Multi-Threaded Network Programming and the Linux Kernel: A Personal Illustration/Demonstration of My Relevant Foundation of Knowledge, Working Approach and Shipping Experience

by Mike Kramlich

2017 June 17

I'll do this in two parts.

First I'll give a talk which illustrates my understanding of this area in general, and the fundamentals, the core goals and reasons why anyone might do either multi-threaded programming, or network programming, in the first place. My understanding of the forces which shape the solution space and therefore the tradeoffs involved.

Second I'll give some example projects and work challenges where I've applied my understanding in this area to do something useful. To solve some actual problem in the real world, and ship.

However as I sketched out in my mind a plan for answering this, especially the first part, I realized that I couldn't escape talking about operating systems and their relation to hardware. In particular, UNIX-like OS's, especially Linux. Since that's what I have the most experience with in recent years. Therefore, I'm going to "cheat" by also making this answer serve a dual purpose of also giving a taste of my knowledge of the Linux kernel, ecosystem and modern high-traffic e-commerce websites on the Internet.

First part... Imagine a simple computer.

Say a microcontroller.

It has one brain (one CPU, one core), a small amount of memory, and various physical interface points (eg. connectors, ports). When electricity is applied this brain comes alive and begins executing whatever instructions are in its memory (which it finds encoded there, mathematically, and in binary form.)

To make it do something useful you write a program, deploy it to this MC then reboot the card. It executes what it sees in memory, which is your program. Your program is now running and it is the sole program running, the only application and it is the master of all it surveys, and doesn't have to share access to any of its resources.

However... a day comes when you've incrementally added so many additional features to your program that it's becoming unwieldy. Most importantly, instead of all these features (these individual subtasks, or flows) needing to run all the time, or

execute all the time, what becomes clear is that many only need to execute in response to certain events. Like a certain time of day, or when a particular signal value is detected at one of the physical interfaces (perhaps a sensor, or a communication port.) Therefore it would be nice if we could re-architect our program so it becomes chopped up into many distinct pieces, each with its own separate control flow and decision logic, and each piece could somehow be scheduled to only run when that piece was truly needed. This would have the added benefit of potentially using the shared resources more efficiently -- for example we could maybe come up with some way to more evenly distribute the work load across time, at least for those tasks which didn't care too precisely about when they ran. So you learn about the existence of a so-called operating system, or OS. You re-design your solution to take advantage of an existing third-party OS, and probably switch hardware models as well (because typically the overhead costs of having an OS result in needing a little beefier hardware, plus an OS itself makes it easier to take advantage of yet beefier hardware.

Your new program runs now on an x86 PC-like SBC and you're using Linux, which belongs to the class of OS's called multi-tasking, protected memory systems, which means its designed both to allow multiple distinct programs to run, concurrently, while also ensuring they can only access their own personal chunks of memory, and otherwise won't trample on each others toes. However, your board has only 1 CPU and 1 core. Therefore only 1 program can ever truly be running at once, inside that CPU, by that core. Linux will gladly orchestrate and coordinate the running of 100's of distinct programs for its users, making sure they each get a relatively even and therefore fair share of CPU execution time (speaking in general; skip talk about priority levels, interrupts, etc.). However, you only have 1 core. So the more and more programs you add to this board's workload, the slower each one individual will seem to run (speaking in general.) So you learn quickly to keep that number as small as possible, in order to give the most CPU execution focus to your program, your payload program.

And then... another realization hits you. You analyze your code and realize that while certain parts of the flow must be serial (because they have an order-of-execution dependency among their component steps), not all do. Some parts of your flow could be done in... parallel. In fact, many sections of your code could easily be rewritten to execute in parallel. (For example, imagine your program's goal was to count votes from ballots in an election. If you had 100M ballots to process, and it took 1 second to process each ballot, and you could only process 1 ballot at a time, serially, then, it would take 100M seconds for you to finish counting all the votes and decide the winner. However, ballot counting is inherently parallelizable. So if you could delegate to say 100 different processors, each given an equal slice of the total ballots, and have them all work in parallel, then, once the last one was done, count up all the subtotals to arrive at overall total, etc. you'd finish the overall count task 100 times faster than otherwise. Have 1M processors working in parallel, you finish 1M times faster: the entire election in 100 seconds. This should sell you on the benefit of parallelization. We'll see how it applies to threads and networking in a minute.)

