Out of Bounds

Gerard J. Holzmann
Jet Propulsion Laboratory, California Institute of Technology

One of the things that we emphasize in software certification courses at JPL is that writing reliable code means understanding bounds. There is always only a finite amount of memory available to perform computations, a finite amount of time to do so, and every single object that we store and modify will have to be finite. All resources are necessarily bounded. Stacks are bounded, queues are bounded, file system capacity is bounded, and yes even numbers are bounded. This makes the world of computer science very different from the world of mathematics, but too few people take this into account when they write code. Perhaps the problem is that in most cases it doesn't matter much if you are aware of the limitations of your computer, because things will work correctly anyway, most of the time.

There is a great Peanuts strip where Lucy explains to a schoolmate that algebra isn't really all that hard: "*x is almost always eleven, and y us almost always nine*," she explains patiently. In math, if x and y are both positive, then it is clear that their sum will have to be larger than both x and y. Not in a computer. If we use signed 32-bit integers this property only holds if the sum is less than about two billion. Since that will be true most of the time, it is safe to say that indeed when you increment x with a positive value the result will "almost always" be greater than the original.

Almost Always

Assume that we are maintaining a signed 32-bit counter. We start it off at zero and increment it once every second. If we keep doing that it would take about 68 years for the counter to overflow. This is in fact how the number of seconds was stored in the original Unix operating system, with a clock value of zero corresponding, by convention, to January 1, 1970. Now clearly there are limitations to this method for recording time. The Unix clock can only record a time span of about 136 years: from December 1901 (recorded as negative numbers counting seconds before January 1970) until January 2038. If you keep your Unix box running long enough, the equivalent of the Y2K problem is going to occur on Tuesday January 19, 2038, when time will appear to switch back to December 13, 1901.

To fix the Unix clock isn't hard. If we simply move the time counter to a 64-bit number, the overflow will not occur for about 293 billion years. Even though this is not an unbounded amount of time, it is likely far enough, given that our solar system is expected to come to a fiery end in a mere 5 billion years.

Things change if we start counting time at a finer resolution than seconds. If, for instance, we increment our time counter once every 100 millisecond, then the signed 32-bit counter will overflow in about 6.8 years, and an unsigned 32-bit counter in 13.6 years. If we increase the precision further to one increment every 10 milliseconds a signed 32-bit counter will overflow in 248.6 days and an unsigned one in 497.1 days. Remember those numbers: we will see them again. Table 1 summarizes them.

Table 1

If you increment a 32-bit counter once every:	If the counter is signed, it will overflow in	If the counter is unsigned, it will overflow in
second	68.1 years	136.2 years
100 millisecond	6.8 years	13.6 years
10 millisecond	248.6 days = 0.68 years	1.36 years

Spacecraft Time

The Deep Impact mission was launched in January 2005. Its embedded controller calculated time as the number of 100 millisecond intervals that elapsed since January 1, 2000. Adding 13.6 years gives you a date somewhere in August 2013, which was well beyond the originally intended lifetime for this mission. The spacecraft was designed to launch an impactor to collide with the comet Tempel-1 and study its composition from the resulting dust cloud. The spacecraft completed that mission on July 4, 2005. It is common for spacecraft to pick up additional duties after their primary mission is successfully completed, provided that it has sufficient resources left to do so. On a first extended mission, Deep Impact completed a flyby of the comet Hartley-2 in November 2010. After that it still had enough fuel left for an additional extended mission, for which the mission planners chose a flyby of the asteroid 2002GT. If all had gone well, the now renamed spacecraft Epoxi would have reached that asteroid sometime in 2020.

But it was not to be. The clock counter on the spacecraft overflowed on August 13, 2013, which triggered an exception. The exception tripped a reset of the spacecraft that was intended to put it into its ‘safe mode.’ Naturally, the reset cleared everything in memory except the spacecraft clock, which meant that the spacecraft ended up in a reset cycle. Even in this desperate mode there are still things that can be done by ground controllers to recover a spacecraft by issuing so-called “hardware commands” that bypass the main CPU, but this type of intervention has to be done quickly before the spacecraft loses track of earth and can no longer receive commands. The help did not come in time, and the spacecraft was declared lost on September 20, 2013: killed by a 32-bit counter.

It is alas not the only example of an integer overflow causing problems even in safety critical systems.

Airplanes

A Boeing 787 passenger airplane (the Dreamliner) has many parts, and many of those parts have embedded controllers. All those controllers are likely to have internal clocks and count time. The embedded controllers in the GCUs, or Generator Control Units, maintain time in increments of 10 milliseconds. The corresponding counters are stored as signed 32-bit numbers. At this point you may want to glance back at Table 1 before reading on.

On July 9, 2015, the Federal Aviation Administration (FAA) issued a directive to all airline companies instructing them to reboot the GCUs on all Boeing 787 airplanes at least once every 248 days. [1] The directive said: “We are issuing this AD [Airworthiness Directive] to prevent loss of all AC electrical power, which could result in loss of control of the airplane." It explained "This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. This condition is caused by a software counter internal to the GCUs (Generator Control Units) that will overflow after 248 days of continuous power."

