SOFTWARE LONGEVITY TESTING

PLANNING FOR THE LONG HAUL

BY STEVEN WOODY

ISTOCKPHOTO

How long can your software operate continuously? Is your answer measured in hours, days, months, or years? More to the point, how long do you let the software run during your test cycle?

Longevity testing is an often-overlooked part of software robustness testing for many projects—from networking routers to Web servers and even planetary rovers. Many software applications are intended to run indefinitely, in an always-on operating environment. And yet, few test plans include more than a memory leak test case or let the software run for longer than a few days.

Longevity testing (also known as soak testing or endurance testing) looks for software problems that appear only after an extended operational time. These problems fall into two broad categories: problems due to the passing of time and problems due to cumulative usage.

The Passing of Time

Accuracy drift

The passing of time can cause software accuracy to drift. A tragic example is the Patriot missile failure in February 1991. The software uptime was stored as an integer and converted to a 24-bit floating-point number, causing the software to have noticeable inaccuracy in the target-velocity tracking after only eight hours of operation and to be inaccurate to the point of failing to track the target after only twenty hours of continuous operation.

The field-mobile Patriot missile launcher was intended to operate for only a few hours at a time. However, nothing prevented the system from being operated for much longer periods. At the time of the failure, the system had been running continuously for one hundred hours. [1]

Don't rely on the software users to reboot their systems periodically. In fact, assume the opposite—that most users will operate the software continuously with never a reboot. This continuous usage goes against the practice in most software test labs, where the software is restarted every few days with a new build if not rebooted hourly between test suites.

Timeouts, expirations, and schedulers

Most software is populated with timers and schedulers, which start or stop activities minutes, hours, or days into the future. Sufficient time must be allowed during testing to exercise these timers. Timeouts and expirations to be tested commonly include user session timeouts, firewall session timeouts, ARP tables and MAC address tables aging out, routing tables expiring, SIP registrations renewing, DHCP leases renewing, DNS mappings expiring [2], and age-based passwords expiring. Software scheduler testing should include automatic maintenance activities such as daily virus scans, nightly database resynchronizations, weekly database backups, and monthly software patches or upgrades.

A software license is a type of software scheduler that commonly expires after six or twelve months. Forgotten software licenses tend to expire at the worst possible time. Even worse, a bug in the licensing software can have dramatic consequences, as demonstrated by VMware virtualized servers, which were prevented from starting up after August 12, 2008. [3] Gentle warnings should be given weeks in advance of a software license's expiring. When the system boots up with an expired license, the user should be presented with a clear path to update the license key. Software applications that depend on the software license need to handle the expiration gracefully. Avoid a total crippling of all applications when just one application has a dependency on an expired license.

Some activities are postponed when the software is very busy, and other activities are only activated when the software is sufficiently idle. Longevity testing should include tests of long busy periods (soak testing) as well as long idle periods (quiescent testing).

Soak Testing: Testing the system under heavy load for an extended period is known as soak testing or endurance testing. This testing differs from typical stress tests, which apply a burst of heavy load or a volume spike for a matter of minutes. Soak testing verifies that data buffers and message queues operate correctly over periods measured in hours or days.

Place the system under increasing amounts of load (increasing users, transactions, traffic, incoming data, or dataset size). Increase the load to three times, five times, then ten times normal, until any load-regulating, rate-limiting, or discard mechanisms become active. Then, maintain that load and don't let the system come up for air. On embedded systems, incorrect hardware watchdog/heartbeat timeouts can occur when the software is busy for extended periods of time.

Keep the software busy for an extended period so that software postpones or preempts nonessential activities in order to perform time-sensitive activities. Make sure that scheduled maintenance activities—such as database backups, database resynchronizations, virus scans, and in-service software patches or upgrades—do not crash an already busy system if the maintenance window inadvertently overlaps with a period of heavy system use.

Quiescent testing: Testing the system with no load is known as *quiescent testing*. Place the system in an idle mode with the minimum amount of activity. Allow sufficient idle time for any timers or leases to expire and any sleep, hibernate, or suspend modes to activate. After a sufficiently long idle period, resume activity on the system, verifying correct operation. This test then can be repeated for longer and longer quiescent periods.

