ECE 350: Real-Time Operating Systems Spring 2023

Lecture 25 — Reliability: Fail-Soft Operation

Jeff Zarnett *2023-03-13*

Task Failed Successfully

Except in the recent discussion of how hard drives may die and in a previous course's discussion of the Byzantine Generals Problem, we usually go through life assuming that computer systems will work as they are designed. That does not rule out the possibility of a design problem or implementation bug, of course. But in many situations, we actually have surprisingly low expectations of computer systems. Things crash, we reboot them. Something going wrong? Retry it.

Downtime is sometimes okay to address a problem, whether it's by fixing the system or replacing it. If my laptop should go for an unplanned swim in a lake, it's out of action, probably permanently. I can just buy a new laptop. I'm not happy, to be sure, but my laptop makes no promises of uptime. In a less dramatic scenario, I might install an OS update and it will be down for some period of time. The OS gives me an estimate of how long it's going to be, but that's a guess and not a promise. But maybe I need a system where that doesn't happen, even if some hardware or software update is required.

Reliability is an important element for real-time systems. Downtime may be intolerable in the case of life- or safety-critical systems, or just very expensive in the sense of needing to meet a service level agreement and having to pay contract penalties to customers if the agreement is not maintained.

If I am travelling to work via a car and get a flat tire, the car is out of action while the tire is changed. After that, I can go on: that's a repair with downtime. But some cars have the concept of run-flat tires, which allow you to travel for a limited distance at reduced speed if a tire is flat. This is to say, the system remains functional at reduced capacity until the problem can be addressed.

What we'd like to have for a real-time system is generally two distinct things: resiliency and fail-soft operation. A system that is more resilient is capable of carrying on in the event of a failure, even if not at full capacity; and fail-soft operation says that if things do indeed fail then the system will preserve as much capability as it can or terminate gracefully if it must [Sta18].

In the previous topic, we discussed what we can do to mitigate the impact of hard drive failure. That established the fact that it's not realistic to try to get ironclad devices that will last a hundred years, but instead through redundancy. If you want the system to be capable of surviving the death of a CPU, it needs to have more than one CPU in the system. Right?

Resiliency

In the words of Rocky Balboa, "It's not about how hard you hit. It's about how hard you can get hit and keep moving forward. How much you can take and keep moving forward.". When designing a system for resiliency, the question to consider is how much resiliency do you really need?

If you are designing something to not have downtime because the company loses some money when the system is down, your considerations are different to a situation where you need to avoid downtime because lives are on the line. Remember also that there are some scenarios that you cannot and should not handle because they're either too outlandish or just plain apocalyptic. If Russia decides it wishes to launch its nuclear missiles[1] and your data

[1] August 29, 1997 2:14 AM Eastern Time... `https://www.youtube.com/watch?v=4DQsG3TKQ0I`

centre is in one of the target zones, does it matter if your app is going to have some downtime right now? I think there are bigger problems to worry about.

Let's imagine that something has actually gone wrong, for whatever reason. There are a few different ways to respond or handle this situation. For the sake of the example, we'll consider primarily hardware failure, but software failures matter too.

Can We Fix It? Yes We Can! The best option for resiliency is if the system can correct the problem and carry on! That may not be sensible if the scenario is an unrecoverable hardware failure: if the magic smoke has come out of the hard drive, no amount of rebooting it will make it go back again. However, if the failure would be solved by re-initializing the device, why not?

Fixing it is somewhat easier if it's a software problem. Remember that we talked in a previous course about the ideas around deadlock recovery. Deadlock is a failure, and perhaps the system can recover from failure by pursuing one of those approaches, such as killing processes or rebooting the system.

During the process of recovery, though, the system is probably running at a lower capacity. So let's consider that...

Stiff Upper Lip. If the system cannot be restored to full capacity automatically, then the next best thing is if the system can continue as best it can, most likely at a reduced capacity.

Suppose the system has 4 CPU cores and one of them dies but the CPU was never used at more than 50% capacity even when all CPU cores were healthy. The system is worse off, yes, and its maximum capacity has been reduced, but in practice the system is still running at its original capacity. That's one reason for redundancy in the system, to be sure. That's the ideal case, but did the system really get designed with that much extra capacity?

The next possibility is that the system is running at reduced capacity until external forces come to repair the system. Reduced capacity in this sense may be that it takes longer to get answers, or less work can be done in the same amount of time.

In a real-time operating system, this loss of capacity might mean it's no longer possible to meet all deadlines. The system is considered *stable* if it will always meet the deadlines of its most critical tasks even if lower-priority tasks do not get completed [Sta18]. In this sense, it means that in the event that painful choices must be made, the most important things are prioritized. We'll consider some other scenarios for handling a problem, but then we'll return to focus on how to carry on at reduced capacity.

Shut It Down! Perhaps instead of throwing the emergency stop when things go wrong, it's possible to do an orderly shutdown. The traditional UNIX system, if it detects kernel data corruption, will write the contents of memory to disk for analysis and stop execution [Sta18]. If the problem encountered is something we cannot live with but also cannot fix, then this is a valid solution to making sure no additional damage has been done.

Stop. Hammertime. One option for failing is to just cease all execution or operation immediately. If the industrial machinery controller is in a bad state, maybe the best thing to do is actually to stop everything so the equipment stops moving. That may reduce the potential for damage to the materials being manufactured or risk to the people working in the facility. Downtime for the system is bad, but not as bad as the system killing someone.

Faults and Fault Tolerance

In the previous section, we just said vague things about something going wrong like hardware dying or something like that. But let's take a look at what is a fault and what does it mean to be fault tolerant.

The IEEE Standards Dictionary and some textbooks define a fault as an erroneous hardware or software state of some variety [Sta18]:

- **Permanent**: A fault that is always present after it occurs and can only be fixed by actually replacing or repairing the faulty component. If a GPU ceases to work due to overheating causing a failure of its internal components, that is now permanently broken and can maybe be repaired but probably has to be replaced, so it's a permanent fault. Software bugs are also permanent faults: the bug was always in the code, even if it wasn't triggered for some reason (e.g., the bug occurs only on the 29th of February) and accordingly must be fixed by a code change.
- **Intermittent**: A fault that recurs at random, unpredictable times. A faulty chip may produce the right answer most of the time but the wrong answer rarely and unpredictably. That's super frustrating because it's hard to track down and identify what's wrong.
- **Transient**: A fault that occurs once but does not recur. The standard example of this is cosmic radiation flipping a random bit from 0 to 1, but I don't think this is actually that great of an example because things like satellites (especially the ones outside the earth's magnetic field) have to contend with this as just a fact of life, making it more intermittent than transient. For terrestrial systems, maybe this is an appropriate example.

References

[Sta18] William Stallings. *Operating Systems Internals and Design Principles (9th Edition)*. Pearson, 2018.