Dan Slimmon

SRE at Hashicorp

The most important thing to understand about queues

ON 2016/08/262022/03/09 / BY DAN SLIMMON / IN STATISTICS AND PROBABILITY

You only need to learn a little bit of queueing theory before you start getting that ecstatic "everything is connected!" high that good math always evokes. So many damn things follow the same set of abstract rules. Queueing theory lets you reason effectively about an enormous class of diverse systems, all with a tiny number of theorems.

I want to share with you the most important fundamental fact I have learned about queues. It's counterintuitive, but once you understand it, you'll have deeper insight into the behavior not just of CPUs and database thread pools, but also grocery store checkout lines, ticket queues, highways – really just a mind-blowing collection of systems.

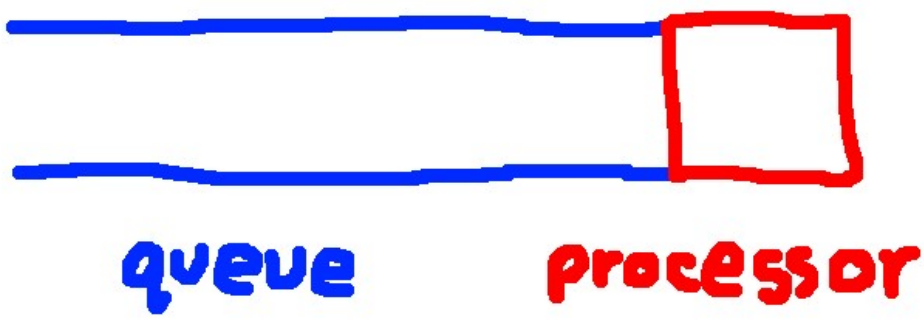
Okay. Here's the Important Thing:

As you approach maximum throughput, average queue size – and therefore average wait time – approaches infinity.

"What???" you say? Let's break it down.

Queues

A queueing system is exactly what you think it is: a processor that processes tasks whenever they're available, and a queue that holds tasks until the processor is ready to process them. When a task is processed, it leaves the system.



A queueing system with a single queue and a single processor. Tasks (yellow circles) arrive at random intervals and take different amounts of time to process.

The Important Thing I stated above applies to a specific type of queue called an M/M/1/∞ queue (https://en.wikipedia.org/wiki/M/M/1_queue). That blob of letters and numbers and symbols is a description in Kendall's notation (https://en.wikipedia.org/wiki/Kendall%27s_notation). I won't go into the M/M/1 part, as it doesn't really end up affecting the Important Thing I'm telling you about. What *does* matter, however, is that '∞'.

The '∞' in "M/M/1/∞" means that the queue is **unbounded**. There's no limit to the number of tasks that can be waiting in the queue at any given time.

Like most infinities, this doesn't reflect any real-world situation. It would be like a grocery store with infinite waiting space, or an I/O scheduler that can queue infinitely many operations. But really, all we're saying with this '∞' is that the system under study never hits its maximum occupancy.

Capacity

The processor (or processors) of a queueing system have a given total **capacity**. Capacity is the number of tasks per unit time that can be processed. For an I/O scheduler, the capacity might be measured in IOPS (I/O operations per second). For a ticket queue, it might be measured in tickets per sprint.

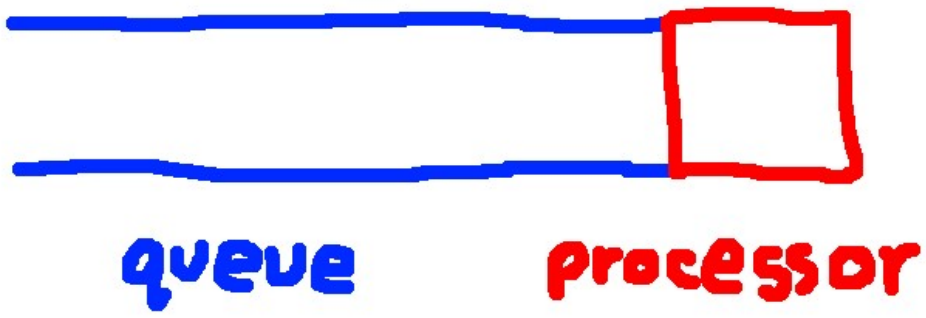
Consider a simple queueing system with a single processor, like the one depicted in the GIF above. If, on average, 50% of the system's capacity is in use, that means that the processor is working on a task 50% of the time. At 0% utilization, no tasks are ever processed. At 100% utilization, the processor is *always* busy.