So... you decide to rewrite your program so that all those sub-flows which could be parallelized, are. But wait... how? This whole time we've been thinking of our program as a process. A single process, with a single shared region of working memory, and a single thread of execution from the perspective of both the CPU and the OS

process scheduler. However, if you think about it, all those things don't need to go together, not always. They are a bundle of different features, but its possible to break them out, keeping some of them but leaving out the rest.

Enter the notion of a thread. A thread and a process are very similar. With some important differences. By default, and at least in Linux, when a program runs, it becomes a process, with a single thread of execution (essentially just a pointer and a little chunk of memory; a relative address within a program's instruction sequence, a so-called "stack" structure, and the state of values in CPU core's registers which are also associated with that same moment in execution time). However, and again at least in Linux, it's possible for a program to have multiple distinct threads of execution --- all while sharing the same shared region of working memory. Speaking of sharing memory, technically each thread has 2 kinds of memory state which are not shared, which are unique to them: their register values, and their stack. The stack is a portion of memory designed to allow the system to keep a kind of historical log of a program's execution, to support cases where generic subroutines are called, and especially to allow recursion. The stack allows you to have a nested graph of function calls, and to allow a CPU core to know where to jump "back" to when an instruction subroutine has ended or otherwise wishes to exit. To jump back in a clean way, and restore the state of registers. As a bonus, the stack structure was also used to allow programs to pass parameters in function calls, by "pushing" them onto the stack, before the call is carried out, then "popping" them off later once no longer needed. Additionally, in many (but not necessarily all) programming languages, the stack structure is also where so-called local variables are stored -- variables which come to life while the flow is inside a subroutine body, and cease to exist when the body exits. Therefore, local variables become another kind of private memory which is only accessible by the currently executing thread, essentially its owner. Since all threads of a process share the same main region of working memory, it enables both good and bad things -- but I'll talk about those later.

Right now the key point to threads and why we're considering them in our example system is, we realized we had opportunities to parallelize parts of our program, and it turns out that threads are one good way of doing that. So... we decide to use threads. To have multiple threads. We rewrite the portions of our code which had obvious ways to parallelize (eg. our ballot counting tasks), and we make them use multiple threads instead. We divide up the overall work into sub-tasks, each with a subset of the overall input data, start some number of additional threads (ie. child worker threads), handing each the appropriate parameters or (pointers to) their assigned data subsets. When all the threads finish, we evaluate all their sub-results, figure out how to combine them up (for ballot counting: pretty obvious), and move on. Our code is perfect. We run for first time.

And... it is not any faster.

In fact, it appears to be slower, for some bizarre reason.

There was a problem.

Remember we said we only had 1 CPU and 1 core? Well this meant that our hardware can only ever execute one program at a time (1 process, 1 thread). Linux will do its darnedest to schedule execution slices evenly across all the the various programs (processes, threads) competing for CPU execution time. However, if there is only 1 core

total available, then if someone adds more threads to their program, it won't finish any faster (for now: assume speaking in broad strokes, and a perfectly compute-bound loads. we'll circle back on this point later.) Therefore... now you realize that the only way you can truly speed up your program via parallelization (well, this ballot counting work, and assuming perfectly compute bound) is if we have more cores. We clearly need more cores or more CPUs.

So we upgrade.

