ECE459: Programming for Performance Winter 2024

Lecture 26 — Finding Bottleneck Devices

Jeff Zarnett *2024-08-08*

Characterizing Performance & Scalability Problems

Studies show that poor mobile app performance, whether that's by high usage of resources (e.g., battery, memory) or just being slow, is a major complaint that users write about in app stores [KSNH15]. The paper is somewhat older at this point, and while we might like to think that developers are doing a better job of proactively identifying issues, chances are it's still the case that a distressing number of issues are first discovered when someone posts a negative review. *"One star, because it's not possible to give zero!"*.

Given that, a user might be able to give a vague idea of what's wrong, or point out a workflow where they're dissatisfied with the performance. Then it's up to the development team to figure out what's the cause of the problem. Most of our next topics in profiling will be around the idea of CPU profiling, which is to say, examining where CPU time is going. That's generally predicated on the assumption that CPU time is limiting factor. But maybe it isn't, and before we go about focusing on how to examine CPU usage, let's check that assumption – because it could be something else.

> *It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.*
>
> - Sherlock Holmes (*A Scandal in Bohemia*; Sir Arthur Conan Doyle)

Keeping the wisdom of Mr. Holmes in mind, we need to collect evidence before reaching conclusions. At a high level we probably have the following potential culprits to start with:

1. CPU
2. Memory
3. Disk
4. Network
5. Locks

Caveats. The list above is, obviously, categories, but they are starting points for further investigation. They are listed in some numerical order, but there is no reason why one would have to investigate them in the order defined there. We'll go in that order in this topic, just for convenience.

If we get to the end of a particular investigative avenue, fixing it is a separate issue involving the application of the techniques covered elsewhere in the course. That might be really difficult, and sometimes the cause of the problem is simply that the user's device/hardware is too old (or lacking important hardware for acceleration). Games are likely the most obvious example of situations where the "minimum" and "recommended" hardware configurations are published and if you don't meet those requirements, you're going to have a bad time (if the game runs at all). That applies to apps too, though dropping older devices from the supported set is more likely a result of the device no longer getting OS/API upgrades rather than being "too slow". All of which is to say, at the end of the analysis, sometimes the correct solution is to advise the user to upgrade their hardware.

Another possible outcome of finding the bottleneck when reported by the user is that it actually isn't a performance problem as much as a programming error. For example, the user may report the GUI lagging or the application not responding (up to and including the actual system not-responding dialog), but these scenarios are not the result of low computational power on the device; they are caused instead by doing some slow or computationally-intensive task in the UI thread rather than in the background [LVVLP15]. The fix there is just to, well, fix the bug.

CPU. CPU is probably the easiest of these to diagnose. Something like `top` or Task Manager will tell you pretty quickly if the CPU is busy. You can look at the %CPU columns and see where all your CPU is going. Still, that tells you about right now; what about the long term average? Checking with my machine "Loki", that used to donate its free CPU cycles to world community grid (I was singlehandedly saving the world, you see. I mean. I did stop in 2016, and look at what's happened since then.):

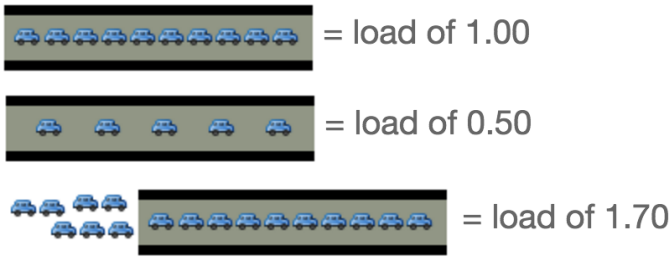
```
top - 07:28:19 up 151 days, 23:38,  8 users,  load average: 0.87, 0.92, 0.91
```

Those last three numbers are the one, five, and fifteen minute averages of CPU load, respectively. Lower numbers mean less CPU usage and a less busy machine. A small guide on how to interpret this, from [And15].