Let's think about 100% utilization a bit more. Imagine a server with all 4 of its CPUs constantly crunching numbers, or a dev team that's working on high-priority tickets every hour of the day. Or a devOp who's so busy putting out fires that he never has time to blog.

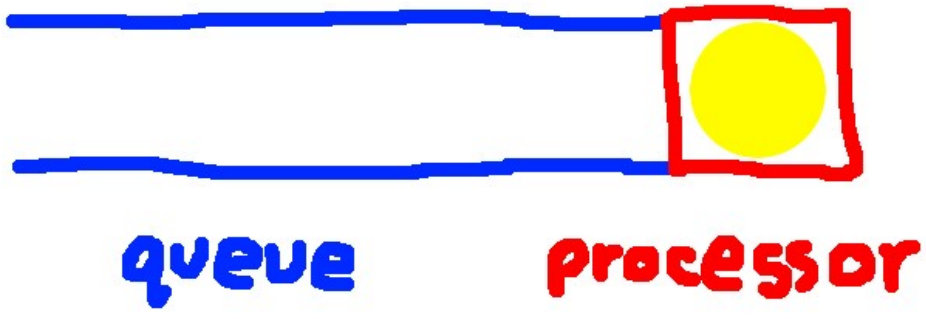
When a system has steady-state 100% utilization, its queue will grow to infinity. It might shrink over short time scales, but over a sufficiently long time it will grow without bound.

Why is that?

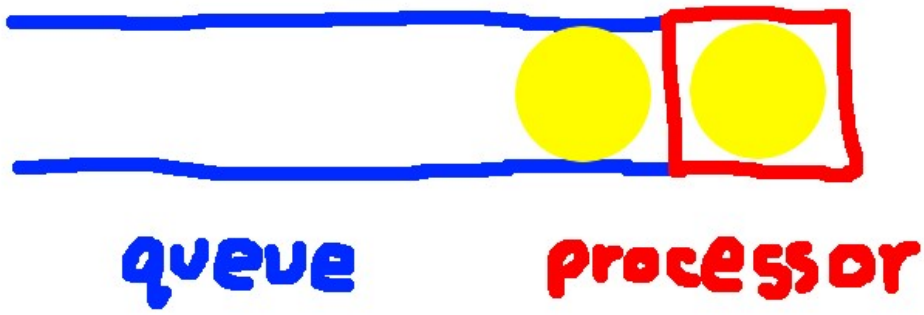
If the processor is always busy, that means there's never even an *instant* when a newly arriving task can be assigned immediately to the processor. That means that, whenever a task finishes processing, there must always be a task waiting in the queue; otherwise the processor would have an idle moment. And by induction, we can show that a system that's forever at 100% utilization will exceed any finite queue size you care to pick:



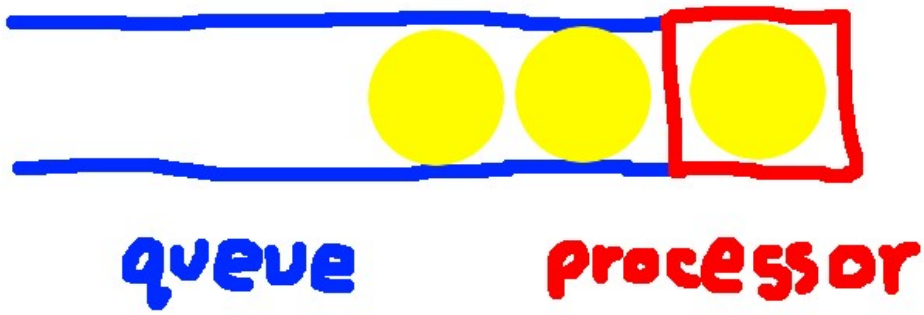
No good. Processor idle, so utilization can't be 100%.



No good. As soon as the task finishes, we're back to the above picture.



No good. As soon as the current task finishes, we're back to the above picture, which we've already decided is no good.



You get the idea.