We migrate our program to a new machine that has 4 CPUs, with (effectively) 4 cores per CPU. Therefore it has 16 cores and can physically execute 16 process threads concurrently. In theory. Still Linux, and everything else is the same. We tweak our code perhaps, then re-deploy, run, test... Bingo. Much faster. But now we have a new realization. No matter how much we parallelize this program, we'll never be able to do more than N tasks concurrently, where N is the number of cores (again, still assume compute bound). So the only way to make our ballot counting program run any faster is to re-architect to take advantage of multiple computers. Several distinct computing nodes or servers. This would also let us "scale up" nicely if our total workload increases. If we could redesign to use 2 servers we could either finish our work in 1/2 the time, or, finish 2x the work in the same time (in broad strokes.) Likewise for 10 servers, or 100, etc. But to do that we need a way to communicate between multiple servers, or otherwise coordinate them.

Enter the notion of networks, communication protocols, and distributed computing.

At this point I'm going to wave my hands and skip over doing a deeper dive on these topics, but I hope I've given you enough sense by now that I can go deeper on those topics as well if needed. For my purpose now, I'm just going to say that in order to continue evolving our example system we're going to design it to be a distributed application, with a client/server architecture, one master control program, and N number of "child worker" nodes. This lets us not only parallelize elegantly every possible sub-flow we find, that's amenable to being parallelized, it also lets us "horizontally" scale up our deployed resources to perfectly match how the total workload grows. In theory (again, in broad strokes) this ensures we have no ceiling. (We will, in practice, but that's for another discussion.) And... we'll keep the multi-threading. But now we'll also have to establish connections to the other servers in the cluster, and communicate with them over the network. And now we have a bunch of other issues to fight and design mitigations around.

For example, in a distributed system, sometimes the other server you're trying to talk to... just isn't there. Or they're there but your IP packets didn't reach them -- perhaps they got dropped by some device in the middle, perhaps a router failed, or there was too much traffic and your packets were de-prioritized, etc. Or your packets reached the destination server's network interface but.... the program there which was supposed to handle it just wasn't running. Or it was running but it's in a hosed state (a zombie, the walking dead). Or it too is multi-threaded (perhaps using a capped pool) and was simply overloaded. Maybe that server itself is not overloaded, not broken, no bugs, but itself is latency dependent on yet another server behind the scenes (perhaps some database it needed to query in order to respond correctly to your own request of it, and that database query is taking a long time to finish and respond.) Perhaps you're

experiencing a MITM attack -- the server you're talking to is not the one you thought you were, it's a "bad guy" controlled system. Etc, etc. I'm essentially rehashing what's called the Fallacies of Distributed Computing. The takeaway is that the moment you switch to a distributed architecture, and do network programming, while they do solve certain problems, it also introduces a set of additional ones.

So, for example.... in our new multi-threaded distributed solution we have to take into account the various ways that communication between the nodes can fail. We have to think much more carefully about state and synchronization. About latency. Throughput. Redundancy. Security. But we are now at least taking advantage of every core on a machine, and we can have N machines working concurrently. But there are some additional problems we haven't talked about yet, dealing with threads.

Returning to threading.... Say on a single standard server we again have 4 CPUs, 4 cores each, thus 16 total cores. Our program previously was perfectly compute-bound. Pure counting, pure number crunching and bit shifting, twiddling memory values, etc. But this is a rare case. Most "real world" software is not perfectly compute bound. Particularly anything with a human-facing UI. Or if it needs to do lots of data IO, like networking or talking to a filesystem. The good news about that is that in most generic modern OS's, like Linux, when a program does not have anything to compute at the moment (because it's waiting for new input data to arrive for reading or its waiting to finish writing to a filesystem or across a socket, eg., or waiting on some other event to happen or signal to fire) the OS will be "smart" and not give it to a CPU to execute. The Linux process scheduler, for one, will essentially skip over any so-called non-runnable processes when it comes to making its execution "load balancing" decision. A process might be "running" in the human sense, in that it was started and has not exited yet. But if it is not currently ready and able to execute its next instruction, because its waiting for something, then, it is not in a "runnable state" from the perspective of Linux. The upshot of this is that on a typical server it's not uncommon to have 100's of processes alive at once, even though there may only be 16 cores or 8 cores, etc. And yet everything runs fast enough, with no complaints. Because many of those processes are IO-bound, or otherwise waiting for some necessary event to happen and trigger their reaction. And Linux/UNIX is smart enough to skip over them in the meanwhile. What this means for your program, as an architect, is that it might be perfectly fine to design your app to have many more threads than the number of cores available on the host machine. If you have a sufficient amount of IO-bound sub-flows.

