



Systems Software

Diomidis Spinellis

SYSTEMS SOFTWARE IS the low-level infrastructure that applications run on: the operating systems, language run-times, libraries, databases, application servers, and many other components that churn our bits 24/7. It's the mother of all code.

In contrast to application software, which is constructed to meet specific use cases and business objectives, systems software should be able to serve correctly any reasonable workload. Consequently, it must be extremely reliable and efficient. When it works like that, it's a mighty tool that lets applications concentrate on meeting their users' needs. When it doesn't, the failures are often spectacular. Let's see how we go about creating such software.

Writing

As an applications programmer, the first rule to consider when writing a vitally required piece of systems software is "don't." To paraphrase the unfortunate 1843 remark of the US

Patent Office Commissioner Henry Ellsworth, most of the systems software that's required has already been written. So, discuss your needs with colleagues and mentors, aiming to pin down the existing component that will fit your needs. The component could be a message queue manager, a data store, an embedded real-time operating system, an application server, a service bus, a distributed cache—the list is endless. The challenge is often simply to pin down the term for the widget you're looking for.

Once you start writing, focus on the data structures and algorithms you'll adopt. You're building infrastructure and therefore you can make few, if any, assumptions about your workload. Use reasonably efficient algorithms to avoid surprising your clients with resource hoarding and unwelcomed bottlenecks. If a design can let you serve requests in nearly constant time, your clients will expect you to implement such a behavior. In such a case, it's unreasonable for the time you take to service a request to increase with the number of elements you've served.

The data structures you choose should also gracefully accommodate the workload without placing any artificial limits on it. That's not as easy as it sounds: you're most likely to program in C and lack access to the sophisticated container libraries available in higher-level application frameworks. Use dynamically expanding buffers,

memory pools, or linked lists to handle arbitrary amounts of data.

Error-checking is a related problem. The C language doesn't offer exceptions, which you're obliged to catch, so functions return error codes, which you should check scrupulously. If you fail to do that, your code might lose data or crash and burn. As an example, at the time of writing, the GNU *time* and Windows *route* commands will silently lose their output if redirected to a full disk. Recovery from most errors is difficult, but your code should handle those well-documented cases in which the proper response to an error or short result is to retry the operation.

Then come the nitty-gritty details that affect efficiency. Be a good citizen by having your code block when it has nothing to do. Looking around for work in a polling loop wastes precious resources. Instead, determine who might have something for your process, and use the POSIX `select` and `poll` calls to wait until such work becomes available. Design your system's communication patterns using this pattern, so that a lack of work will idle all its processes.

Modern memory is at least an order of magnitude slower than the CPU, so stay away from it. Avoid repeatedly processing data in memory. Cache intermediate results, and try to obtain all the data you need from a memory location with a single access. Where possible, sidestep memory copying. For instance, the POSIX `mmap` system call

Post your comments online
by visiting the column's blog:

www.spinellis.gr/tools

allows you to transfer data between files and your application without having the operating system copy it to its buffers, while the `readv` and `writv` calls allow you to combine data from multiple buffers into a single I/O request. These two things save you the cost of copying data into a single buffer or that of multiple system calls (another fine way to waste CPU time). Thus, you exploit the goodies that modern hardware and operating systems offer you to make your code more efficient.

Although intricate dependencies on lower layers are fair game for systems software, horizontal ones aren't. Systems software should be free-standing as much as possible; your client software is likely struggling to balance multiple conflicting requirements. Arriving at the party with your own long list of uninvited guests isn't polite. Therefore, eschew dependencies on obscure libraries, tricky-to-install components, and large frameworks that might not be available by default. Make your software play well with package management systems, allowing its painless installation and updating.

In contrast to application software, where the lack of a thick manual can be a virtue, systems software should be accurately and comprehensively documented. The documentation is the contract you draw with clients; strive to write precisely how your tool will behave, how it can be configured, and how it can fail.

Testing

Testing systems software can be tricky because it often contains complex algorithms that are subjected to grueling stress levels. Instead of the leisurely input that many application programs receive from the keyboard and mouse over a working day, systems software typically has to deal with machine-generated input arriving through a fire hose over a period of months. Worse,

input coming from the outside world can even be maliciously crafted for diverse nefarious purposes.

You can accelerate stress testing your software by configuring your testing environment to exercise its edge cases. For instance, if your software's dynamically grown buffers are 64-Kbytes, test its behavior when they're just 16 bytes. If you expect to service 10 clients, check what happens when you service 500. On top of that, write a test har-

the infrastructure in which the debugger would normally run,

The solution to this problem involves instrumenting your software with copious amounts of configurable logging. This will present the software's internal state, data structures, and how one step leads to another. Hopefully, you can reproduce the error with logging turned on and then locate its cause by trawling through the detailed log records. I recently had a case where just 3 out of

Accelerate stress testing your software, by configuring the testing environment to exercise its edge cases.


ness to feed your software with a huge number of test requests of all shapes and sizes.

You can go a step further by actively downgrading the environment in which your software runs. We saw the importance of error checking; you can verify how you handle errors by introducing faults behind your software's back using tools like the *libfiou* library (<http://blitiri.com.ar/p/libfiu/>) or Chaos Monkey.

Debugging

Debugging systems software when rare, nondeterministic errors crop up is just as difficult as testing it. These aptly-named *heisenbugs* will appear only when input, timing conditions, and the software's internal state line up. Reproducing such errors can take days of stress testing. Good luck tracing them by single-stepping through a debugger. Worse, a decent debugger might not even be available, either because your code runs on a resource-constrained system or because your code is part of

7 million requests were mishandled. I was fortunate, for I could find a rare misalignment issue in the logs. Some colleagues were less lucky and had to hook a logical analyzer in the computer's guts to locate an operating system error.

So, with mean and lean code, paranoid testing, and comprehensive logging, you'll write the systems software that your applications deserve. 

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of the books *Code Reading* and *Code Quality: The Open Source Perspective* (Addison-Wesley, 2003, 2006). Contact him at dds@aueb.gr.



See www.computer.org/software-multimedia for multimedia content related to this article.