Picture a single core of a CPU as a lane of traffic. You are a bridge operator and so you need to monitor how many cars are waiting to cross that bridge. If no cars are waiting, traffic is good and drivers are happy. If there is a backup of cars, then there will be delays. Our numbering scheme corresponds to this:

1. 0.00 means no traffic (and in fact anything between 0.00 and 0.99) means we're under capacity and there will be no delay.
2. 1.00 means we are exactly at capacity. Everything is okay, but if one more car shows up, there will be a delay.
3. Anything above 1.00 means there's a backup (delay). If we have 2.00 load, then the bridge is full and there's an equal number of cars waiting to get on the bridge.

Or, visually, also from [And15]:



Being at or above 1.00 isn't necessarily bad, but you should be concerned if there is consistent load of 1.00 or above. And if you are below 1.00 but getting close to it, you know how much room you have to scale things up – if load is 0.4 you can increase handily. If load is 0.9 you're pushing the limit already. If load is above 0.70 then it's probably time to investigate. If it's at 1.00 consistently we have a serious problem. If it's up to 5.00 then this is a red alert situation.

Now this is for a single CPU – if you have a load of 3.00 and a quad core CPU, this is okay. You have, in the traffic analogy, four lanes of traffic, of which 3 are being used to capacity. So we have a fourth lane free and it's as if we're at 75% utilization on a single CPU.

Memory and Disk. Next on the list is memory. If you are using some garbage-collected language or framework, you will find lots of runs of the garbage collector, or at least very long ones when it does run; in the worst case scenario you'll see your application run out of memory and either crash or recover from it [LVVLP15].

One way to tell if memory is the limiting factor is actually to look at disk utilization. If there is not enough RAM in the system, there will be swapping and then performance goes out the window and scalability goes with. That is of course, the worst case. You can ask via `top` about how much swap is being used, but that's probably not the interesting value.

```
KiB Mem:   8167736 total,  6754408 used,  1413328 free,   172256 buffers
KiB Swap:  8378364 total,  1313972 used,  7064392 free.  2084336 cached Mem
```

This can be misleading though, because memory being "full" does not necessarily mean anything bad. It means the resource is being used to its maximum potential, yes, but there is no benefit to keeping a block of memory open for no reason. Things will move into and out of memory as they need to, and nobody hands out medals to indicate that you did an awesome job of keeping free memory. It's not like going under budget in your department for the year. Also, memory is not like the CPU; if there's nothing for the CPU to do, it will just idle (or go to a low power state, which is nice for saving the planet). But memory won't "forget" data if it doesn't happen to be needed right now - data will hang around in memory until there is a reason to move or change it. So freaking out about memory appearing as full is kind of like getting all in a knot about how "System Idle Process" is hammering the CPU[1].

You can also ask about page faults, with the command `ps -eo min_flt,maj_flt,cmd` which will give you the major page faults (had to fetch from disk) and minor page faults (had to copy a page from another process). The output of this is too big even for the notes, but try it yourself (or I might be able to do a demo of it in class). But this is lifetime and you could have a trillion page faults at the beginning of your program and then after that everything is fine. What you really want is to ask Linux for a report on swapping:

```
jz@Loki:~$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0 1313972 1414600 172232 2084296    0    0     3    39    1    1 27  1 72  0  0
 0  0 1313972 1414476 172232 2084296    0    0     0    21  359  735 19  0 80  0  0
 0  0 1313972 1414656 172236 2084228    0    0     0   102  388  758 22  0 78  0  0
 4  0 1313972 1414592 172240 2084292    0    0     0    16  501  847 33  0 67  0  0
 0  0 1313972 1412028 172240 2084296    0    0     0     0  459  814 29  0 71  0  0
```

In particular, the columns "si" (swap in) and "so" (swap out) are the ones to pay attention to. In the above example, they are all zero. That is excellent and tends to indicate that we are not swapping to disk and that's not the performance limiting factor. Sometimes we don't get that situation. A little bit of swapping may be inevitable, but if we have lots of swapping, we have a very big problem. Here's a not-so-nice example, from [Tan05]:

```
  procs                      memory    swap          io     system cpu
r  b  w   swpd   free  buff cache  si  so   bi   bo   in    cs us  sy  id
. . .
1  0  0  13344   1444  1308 19692   0 168  129   42 1505   713 20  11  69
1  0  0  13856   1640  1308 18524  64 516  379  129 4341   646 24  34  42
3  0  0  13856   1084  1308 18316  56  64   14    0  320  1022 84   9   8
```

[1] Yes, a tech journalist named John Dvorak really wrote an article about this, and here I am roasting him about it decades later because it's just so ridiculous a theory that I can't help it.