Now... how does this apply in our case? Well before we talked about how we did ballot counting, it was pure compute, great, but meant could only take advantage of the 16 cores on our host, nothing more. We have to rewrite to have a distributed architecture, and communicate over the network (via sockets, etc.) to worker procs on other nodes. Well, now we're talking about IO, and network coordination among servers. In our master control program we are no longer necessarily doing any ballot counting. That can all be done by other servers, the child worker nodes. Therefore our master process may not have as much compute to do as before, but it will have more network IO to do. So we might not need as many cores, or as fast of a CPU clock, as we had before, on our master server's host. We will have to redesign it so the parallelizable, delegate-able work flows get chopped up, assigned and delivered to (or otherwise made readable by) the worker nodes. So it will take some work to do that. And that master

program will need to spend time waiting for all the sub-tasks to finish on the other servers, and wait for them to all respond back with results.

Ok then... So I mentioned some problems related to threading (and networking, and distributed architectures.) One of the benefits of having a single thread, single process architecture was that... it was incredibly simple. You had one shared region of working memory to read and write from, and all your code in that running program could access it as needed. When you evolved that program to have multiple threads you still had the advantage of that single shared memory region.

Therefore when we wrote our multi-threaded solution the first time, we sprinted gleefully towards the promised land of parallelized computation, drunk on its power. We ran it and then... uh oh.

Weirdness. Weird bugs. Mysterious crashes. Corruption.

Here's what happened: we were... neophytes -- knowing just enough to design a parallelized solution, and just enough to launch threads, divvy up the work, wait for them all to finish, aggregate, etc. But we overlooked some very important effects. We didn't know about race conditions. Or deadlocks (or their variant: live-locks.) Or the totally counter-intuitive pitfalls of double-checked locking, especially in Java -- a phenomenon where the reality of the implementation at runtime did NOT honor the semantic promise expressed in the original source code. Which is a deeply, deeply, extremely dangerous state of affairs. At least to a programmer's mind. Because it's a bit like saying, "Well when you press down on the break pedal of your car it does NOT necessarily actually apply the breaks. The engine might have decided, as a matter of execution optimization, to keep going full speed." Thereby causing a painful surprise when you ram into a brick wall. And I have to admit something a little embarrassing. I remember a point in my career where I thought I knew everything there was to know about thread issues in Java, and therefore correct multi-threaded design in Java, and so was a relative badass programmer in terms of shipping "perfect" code because I personally could avoid all the anti-patterns.

And then I learned yet more.

I learned of the existence of the (potentially) broken effects of double-checked locking. While working at Orbitz. Where it was helpful, and arguably necessary, to have a very deep and complete grasp of thread effects in Java. Because if you did not then things like site performance and business metric collection and aggregation could never truly be trusted. Especially at the scales we worked at. Anyway, learning about the fallacy of double-checked locking was a little like Neo hearing Morpheus describe the "real world" to him for the first time. World view? Shattered. Though if anything my previous private takeaway about threads was yet further reinforced: tread carefully.

Be very very careful. Avoid if possible. Tread carefully if you cannot.