So basically, the statements "average capacity utilization is 100%" and "the queue size is growing without bound" are equivalent. And remember: by Little's Law (https://en.wikipedia.org/wiki/Little%27s_law), we know that average service time is directly proportional to queue size. That means that a queueing system with 100% average capacity utilization will have wait times that grow without bound.

If all the developers on a team are constantly working on critical tickets, then the time it takes to complete tickets in the queue will approach infinity. If all the CPUs on a server are constantly grinding, load average will climb and climb. And so on.

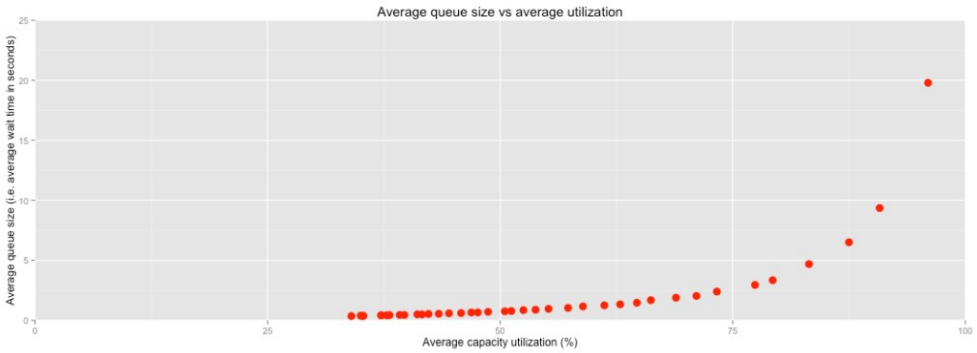
Now you may be saying to yourself, "100% capacity utilization is a purely theoretical construct. There's always some amount of idle capacity, so queues will never grow toward infinity forever." And you're right. But you might be surprised at how quickly things start to look ugly as you get closer to maximum throughput.

What about 90%?

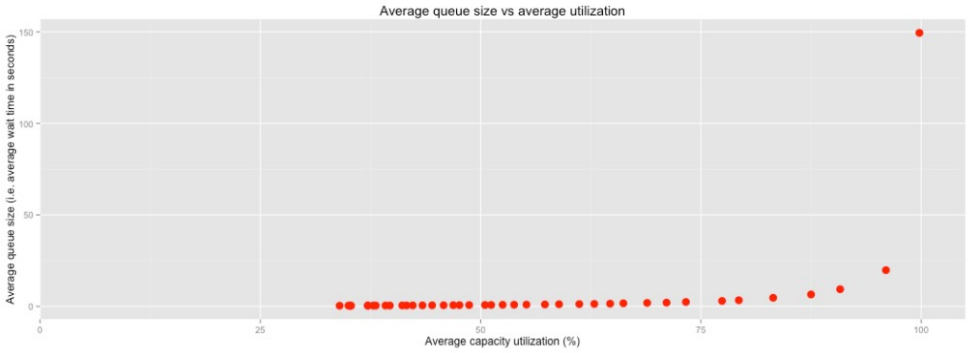
The problems start well before you get to 100% utilization. Why?

Sometimes a bunch of tasks will just happen to show up at the same time. These tasks will get queued up. If there's not much capacity available to crunch through that backlog, then it'll probably still be around when the next random bunch of tasks show up. The queue will just keep getting longer until there happens to be a long enough period of low activity to clear it out.

I ran a qsim (https://github.com/danslimmon/qsim) simulation to illustrate this point. It simulates a single queue that can process, on average, one task per second. I ran the simulation for a range of arrival rates to get different average capacity usages. (For the wonks: I used a Poisson arrival process and exponentially distributed processing times). Here's the plot:



Notice how things start to get a little crazy around 80% utilization. By the time you reach 96% utilization (the rightmost point on the plot), average wait times are already 20 seconds. And what if you go further and run at 99.8% utilization?



Check out that Y axis. Crazy. You don't want any part of that.

This is why overworked engineering teams get into death spirals (https://danslimmon.wordpress.com/2015/07/09/when-efficiency-hurts-more-than-it-helps/). It's also why, when you finally log in to

that server that's been slow as a dog, you'll sometimes find that it has a load average of like 500. Queues are everywhere.

What can be done about it

To reiterate:

As you approach maximum throughput, average queue size – and therefore average wait time – approaches infinity.