If we're not doing significant swapping, then memory isn't holding us back, so we can conclude it is not the limiting factor in scaling the application up. On to disk.

Looking at disk might seem slightly redundant if memory is not the limiting factor. After all, if the data were in memory it would be unnecessary to go to disk in the first place. Still, sometimes we can take a look at the disk and see if that is our bottleneck.

```
jz@Loki:~$ iostat -dx /dev/sda 5
Linux 3.13.0-24-generic (Loki)  16-02-13  _x86_64_ (4 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.24     2.78    0.45    2.40    11.60   154.98   116.91     0.17   61.07   11.57   70.27   4.70   1.34
```

It's that last column, %util that tells us what we want to know. The device bandwidth here is barely being used at all. If you saw it up at 100% then you would know that the disk was being maxed out and that would be a pretty obvious indicator that it is the limiting factor. This does not tell you much about what is using the CPU, of course, and you can look at what processes are using the I/O subsystems with `iotop` which requires root privileges[2].

Network. That leaves us with networks. We can ask about the network with `nload`: which gives the current, average, min, max, and total values. And you get a nice little graph if there is anything to see. It's not so much fun if nothing is happening. But you'll get the summary, at least:

```
Curr: 3.32 kBit/s
Avg: 2.95 kBit/s
Min: 1.02 kBit/s
Max: 12.60 kBit/s
Ttl: 39.76 GByte
```

So if you saw here, for example, that data was leaving at 100 Megabits per second you'd have a pretty good idea that was the limitation, but you may still be network limited at lower speeds. Intermediary devices or other non-optimal hardware can get in the way. For example, it's possible to run a wired network over power lines – this is not optimal, but it's effective in older buildings where it would be difficult or expensive to run network cables through the walls. This will limit your speed based on the condition of the wires in the wall, alongside any other devices on the same circuit adding noise to the signal. So what should be 1000 or 100 MBit might actually only be more like 32 Mbit. Wireless networks have the same problem, being affected by walls, floors, electromagnetic interference, humidity...

Testing network speed can be done using tools like speedtest.net, which gives some indication of what the network connection speed is like from a given device (upload and download). You may need to test multiple times to get a realistic picture of speed. The test validates the speed of connection to the upstream system (e.g., some server operated by the speed test service), but not necessarily to your own data centre. Both locations can have good network connections to the outside world, but might not be easily able to talk to each other well.

In my [JZ] experience, I've seen network speed issues for people working in Hong Kong using an application that has the backend deployed in a Frankfurt (Germany) data centre. The problem here is not bandwidth but latency. Tools like speedtest can tell you the latency of communication alongside the bandwidth. If you want to get an idea of the path and the latency to a particular remote system, you can use the `traceroute` tool. Here is an example from Catchpoint `https://www.catchpoint.com/network-admin-guide/how-to-read-a-traceroute`, which also provides some guidance on how to interpret a traceroute result:

```
Microsoft Windows [Version 10.0.19043.1288]
(c) Microsoft Corporation. All rights reserved.
C:\Users\Michael>tracert catchpoint.com
```