It's a bit like in physics when you cross the threshold from the simple falling apples and bouncing billiard balls of Isaac Newton's mechanics into the realm of Einstein's relativity, and then into Heinsenburg's quantum mechanics. Things get very very Weird and counter-intuitive. Things can become so weird and counter-intuitive that multi-threaded programming issues are often associated with so-called Heisenbugs. Meaning a kind of bug which, when you go to look for it, it seems to disappear. You'll get reports of it happening out in production but, when you go to try to reproduce it, you can't. It can be extremely hard if not impossible. The name Heisenbug was coined to

suggest quantum mechanics. A reference to a kind of statistical uncertainty baked into all of physics from which you could never escape. On a large enough scale it wasn't an issue because it tended to cancel itself out. But if you zoomed in enough it began to manifest. Once you were zoomed in enough, you could either determine the position of a particle, or it's velocity/momentum, but not both. At least not at the same moment, or with the same measurement. And that the very act of measuring a thing would cause that thing itself to change. Enter the realm of multi-threaded programming. Run your app in a dev environment, with maximum logging, with a debugger attached, breakpoints set, metrics counted and latencies timed by a seemingly otherwise omniscient profiler? You won't be able to reproduce this particular kind of evil bug. Because the conditions now are entirely different than in production. The act of trying to observe it, to measure it, has caused the thing itself to change. Perhaps to never happen at all.

But we digress, a little.

Anyway... when you do multi-thread programming what happens for a program is the following, in effect. That unified region of working memory? It becomes like a shotgun and we just blew off our foot. Because in our 1st implementation with multiple threads, we committed the sin of having the threads reading and writing to the exact same parts of memory, and we didn't ensure it was done in a coherent, graceful way. It was as if two cooks were working in the same kitchen, each preparing a different meal, each trying to use the same stove, the same sink, the same bowls, and they clobbered each other's steps, intermixing their recipes. Our code had several races between threads, and these races were causing data to be corrupted, because multiple threads were having their executions "interwoven" in time, by the CPU/OS, in such a way that caused the basic order-of-execution semantics of a piece of source code to be... violated --- at least in terms of effect.

Our first fix also caused problems. We quickly learned about "locks" (as one strategy for preventing races), coded up a solution using them, ran it, the races seemed to stop but now another kind of problem started happening -- deadlocks. I'm going to wave my hands and jump forward a bit and say, ultimately, what happened is we learned that to do multi-threaded programming correctly, with 0 bugs, we had to have a very good understanding of both what it enables and makes easy, as well as the pain points, and best practices for avoiding those pains and anti-patterns.

To continue our example system evolution, what we might have ended up doing to make our parallelized, multi-threaded (yet single server only) architecture work, was to perhaps introduce... queues. In our working memory. And then ensured that all communication between our threads was done via passing immutable messages through these queues. Messages indicating which ballot subsets to count (the work requests) and messages indicating the ballot counts returned for a given subset (the associated work results or response.)

If we were very lucky all of the raw ballot data combined might be small enough, size-wise, to fit in memory, in that single process region of working memory. (And we'd just be sure that our counting worker threads only read from that data, never modified (mutated) it.) But if we were unlucky the ballot data was too large to fit in memory, and thus would have to be read from disk (well, locally attached storage.) Or perhaps fetched across the network, from a database.

Which then gets to another problem, or constraint, related to threads, processes and distributed architectures. A multi-threaded/single-process architecture can allow you to "cheat" by keeping an otherwise large chunk of working memory all in one contiguous, shared, easily accessible space. With very very low latency access. (Meaning access to read it or write it.) Much lower latency than talking to disk, or across a network to a remote computing node. If you have to switch to a multi-process (yet single server) architecture, while it is attractive from the perspective of reduced risk of races/deadlocks, it also increases chance you'll be more wasteful of memory -- you might end up building multiple redundant copies in memory of what are essentially the same structures, the same data, the same shared libraries, etc. Now this is a very broad strokes complaint, because it depends on the sophistication (or not) of the specific OS, of your programming language, your VM (if any), etc. But I can say that in the case of many types of stereotypical "enterprise" Java web apps a pretty common issue is dependency bloat. The jar explosion. And if you combine that with a monolithic app architecture (one program codebase that tries to do everything as one process, even though each task flow might only be called rarely) you'll have a harder time scaling up your solution, because you simply won't be able to "fit" as many Java procs into a single server's memory, at once. (Assuming you tried a high process count, low threads per proc, model.) Whereas if you had only 1 Java program process, but with say 60 threads (or whatever), you might be able to take advantage of all 16 cores for workload compute, and also do lots of network communication (with worker delegates or service/database dependencies, spending most of their CPU time blocked/asleep while waiting for data transfer confirmation), with low latency and low load, reliably.

