrive exactly every 30 seconds. Since the rates are constant we can reason that the total number of tasks in the system has to be 40.

Exercise: *Now lets try another more interesting example. In an amusement park a given ride services 20 people instantly (yes its a made up example), however because of setup considerations it can only be run once every 10 minutes. Thus the first person in the queue has to wait 10 minutes in the system (wait time), however the last person that arrives walks right on and therefore his response time is 0 minutes. If a person arrives every 30 seconds, what is the length of the queue?*

(Hint:) What is the average response time?

3.2 Time spent by the server

Another important consideration in queueing theory is analyzing how much time the server spends with a particular task since it is variable. Systems in which the server spends a constant time with the tasks are not very interesting since this is not a realistic model of the world (as anyone who has ever waited in line to by groceries knows). Lets first only deal with a discrete set of times that a server can spend with a task (extending these ideas to the continuous case is interesting, but the discrete case is enough to give us a good intro

into the material). Let $p(T)$ be the probability that the server spends time T with any given task then the average time that a server spends with a task T_{serv} can be written as:

$$T_{serv} = \sum_T (p(T) \times T)$$

We can also calculate a variance on response time as follows:

$$\sigma^2 = \sum_T (p(T) \times (T)^2) - T_{serv}^2$$

So far we haven't done anything interesting and are calculating simple statistics on the servers response time. However define a new term

$$C = \frac{\sigma^2}{m1^2}$$

In this context C is a what is called the squared coefficient of the variance. It is a measure of how dispersed the data is compared to the mean. If C is 0 then we know that that there is no standard deviation and thus like our first example the service time is deterministic. However if C is 1 then we know that the standard deviation is equal to the mean and we call this "memoryless." If the standard deviation equals the mean then we know that we can draw very little conclusions about how long the "typical" task is going to take since there are no such things as "typical" tasks and thus the system is considered memoryless. Again good statistics books will have a much more rigorous definition of these concepts. For our purposes imagine that C is either 0 (a deterministic system) or C is 1 (a memoryless system).

3.2.1 Mean Residual Wait time

When a task enters a system another task is getting served by the system. The *mean residual wait time* gives you a formula on how to estimate the amount of time the current task that is getting served will be in the system has left. The formula given (again look at stats books for a formal proof) is

$$m1(z) = \frac{1}{2} \times T_{serv} \times (1 + C)$$

If we drop in at an arbitrary time, then on average half the task is left if the system is deterministic. However if the system is memoryless then we have no idea how much of the task is left and we have to guess that the entire task is left which is the result when C is 1. The $(1+C)$ captures the variance in the service times.

3.3 Variables

In this section I just list the variables that will be useful in our queueing theory examples.

- λ : arrival rate
- T_{serv} : average time taken to serve a task

- C : squared coefficient of variance
- μ : service rate ($\frac{1}{T_{serv}}$)
- u : server utilization ($\frac{\lambda}{\mu} = \lambda \times T_{serv}$)
- T_q : Time spent in the queue
- L_q : Length of the queue

3.4 Derivation of M/G/1 Queues

The amount of time we wait in the queue is equal to

T_q = time to service other tasks in front of you in the queue + time left for current task being served

$$T_q = L_q \times T_{serv} + \text{mean residual wait time} \times \text{probability that something is at the server}$$

We then substitute Little's law for L_q and the variables above to get

$$T_q = \lambda \times T_q \times T_{serv} + u \times m1(z)$$

We can then use the definition of utilization

$$T_q = T_q \times u + u \times T_{serv} \times \frac{1}{2} \times (1 + C)$$

Rearranging we get

$$T_q = T_{serv} \times \frac{1}{2} \times (1 + C) \times \frac{u}{1 - u}$$

Notice that this says that designing a system for 100% utilization is a very bad idea since the time in the queue blows up to infinity even if the arrival rate and departure rates are constant!