What's the solution, then? We only have so many variables to work with in a queueing system. Any solution for the exploding-queue-size problem is going to involve some combination of these three approaches:

1. Increase capacity
2. Decrease demand
3. Set an upper bound for the queue size

I like number 3. It forces you to acknowledge the fact that queue size is *always* finite, and to think through the failure modes that a full queue will engender.

Stay tuned for a forthcoming blog post in which I'll examine some of the common methods for dealing with this property of queueing systems. Until then, I hope I've helped you understand a very important truth about many of the systems you encounter on a daily basis.

14 thoughts on "The most important thing to understand about queues"

1. Pingback: The Latency/Throughput Tradeoff: Why Fast Services Are Slow And Vice Versa – Dan Slimmon

2. Pingback: Latency- and Throughput-Optimized Clusters Under Load – Dan Slimmon

3. **Antoine**
   I am not certain of how rigourous is this paragraph:
   "If the processor is always busy, that means there's never even an instant when a newly arriving task can be assigned immediately to the processor. That means that, whenever a task finishes processing, there must always be a task waiting in the queue; otherwise the processor would have an idle moment. And by

induction, we can show that a system that's forever at 100% utilization will exceed any finite queue size you care to pick:"
if the queue processes x elements per second, and there is initially 1 element in the queue plus x elements per second, it seems to me that the queue won't be infinite, but the processor will still have a 100% utilization.

2021/01/08 AT 23:01 ↪ REPLY

Dan Slimmon

You're correct that it's not stated rigorously. What I mean by "will exceed any finite queue size you care to pick" is that the queue size will increase without bound. Given enough time, it will exceed any arbitrary size.

2021/03/14 AT 23:03 ↪ REPLY

1. **Arthur**

   it is not only lacking rigour, it is simply wrong. What prevents the system from boucing between pictures 3 and 4, while mantaining 100% utilization? the author rushed away from this possibility to "prove" his point

2021/04/23 AT 12:26 ↪ REPLY

Dan Slimmon

> What prevents the system from boucing between pictures 3 and 4, while mantaining 100% utilization?

What's to prevent the system from bouncing between "1 task in processor & 1 task in queue" and "1 task in processor & 2 tasks in queue" while maintaining 100% utilization?

Nothing! That could totally happen in a queueing system. However, the arrival process would need to be tuned quite precisely to the processing rate. You would need to watch the status of the processor and, when a task finishes, only then insert a new task into the queue. But this implies a shell game: if you always have a task ready to put into the queue when it needs to be padded back out, then isn't that task in a sense already "queued"?

Instead, in queueing theory we usually assume a random arrival process: sometimes a minute may pass between arrivals into the queue; sometimes only a second. So the system can't bounce for arbitrarily long between 1-in-queue and 2-in-queue states. Eventually, one of two things will randomly occur:

1. From the 1-in-queue state, the active task finishes processing before a new task arrives, bringing queue size to 0.
2. From the 2-in-queue state, a new task arrives in the queue before the active task finishes, causing the queue size to grow to 3.

2021/04/23 AT 13:45 ↪ REPLY

1. **Arthur**
   > So the system can't bounce for arbitrarily long between 1-in-queue and 2-in-queue states

   Now that addresses the unbounded size. The example of the article excluded this possibility of bouncing between two states for an extended time, and seemed to exchange "unbounded size" for "bounded to grow", which is totally not the case.

   2021/04/23 AT 17:00 ↪ REPLY

   1. **Arthur**
      bound to grow*

      2021/04/23 AT 17:02 ↪ REPLY

4. Pingback: The most important thing to understand about queues (2016) - The web development company Lzo Media - Senior Backend Developer

5. **Graham**
   I like the analogy of an overflowing sink

   https://ferd.ca/queues-don-t-fix-overload.html

   While backpressure is important, It's also useful to understand what is in the queue. Some tasks don't mind if they are completed an hour late, whereas others might as well be discarded if queued for too long (e.g. if the client has already timed out)

   2022/03/08 AT 13:40 ↪ REPLY

6. **Dan**
   some about of idle capacity -> some amout of idle capacity

   2022/03/08 AT 13:44 ↪ REPLY
   Dan Slimmon
   Fixed. Thank you!

   2022/03/08 AT 20:33 ↪ REPLY

7. Pingback: The most important thing to understand about queues - blog.wuyuansheng.com

8. Pingback: The most important thing to understand about queues (2016) | Movilgadget