Next issue, but related. If you have to have many servers in order to parallelize, as well as to handle higher traffic, you also have to now think about where data lives, how it moves around, latency, availability, retries, cascading failures, data loss, synchronization, etc. What can be cached? What should be cached? Do you need a database? How do we do deployments? Testing? Monitoring? Backup & restores?

And then... I'll end this part and move on. There's a lot more detail and nuance I've left out above, and I'm sure I can revise it and improve it. But hopefully in broad strokes it gave a good illustration of my understanding of those topic areas.

Part two...

In my game Shattered Stars (Java desktop app) I kept the threading situation pretty simple. All reactions to user input events, as well as repainting the visuals from state, occurred by default in the main thread of the JVM. However for features like animations (showing a fleet moving/sliding across the galaxy map from one province to another) I did those state update calculations in a background thread -- I was careful to manage its lifecycle cleanly, handle exceptions in a bulletproof way, and to not modify any shared state (I believe I used a FSM pattern for the app's current mode, but it's been a while.)

In EduGamon (in Python) I gave it a multi-process, distributed web app architecture, sitting behind a reverse proxy web server at the very front. And therefore it was horizontally scalable. I didn't need threads, and pref to avoid them by default unless I had a strong reason or clear benefit. I put nginx out in front, as that web server. It

served static files, and then for certain URL patterns it was considered a dynamic request and so I configured it to hand off to a WSGI-compatible Python process running behind the scenes. Therefore nginx might have used threads and events, etc. in its own process architecture. But down in my own Python application code I designed it to use one process per concurrently handled request. Because so much safer and simpler. I always knew that if total memory usage per host ever became an issue that I could always then, at that time, refactor it to become multi-threaded. But I would not cross that bridge until or unless I needed it. As with so many other things in life it was wisest to be evidence-driven. You could imagine up the most cool sounding software possible, in theory, and yet if you never got enough users in actuality, or especially not enough paying customers, then it might not ever truly matter.

Speaking of nginx...

Every time I've personally ever used nginx in a solution I've configured it to run with multiple front-end worker threads, because in that situation it's appropriate to minimize the amount of memory needed to handle any given amount of concurrent requests (historically, nginx used much less than Apache, for example, even with otherwise similar worker configurations.). And because I trusted the nginx devs to have (probably) bullet-proof multi-threading code. I believe, and IIRC (without googling now to confirm/deny -- which would be trivial when needed) nginx used an additional strategy called non-blocking IO, where it used mechanisms like epoll, on Linux or compatible platforms. Which meant it could shrink the outstanding resource usage required, at any given moment, yet further. But even in a traditionally blocking solution (where each thread goes to sleep until woken up when any data-writing call has finished writing and can return, or new data available in the buffer to read) a multi-threaded solution is capable of using less memory than an equivalent one using multiple processes. Threads beat processes, for sake of memory use efficiency, but events always beat threads -- speaking as a general rule only, because it is painted in broad strokes.

In several of the iOS apps I wrote (for contract clients) I would have background work, or network bound tasks (like fetching files or making web service calls), be done in worker threads -- though (IIRC) I never needed to directly launch/manage threads, rather, would define "jobs" and hand them off to an iOS API which in turn was responsible for ensuring they were executed. iOS has a different set of constraints than a typical Linux-based service would have, because on a mobile device things like battery power need to be conserved, and there are less cores, running at a lower clock speed, with less memory, and both the hardware and OS are much more aggressive about not wasting resources. Also the OS wants to keep the touch UI very responsive to the enduser so it's important to be more careful about when/where/how jobs are executed, and their worst-case latency.