Exercise: *The express line in a grocery store is usually reserved for 15 items or less, however the lines always seem much longer than normal the other lanes. Lets go through an example where we calculate how the wait times differ. Lets say that in the express lane the arrival rate (λ) is 1 per 3 minutes and the average service time is 2 minutes with a variance of 30s. What is the expected time in the queue for the fast lane? In the slow lane people arrive once every 8 minutes and the average service time is 4 minutes with a variance of 1 minute. What is the expected time in the queue for the slow lane?*

(Hint:) calculate all the intermediate variables and use the formulas ... its mainly plug and chug

4 File Systems

Fundamentally a disk or any I/O device is block addressed, that is you give it an address and it fetches the block out of that address. These blocks are fixed size entities, however most of the files we store on hard drives are not the same size. They vary a lot in their size and how you would normally access the data within them. Many research studies have shown that most files on a typical disk tend to be small however there are a few large files. The main goal of a file system is to convert this clunky block interface into the simplicity of named files and directories that we are all used to such that we can efficiently access the files on the disk. The design of a file system is actually a very critical performance aspect of any modern operating system since a file system with heavy seeking will exhibit very poor performance. In this section we will take a look at a few file systems and their characteristics.

4.1 Contiguous Allocation

The simplest of the allocations. In order to create a file, it first finds a run of blocks such that the entire file will fit and just allocates that block to that file. In some index we then store the start and length of each file. The idea is very simple but it has many drawbacks.

Question: What are the pros and cons of contiguous allocation?

Question: What applications is this file system good for?

4.2 Linked Allocation

In this scheme the files are arranged in a linked-list like fashion. The directory contains the start of the files and not the length. Then each block contains a pointer to the next block.

Question: This idea solves the problem of growing files easily, but what problem does it create?

Question: How does one do random access in this case?

Question: What applications is this file system good for?

4.3 File Allocation Tables

The next idea that we saw in lecture is the idea of using a file allocation table (FAT). In this scheme there is a table that has a massive index of files and their location. It is similar in spirit to the linked allocation except that we store more information about the where the links are in the file allocation table and therefore improving random access performance. However the problem is that the indices are stored in the table so there are still too many seeks for sequential access.

4.4 Indexed Files

Like before each file is stored in a directory and has a pointer to the block that contains its index. The index then has pointers to the actual data blocks. This is clean in that random access is very fast and simple but sequential access could still require lots of seeks.

Question: What are the pros and cons to the indexed files?

Question: What applications is this good for?

4.5 Multi-level Indexing

Same as the multi-level page tables we have the index block point to other index blocks which lead us to the data. It allows us to grow the file sizes to order of magnitudes larger than what one-level index files allow, however we have added extra overhead.

4.6 Combination, BSD 4.2

In the BSD 4. 2, the files are broken up into indexes called inodes. An inode is a representation for a file which contains pointers to the block data. To allow the file to grow, the last three pointers are as follows: the third to last is a singly indirect block so that the block it points to is an index of more blocks, the second to last is a doubly indirect block so it points to indices which themselves point to indices, and the last index points to a triply indirect block (yet another level). This scheme allows the file sizes to grow very large while maintaining low overhead for small files, however for large file, sequential access still hasn't been fixed and fragmenting could still be a problem.

Question: What are the other pros and cons of this approach?

Exercise: *If we had a file that was 10kB how much actual storage on the disk with 4kB blocks would that take up (assume that each entry takes 4 bytes and we have room for 15 slots for block pointers)? 40kB? 100kB? 1000kB? 1MB? 1GB?*

4.7 Cray DEMOS

I think here i will defer you to Kubi's lecture notes on the topic since he went over them well. Think of this approach as having multiple segments and thus does well with long runs of sequential data. However the disadvantage is that finding free pages and contiguous pages becomes more and more difficult as the disk gets fuller.

Question: What types of applications would be well suited to this File System?

Question: Why would a supercomputer company design this file system?

Exercise: *Lets say you wanted to design a file system for a media server (i.e. a server that reads big movie files off the hard drives and pipes them out over the network card). How would you do it? What kinds of design considerations do you have, what kinds of limitations exist? Is it easier or harder than constructing a file system that is good at random access?*