We already know where the 248 days came from, but it is still remarkable to see how often this type of programming flaw can occur in practice, even in systems that have passed fairly rigorous certification. The software for a commercial airplane is generally developed and tested with great care. The development process has to comply with the stipulations of the DO-178B standard (and its successor DO-178C). Especially the software for the Boeing 787 was most certainly not written in a rush. The first 787 was shown publically on July 8, 2007, and the first flight took place a little over a year later on December 15, 2009. The first commercial flight was another two years after that. We can assume that the plane, and its control software, had been in development for a good number of years at that point. When the overflow problem was discovered in early 2015, there were already close to 300 of the planes in operation.

> *"It is always the simple stuff that kills you… With all the testing systems, everything looked good."*
>
> *James Cantrell, main engineer for a $7M joint American-Russian Skipper mission, which failed because its solar panels were connected backwards. [Manila Standard, March 21, 1996, p. 26B.]*

Resource Limits

It seems counter-intuitive that failures in highly complex systems can have these embarrassingly simple causes. The reason may be that the really difficult design issues typically get ample attention, but it is the simple stuff that can get neglected. This can explain why hitting a known resource bound can bring a system to the brink of failure. Examples are easy to find, so I'll pick just two that were in the news fairly recently.

LightSail

On May 20, 2015 a small spacecraft was launched into space for a test flight. The mission, called LightSail, was designed by The Planetary Society to test the use of a large solar sail for propulsion, using the small force that is provided by the sun's rays hitting the sail. The test flight hit a snag just two days into its flight. In a mission update from May 26 the problem was described as follows: "LightSail is likely now frozen, not unlike the way a desktop computer suddenly stops responding." [2] The cause was relatively simple: a record of all telemetry data transmitted by the craft was stored in an on-board log-file, and when that log-file grew larger than 32 megabytes it crashed the flight system. Why 32 megabytes? It was not explained in the press releases, but this limit matches a limit from the old FAT12 file-system design from long ago that is still used in some embedded systems. The FAT12 design used 16-bit addresses to store sequences of 512 byte size blocks in the file system, and yes: $2^{16} \times 512$ bytes comes out to 32 megabytes.

Curiosity

Another example is much closer to home for me. The Mars Science Laboratory mission was designed and built at the lab where I work. It successfully landed a large rover, named Curiosity, on the surface of Mars in August of 2012. Since then the rover has been functioning very reliably, but its software did experience a few glitches along the way. One of those glitches occurred about six months after the landing, on February 27, 2013. The trigger for this anomaly was the sudden failure of one bank of flash memory. The rover software uses flash memory to maintain a file system for storing temporary data files, containing telemetry and images from the mission, before they are downlinked to earth. The software was designed to place the flash file-system in read-only mode when something unexpected happens. So when one bank of the flash hardware failed that is precisely what happened.

Ripple Effects

During normal operation of the rover there are many tasks that use the file system to store temporary data-products. Since the flash hardware can be slow, and the amounts of data that need to be stored large, the software is designed to use a buffering method where data can be temporarily stored in RAM memory, before it is migrated to the flash file-system, and then later downlinked to earth. All this worked perfectly, almost always.

Having been freshly reminded that all resources in an embedded system are bounded, you can now ask the question: what happens when RAM memory fills up? If this system is designed properly this should happen rarely, and it should not last long since the data temporarily stored in RAM will eventually drain to flash. So in this rare case the data-producing tasks are suspended until enough RAM memory frees up for them to resume executing. We should also ask what happens when flash memory fills up? This should be an even rarer event, because mission managers are required to keep a substantial margin of flash memory free. If it were to happen though, the software is designed to start freeing flash memory by deleting low priority files, until sufficient margin is restored. Eventually also, the higher priority files that live in flash memory will be transmitted to earth, so that the space they occupy can also be freed again.

All of that looked solid enough to pass all design reviews, code reviews, and unit tests that try to push the known limits. But then there's this one case that wasn't tested, and that's the one that happened on Mars: when the flash memory is in read-only mode due to a hardware error no further changes can be made: it cannot add or delete any more data. RAM memory now slowly fills up with new data products, waiting to migrate to flash memory, and at some point also RAM cannot accommodate any more data. All the data producing tasks now end up being suspended one by one, until all activity on the rover freezes. If there is an upside to this story it is that even in seemingly desperate cases like this the ground controllers were still able to recover the spacecraft and restore normal operations within a matter of days. And our log of lessons learned grew by one more item. We can only hope that that log is going to be bounded as well.

Should we be worried?

Programmers do of course know that in principle all resources they work with are bounded, but It can be very hard to keep reminding oneself of this sobering fact. Given the frequency with which problem related to bounds issues happen, it is wise though to plan for this. Plan on performing deliberate unit and system tests that reach, and that try to exceed, known system limits. When using a 32-bit clock, make sure you perform a test where you deliberately set the clock forward 248 days, 1.3 years, and 13.6 years, and see if your system will still work correctly. Of course you'll have a long time to think about it if you forget to do one of these tests, but you surely will sleep better if you don't.

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1]
http://rgl.faa.gov/Regulatory_and_Guidance_Library/rgad.nsf/0/584c7ee3b270fa3086257e38004d0f3e/$FILE/2015-09-07.pdf

[2] http://www.space.com/29502-lightsail-solar-sail-software-glitch.html