When I was a senior engineer on staff at Cheaptickets.com in Denver, there was a period where I worked on the core architecture of the site's codebase (a big Java web app on Linux, distributed, horizontally scalable, multiple tiers). And did production troubleshooting. I investigated and solved many legacy bugs involving their threading and networking code. I fixed many races. I fixed bugs where, say, exceptions were thrown which ended up killing manager threads, and housekeeping threads, which in turn caused downstream problems like leaks and stale caches, event loss, corruption.

There was a period where I used to joke there I should wear a T-shirt to work that said, "I see race conditions!" because I had spent so much time studying code and reading about the Java threading and memory model (books, magazines, papers, posts, etc.) that it seemed I couldn't go a week without looking at some piece of legacy code (ours or in third party libs we used) and discovering race conditions, deadlock vulns, etc.

I'll finish with a project example which might be the most pertinent to the reader if you worked for, say, a database company of some kind. Because this example of work sat precisely at the intersection of the topic of databases, plus multi-threaded programming and network-distributed architecture design. On Linux. For a major modern web e-commerce shop.

First, some context.

When I started at Cheaptickets the Ops team had evolved into a standard procedure, every night, of manually restarting all of the site's Java web app procs. All the JVM instances. Gracefully shutting them down, if they could, then restarting. With no other code/config changes. And they tried to do it in a rolling pattern, to help minimize impact on customers. And they'd do it at say 2am (local; and IIRC exactly), to try to minimize disruption further. Despite these precautions, it still caused disruption for some users, some annoyance, because we didn't haven't have a purely stateless design -- meaning some aspects of a user's surfing session were "stuck" inside particular JVM instances, in memory caches (sessions, shopping carts, etc.) Plus, there's always risk of "fat fingering", even doing a graceful, scripted rolling restart. So a human Ops staff had to be on-call to do it. Plus anyone else on call (pager duty, one from database, one from softeng (like *me* personally, very often, especially during the latter half of my employment period there)) also had to be ready to jump in and put out fires if needed. Why were they doing it? Because if they didn't, the JVMs would crash otherwise. They had leaks which would slowly wreck them otherwise. If they ran for more than a few days in prod under normal traffic they'd leak enough to start experiencing OutOfMemoryError events, and those themselves would cause mischief, potentially putting JVM instances into a "zombie" state -- technically alive and capable of serving traffic, but something was broken inside, so not really. Short story, I was asked to investigate and solve.

I found several different flaws causing leaks.

One of which was that in the purchase path flows there were cases where we started threads to do background tasks (generally related to reporting, customer service, backend, reco, etc.) and in some of those cases these threads would leak and live forever. Some would be started but never end. Some would never be started, just instantiated, and yet still live forever -- never be GC'ed. One of the root causes was a quirk I discovered (independently) in Sun's implementation of Java threads where, it was possible to instantiate java.lang.Thread, not start() it, you could then release all your references to it (thus you'd assume it would become GC'ed) and... it lived forever. Because under the hood in Sun's C/Java impl code of their JVM/JRE, they would secretly add your thread to a default ThreadGroup, as a convenience to you. If you start()-ed the thread, Sun's code would also be sure to remove it from that ThreadGroup eventually, and thus get GC-ed. But if you never called start(), it never got removed from that TG, and that TG lived forever, therefore your thread lived forever. Now that thread never truly ran, it was just a chunk of memory, so did not add to CPU load, but did

consume memory. So this slowly leaked away heap until the OOME fest began. I fixed that, and other issues, in our app. (I did not make a patched build of their JVM/JRE, just of our app codebase, to workaround this quirk.) Then Ops started to let our JVMs keep running. No manual rolling restarts anymore. Boom.