Calendar date and time rollovers

The passing of time eventually causes clock and calendar rollovers for any software that uses a real-time clock. The Y2K rollover is the most well-known example of this type of bug [4], but there are many standard date and time rollovers that need to be tested [5], depending on the longevity of your software.

All date and time software should be tested for near-term rollover dates and times such as the one-hour adjustment for daylight saving time (in fall and spring) and February 29 (Leap Day, occurring mostly every four years). Systems using UTC or NTP need to contend with an extra leap second being added (23:59:59, 23:59:60, 00:00:00) yearly in June or December, as needed.

Longer-term rollover dates include the GPS system seconds reaching 999,999,999 on September 14, 2011, and the GPS week number rollover on April 7, 2019. Operating systems and compiler-date and time-library routines have rollover dates as well. [5] The most well-known operating system calendar rollover is the Unix 32-bit time overflow that will occur on January 19, 2038. [6]

Some calendar rollover bugs occur on non-standard rollover test dates, as demonstrated by the Tandem CLX fault-tolerant workstations, which stopped working at 3:00 p.m. on November 1, 1992. [1] A bit of research and code inspection will be needed to determine the key rollover dates to be tested for your particular software. Software testers should not hesitate to "look under the hood" at the code, as advocated by Len DiMaggio. [7]

For each rollover date and time test, manually set the date and time of your system and then verify that your software operates correctly during and after each rollover. Verify that billing cycles, reservation dates, alarm clocks, and other date-based calculations operate correctly. Verify that age-based calculations work equally well for someone born today as for someone born 122 years ago. Set the date years into the future to verify that your software does not have an unexpected date or time limitation over its expected product lifespan. Reboot the software after each rollover to verify that the system will boot up, initialize, and then operate correctly.

System Uptime and Clock Ticks

System-uptime counters and any algorithms that use the system uptime are particularly prone to failure when the counter reaches its maximum value and overflows, restarting the uptime count to zero. These uptime counters are incremented every few milliseconds by a hardware clock, making the rollover occur quite predictably after a specific number of days, depending on the period of the clock tick, as shown in table 1.

Clock tick period	Uptime counter size	Days until rollover
1 mS	signed 32 bit	24.9
2 mS	signed 32 bit	49.7
2 mS	unsigned 32 bit	99.4
5 mS	signed 32 bit	124.3
10 mS	signed 32 bit	248.6
10 mS	unsigned 32 bit	497.1

Table 1

Examples of software problems caused in 2008 by system-uptime rollovers included crashes of the Cisco Unified Communications Manager after 248 days [8] and the Cisco MDS 9000 Series Multilayer SAN Switches restarting after 497 days. [9]

Even longer uptime problems are possible, of course, as demonstrated in 2006 by the Sun StorEdge RAID Manager resetting after 828 days. [10] For accelerated testing of system-uptime rollovers, it is quite helpful to have a debug command allowing manual setting of the system-uptime counter.

Cumulative Usage

Resource Exhaustion

The cumulative usage of software tends to create more and more intentionally stored data. If storage resources are not managed carefully, this stored data causes file systems to fill up or free memory to be depleted, a problem known as resource exhaustion.

A dramatic example of resource exhaustion occurred on NASA's Spirit rover, which stopped communicating with Earth on January 21, 2004, after having landed on Mars just seventeen days earlier. Suspecting a problem with the flash memory, JPL engineers commanded the rover to boot up without reading the flash, and then deleted hundreds of unneeded files on the flash memory, which quickly addressed the problem. [11] The rover has now been running for more than five years, well surpassing its longevity design goal of ninety days of operation.

Resource exhaustion also can occur due to unintentional consumption of resources such as memory—commonly known as memory leaks. A simple bug that neglects to free up a small block of memory after it has finished using it can eventually cause the whole system to crash when no free memory is left. Unix daemon programs must be particularly well tested for memory leaks, as these programs are intended to run indefinitely. [12]

A recent example of a memory leak problem was discovered in the Cisco MDS 9000 Family SAN-OS Release 3.0 in February 2008, which caused the switches to reload after running out of memory after about 233 days. [13]

When testing longevity, verify that the software gracefully handles commonly occurring resource exhaustion scenarios. According to Jim Gray's often-cited study of Tandem computer outages, "The most common procedural mistake is letting the system fill up: either letting some file get so big that there is no more disc space for it, or letting the transaction audit trail get so large that no new log records can be written." [14] At the very least, software should monitor the resources and give clear warnings as critical resources are consumed past 90 percent, 95 percent, and 99 percent levels.