[2] https://xkcd.com/149/

```
Tracing route to catchpoint.com [64.79.149.76]
Over a maximum of 30 hops:
1 2ms 1ms 1ms 10.0.0.1
2  10ms 10ms 10ms 96.120.40.245
3 10ms 11ms 12ms 96.110.175.85
4 10ms 16ms 10ms  162.151.63.57
5 19ms 16ms 20ms 96.108.21.57
6 15ms 19ms 14ms 96.216.134.10
7           19ms  22ms  21ms  be-32121-cs02.350ecermak.il.ibone.comcast.net [96.110.42.181]
8           22ms  34ms  22ms  be-2204-pe04.350ecermak.il.ibone.comcast.net [96.110.37.38]
9           22ms  20ms  20ms  50.208.234.106
10        51ms  50ms  49ms  ae18-0.cr02.dlls02-tx.us.windstream.net [40.128.10.135]
11        73ms  72ms  72ms  ae4-0.agr03.phnd01-az.us.windstream.net [169.130.193.231]
12        84ms  73ms  75ms  ae1-0.pe05.phnd01-az.us.windstream.net [169.130.169.31]
13        85ms  84ms  85ms  h241.23.132.40.static.ip.windstream.net [40.132.23.241]
14        *          82ms  78ms  be181.las-n10s1-core1.switch.com [66.209.64.121]
15        79ms  77ms  80ms  bell011.las-agg7s5-1.switch.com [66.209.72.26]
16        79ms  77ms  79ms  64.79.139.18
17        77ms  77ms  87ms  64.19.149.76
Trace complete
```

Remember that communication latency can never be truly eliminated, because it takes non-zero time for packets to get processed through network hardware (e.g., switches), and, more importantly, the speed of light. I looked up some data on `https://wondernetwork.com/pings/New%20York` – use with a grain of salt – that says the ping from New York to Lyon is 73.21ms which is something like 83.79% of the speed of light in fibre-optic cable (as of August 2024). Even if we got it up to 100% of the speed of light in fibre-optic cable, or used some other material that had a higher speed of light in it, it can't ever get down to nothing.

One more thing that can cut into your effective bandwidth is packet loss: data getting dropped or corrupted en route. That requires some device along the line to identify that the packets are not as they should be, re-request the needed packets, and wait for them to arrive. Packet loss may be environmental, but it also might mean a device needs replacing.

Locks. The last possibility we'll consider is that your code is slow because we're waiting for locks, either frequently or for lengthy periods of time. We've already discussed appropriate use of locks, so we won't repeat that. The discussion here is about how to tell if there is a locking problem in the first place.

We'll exclude the discussion of detecting deadlock, because we'll say that deadlock is a correctness problem more than a performance problem. In any case, a previous course (ECE 252, SE 350, MTE 241) very likely covered the idea of deadlock and how to avoid it. The Helgrind tool (Valgrind suite) is a good way to identify things like lock ordering problems that cause a deadlock. Onwards then.

Unexpectedly low CPU usage, that's not explained by I/O-waits, may be a good indicator of lock contention. If that's the case, when CPU usage is low we would see many threads are blocked.

Unlike some of the other things we've discussed, there's no magical `locktrace` tool that would tell us about critical section locks, and the POSIX pthread library does not have any locks tracing in its specification [Sit21]. One possibility is to introduce some logging or tracing ourselves, e.g., recording in the log that we want to enter a critical section A and then another entry once we're in it and a third entry when we leave A. That's not great, but it is something!

I did some reading about `perf lock` but the problem is, as above, that it doesn't really find user-space lock contention. You can ask tools to tell you about thread switches but that's not quite the same. Other commercial tools like the Intel VTune claim that they can find these sorts of problems. But those cost money and may be CPU-vendor-specific.

But it's probably CPU...

Most profiling tools, and most of our discussion, will be about CPU profiling. It's the most likely problem we'll face and something that we are hopefully able to do something about.

References

[And15] Andre. Understanding Linux CPU load—when should you be worried?, 2015. Online; accessed 13-Feburary-2016. URL: http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages.

[KSNH15] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015. doi:10.1109/MS.2014.50.

[LVVLP15] Mario Linares-Vásquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 352–361, 2015. doi:10.1109/ICSM.2015.7332486.

[Sit21] Richard L. Sites. *Understanding Software Dynamics*. Addison-Wesley Professional, 2021.

[Tan05] Brian K. Tanaka. Monitoring Virtual Memory with vmstat, 2005. Online; accessed 13-February-2016. URL: http://www.linuxjournal.com/article/8178.