Then a new report came in. Database team said they saw a weird pattern where in production it looked like there was a slow growth in the number of database connections. A kind of leak of database connections. From some of the app servers, but not all. And it grew over time. Seemingly at a fixed pace, about once per day, at night.

Now in our prior normal QA testing, this was never seen, but they also never exercised the same conditions. Perf testing and stress testing didn't see this. So it was something that happened only in prod, and only with the database tier, and only some of our app server JVMs. I investigated. I learned there was something the database team did every night, or perhaps their Oracle databases were configured to do it, automatically (I don't remember anymore), and only in prod. As a kind of defensive precaution on their part, IIRC, some kind of rolling "refresh" signal was sent across the database sockets. Or perhaps it was a kind of heartbeat or "still alive?" message --- again, I forget exactly. The point was to try to gracefully wake-up the app side of the socket, or at least, suggest to that it should rebuild it's end. Every database connection would use resources on both sides, in terms of memory (and on database side, perhaps other kinds of things like handles or cursors -- again, I forget exactly and would have to refresh when needed.)

Anyway, what happened is on the app JVM side, our code was not simply killing or rebuilding its connections, it would create a new set, but also keep the original set around, thus effectively leaking them. Big multi-threaded app, and we had the common pattern where we weren't creating database connections as we went along, ad hoc. Because that would (we believed) cause a lot of GC churn, there's extra latency cost, etc. So instead our apps built a pool of database connections, with a fixed count (like 20 or 50, I forget exactly), and reuse them. We had code to ensure it was reasonably thread-safe, no more than 1 thread at a time using any given database connection, etc. Like a library book you would check one out then return it when no longer needed. Rather than write our own connection management layer they picked a FOSS OTS library named Proxool, for the pool. Otherwise implemented the standard JDBC interfaces. What I saw in the logs and at runtime in a debugger was that, in some cases, but not all, Proxool would build a new set of conns, but keep the previous set around -- no longer used but still using resources. Not on all servers in prod, but some. Binary jar. So I found the source associated with that public release version. Studied the database connection lifecycles carefully, the pool management code, init, teardown, resets, edge cases, error handling, thread safety, everything.

Then... Bingo!

Discovered a race condition in the init flow. What this race could cause was something exactly consistent with the evidence I saw in prod. It could cause one of it's pool record keeping structures to become... out of synch, with another one. Essentially, it was possible for connections to be pointed to by one collection (and thus kept alive forever) but not in another. If that happened, when the otherwise smart cleanup/close/recycle/reset logic would trigger, the code in question would not "see" those victim connections. So it only half-processed them. They did get taken out of the set of active

usable connections, but they didn't get taken out of a different set, which kept them alive. Since it was a race it didn't always happen. Only on some servers, some times. When it did happen in a particular JVM, then forever after they could leak, within that JVM instance's lifetime. And that is what happened in prod, causing what the DBA's saw.

After I confirmed it was happening, in actual reality, and the complete root cause, I next figured out what might be a clean safe fix. Made it. Built a patched in-house version of the Proxool binary jar. Bumped our dependency config up to use it. Deployed to test, QA, perf, stress, etc. All green. Approved for prod. I volunteered to "escort" the deploy (be on-call, watching status metrics, hands on keyboard, etc.) Went live.

And... the database team reported it was solved.

The mystery connection leak was gone.

One last note: I do geek out on trying to understand the performance and scalability of computing systems. Being aware of different ways to impact these qualities. And so as an exercise one day I thought it would be interesting to do a brain dump, to write down everything I was aware of from memory. I did it, then kept adding to it over time, and finally decided to share it online and maintain it as a portfolio piece. A kind of professional cheatsheet list for this topic area. Its reached around 100 entries (distinct techniques, patterns, issues, trade-offs, or rules-of-thumb), and includes several of the techniques discussed above in this piece. Like multi-threading, networking, and distributed architectures. If you're interested in checking it out, here's the address:

https://synisma.neocities.org/perf_scale_cheatsheet.pdf

thanks!

Mike