Resource monitoring is a vital tool for quick detection of resource management bugs during testing and operation, but resource exhaustion problems often creep in on resources that are not monitored, including inodes, file handles, sockets, process threads, and data buffers and queues. Longevity testing attempts to find resource exhaustion bugs in as short a time as possible.

Overflow and Wraparound

Integer overflows happen as the cumulative use of the system causes integer counters in the software to reach their maximum values.

After reaching the maximum value, the counters then go negative (for signed numbers) or rollover to zero (for unsigned numbers). These unexpected jumps can cause catastrophic problems for the software.

Transaction-based protocols that increment the transaction ID or session ID with each new transaction must handle increasingly larger session identifier numbers as time goes on. Database applications must handle increasingly larger database record ID numbers. These overflow issues can occur more rapidly with larger scale systems, with larger dataset sizes, and with higher transaction rates, all of which can be used to accelerate longevity testing.

Reentrant Code

Some bugs don't show up until the same code has been run more than once. A subroutine that works flawlessly the first time may crash the software when it runs later, due to an improperly re-initialized variable affecting the results the second, third, or nth time around.

Just such a reentrant problem occurred during simulator testing just prior to the NASA shuttle flight on the STS-2 mission in 1981, causing all four of the flight computers to lock up. The subsequent investigation revealed that a reentrant subroutine for fuel control was not properly initializing a variable during subsequent passes. [15]

Similarly, a reentrant bug in Microsoft's Internet Explorer was found in December 2008, requiring a security patch to address it. [16] In order to catch reentrant code bugs, software features must be tested repetitively during longevity testing—without any reboots or restarts of the software.

Accelerated Testing Techniques

Scaling up the system to the maximum that resources allow will help accelerate the occurrence of many longevity bugs. Increase the number of users, the number of transactions per second, the interface traffic rate, the incoming data rate, or the dataset size as your system and testing budget allow.

Another technique to accelerate longevity testing is to preset any counters, session IDs, or record numbers to within a few counts prior to the rollover point. Then, start normal operation and observe that the integer rollovers are handled correctly in the software.

A third acceleration technique is to pre-consume the software resources. Nearly fill each storage type, including flash memory, RAM, and hard disks. When creating test files to fill storage space, perform two types of accelerated tests: the first using one or two extremely large files, and the second creating thousands of minimum size files in multiple subdirectories. For both tests, repeatedly create and delete files and directories to exercise the file system data structure.

Which Longevity Test to Run?

When running a longevity test, should the software be idle, as busy as possible, or somewhere in between? Each type of environment has its advantages and disadvantages. An extended period of heavy usage followed by an extended period of normal usage will find many longevity problems. Try to compress at least one year of expected activities (logins, refreshes, transactions, calls, packets) into the first seventy-two hours of testing.

During any longevity testing, increase the sensitivity on system alarms and logs, and then periodically check for any unexpected error messages. If supported, start a run-forever command such as a continuous ping. Monitor critical software resources for leaks. Continuously verify the software operation for correctness, accuracy, and performance.

How Long to Run Longevity Tests?

Testing for the full operational life of the system is only practical on software with the maximum run time measured in just hours or days. Most software will be in operation for months or years or will be expected to run indefinitely. Even with automated testing tools and accelerated testing techniques, how long should you test software longevity?

If the test overhead or equipment costs are prohibitively high for a product, a full suite of longevity tests might only be done once. Communications satellites and space exploration probes are operated continuously for weeks at a time during system testing prior to launch. [17]

Other products may benefit from a regular verification of longevity test cases, if only for major releases. High-availability, continuous-uptime, and nonstop software should be endurance tested with a continuous, heavy load for a full eight days, as some problems will not appear until after a full seven days of continuous stress testing.

Keep track of how long the software runs continuously during testing. Document the maximum uptimes tested for idle, normal, and stress operation for each version of software. A software feature useful for longevity testing is a log message giving the system uptime at the point when any reboot, restart, or reload command is issued.

To determine how long to test longevity, perform a code review and make a list of the integer counters. Calculate how long it will take for each counter to overflow. Your longevity testing should run at least long enough to verify that counter rollovers are handled correctly.

Start longevity testing early with the available software version under development, well before the planned release date. One approach is to set up six longevity test systems (powered via an uninterruptible power supply). Once a month, verify correct operation on all six systems. Once a quarter, update the shortest-running system to the latest software version under test. Do this every three months until you have six systems, one each running 90, 180, 270, 360, 450, and 540 days. When the oldest system reaches 540 days of uptime, update the software on that system to the current version.

To summarize the steps for longevity testing, start early, put the system under heavy load, continuously monitor the performance, and let it run for days, months, or years. Remember that a few simple longevity tests can prevent major problems from eventually occurring—it's just a matter of time. **{end}**

References

[1] Neumann, Peter G. *Computer Related Risks.* Addison-Wesley Professional, 1994.

[2] Brewer, Eric A. "Lessons from Giant-Scale Services." *IEEE Internet Computing.* July/August 2001. www.cs.berkeley.edu/~brewer/Giant.pdf

[3] Keizer, Gregg. "VMware licensing bug blacks out virtual servers." *ComputerWorld.* August, 2008. www.infoworld.com/article/08/08/12/VMware_licensing_bug_blacks_out_virtual_servers_1.html

[4] Collard, Ross. "The Y2K Bust." StickyMinds.com www.stickyminds.com/s.asp?F=S3241_ART_2

[5] "Potential problem dates for computers." The Institution of Engineering and Technology, 2001. www.theiet.org/factfiles/it/index.cfm

[6] "RFC2550 – Y10K and Beyond." The Internet Society, 1999. www.faqs.org/rfcs/rfc2550.html

[7] DiMaggio, Len. "Looking Under the Hood—An Investigative Approach to Software Testing." *Software Testing and Quality Engineering*, January 2000. www.stickyminds.com/getfile.asp?ot=XML&id=5023&fn=XDD5023filelistfilename1%2Epdf

[8] Field Notice: FN – 63174- Cisco Unified Communications Manager. Cisco, 2008. www.cisco.com/en/US/ts/fn/631/fn63174.html

[9] Field Notice: FN - 63178 - MDS Fabric and Blade Switches May Reload After 497 Days of Uptime. Cisco, 2008. www.cisco.com/en/US/ts/fn/631/fn63178.html

[10] Sun StorEdge RAID Manager. Sun Microsystems, 2006. sunsolve.sun.com/search/document.do?assetkey=1-66-201560-1

[11] Reeves, Glenn and Tracy Neilson. "The Mars Rover Spirit FLASH Anomaly." Jet Propulsion Laboratory, 2005. trs-new.jpl.nasa.gov/dspace/bitstream/2014/39361/1/04-3354.pdf

[12] DiMaggio, Len. "Testing Unix Daemons." Dr. Dobb's Journal, March 2000. www.ddj.com/184404033

[13] Cisco MDS 9000 Family SAN-OS Release 3.0. Cisco, 2008. www.cisco.com/en/US/ts/fn/620/fn62818.html

[14] Gray, Jim. "A Census of Tandem System Availability between 1985 and 1990." HP Technical Reports, 1990. www.hpl.hp.com/techreports/tandem/TR-90.1.html

[15] "Excerpt from the Case Study of the Space Shuttle Primary Control System." Communications of the ACM, 1984. www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Ariane/shuttle.html

[16] Howard, Michael. "MS08-078 and the SDL." blogs.msdn.com/sdl/archive/2008/12/18/ms08-078-and-the-sdl.aspx

[17] Academy of Program / Project & Engineering Leadership. "Redesigning the Cosmic Background Explorer (COBE)." National Aeronautics and Space Administration. www.nasa.gov/pdf/293139main_COBE_case_study.pdf

Sticky Notes

For more on the following topics go to www.StickyMinds.com/bettersoftware.

- Longevity bug examples
- More reading on longevity testing