

Increasing safety-critical software development efficiency

By David Kleidermacher, Green Hills Software

This article suggests seven key technology and process decisions software developers can make to maximise safety-critical development efficiency.

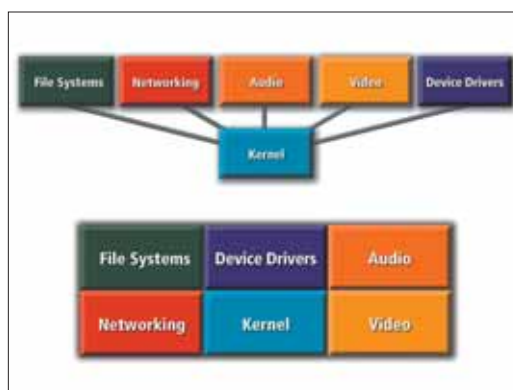


Figure 1. Microkernel vs. monolithic systems software

■ Many of the problems relating to loss in quality and safety in software can be attributed to the growth of complexity that cannot be effectively managed. Projects are often saddled with complex tangles of legacy code. A key decision designers must make is to invest in refactoring poorly architected legacy code and to properly partition new software. A large software system must be decomposed into smaller modules, each of which can be more easily understood and maintained.

Dividing a system into partitions requires that each partition have well-defined interfaces. Instead of hacking the same, shared piece of code, developers must define simple, clear interfaces for partitions and only use a well documented, or at least well understood, partitions interface to communicate with other partitions. Partitioning enables developers to work more independently and therefore more efficiently, minimizing time spent on meetings in which developers attempt to explain behaviour of their software. Refactoring a large software project in this manner can be time consuming. However, once this is accomplished, all future development will be more easily managed.

Partitioned applications should be hosted on a safety-rated, microkernel-based real-time operating system. Advances in microprocessor performance have removed the barrier to

adoption that microkernels faced when they were introduced as an alternative to monolithic operating systems decades ago (figure 1). The microkernel employs memory protection to prevent corruption of one partitions memory space by another partition. Inter-partition communication is accomplished with standard message passing constructs, such as message queues or sockets. Unless absolutely essential for better performance, shared memory should be avoided, since it blurs the lines of the designed separation.

Different operating systems (and microprocessors) have varying capabilities in terms of enforcing strict separation between components. For example, a simple real-time kernel may not make use of a computer's memory management hardware at all; multiple software applications cannot be protected from each other, and the operating system itself is at risk from flaws in application code. These flat memory model operating systems are not suitable for complex, partitioned software systems. General purpose desktop operating systems such as Linux and Windows employ basic memory protection, in which partitions can be assigned processes that are protected from corruption by the memory management unit, but do not make hard guarantees about availability of memory or CPU time resources. Some modern operating systems, such as Integrity from Green

Hills, provide strict partitioning of applications in both time and space. A damaged application cannot exhaust system memory, operating system resources, or CPU time because the faulty software is strictly limited to an assigned quota of critical resources. In addition, the kernel provides mandatory access control for peripheral I/O devices and other system objects. A rigorous partitioning of applications at the operating system level ensures that the benefits of partitioning policies used in the development process are realized during run-time. Figure 2 shows how a quota system prevents denial of service relative to the typical centralised resource pool.

Most safety-critical development processes espouse the use of a coding standard that governs how developers write code. The goal of the coding standard is to increase reliability by promulgating intelligent coding practices. For example, a coding standard may contain rules that help developers avoid dangerous language constructs, limit complexity of functions, and use a consistent syntactical and commenting style. These rules can reduce the occurrence of flaws, make software easier to test, and improve long term maintainability. Coding standard enforcement is often accomplished with human code reviews. Employing a coding standard with dozens of rules that must be verified manually is a sure way to reduce developer efficiency

■ SOFTWARE DEVELOPMENT

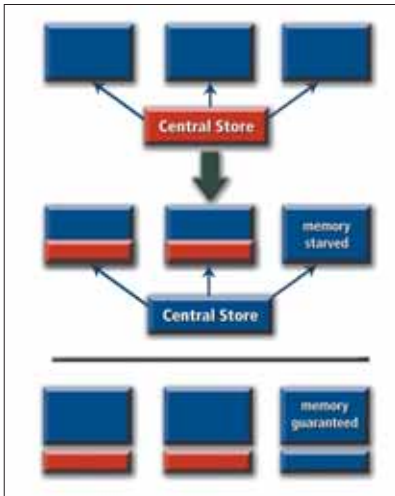


Figure 2. Operating system enforced resource availability

and time to market, even if it is increasing the reliability of code. Compiler warnings are an indicator to the developer that a code construct may be technically legal but sometimes the cause of subtle bugs. To ensure that warnings are not intentionally or accidentally ignored by developers, tell the compiler to treat all warnings as errors. This is a simple example of automated enforcement of a sensible coding standard rule. The Motor Industry Software Reliability Association (MISRA) has published guidelines for the use of the C language in critical systems, and some compilers can automatically enforce these guidelines as well.

Much has been published regarding the benefits of reducing complexity at the function level. Breaking up a software partition into smaller functions makes each function easier to understand, maintain, and test. Important quality and productivity metrics, such as McCabe complexity, subroutine interface complexity, and source comments-to-code ratio can be

used to monitor quality, with benchmarks enforced as part of the coding standard. A complexity limit rule is easily enforced at build time by calculating a complexity metric and generating a compile-time error when the complexity metric is exceeded. Once again, since the compiler is already traversing the code tree, it does not require significant additional build time to apply a simple complexity computation. Because the compiler generates an actual error pointing out the offending function, the developer is unable to accidentally create code that violates the rule. The burden of system safety certification usually falls on the product developers who frequently use operating systems that themselves have not been certified to any applicable safety standard. Thus, the developer must invest in generating the appropriate artefacts, including test cases, requirements, hazard analyses, and design documentation, for the systems software.

Safety can be improved, while reducing time to market, by choosing an operating system that has already been safety certified and a vendor that can provide safety evidence artefacts out of the box. If the operating system vendors safety certifications do not match the developer's applicable standard, then select an operating system that has successfully met a variety of safety-related standards, such as IEC-61508 and RTCA/DO-178B. The development artefacts and pedigree from such standards should go a long way towards meeting industry-specific requirements. Effective testing is well known to be one of the best mechanisms to assure that software is reliable. Therefore, it is an important component of many safety critical standards and guidance documents, such as that promulgated by the U.S. Food and Drug Administration. Testing includes functional testing, regression testing, performance testing, and coverage testing. Organisations that do not follow a rigorous development process often resort to ad-hoc testing that is an afterthought when

most of the software has already been written. Organisations that follow a rigorous process often focus testing during a release process, again after much of the software has been written. Instead, automated testing should be used throughout the development process, helping to ensure that defects are discovered as early as possible, saving development cost and time to market. One of the ways to help realise this goal is to use automated or semi-automated testing systems. Automation should be applied both to test generation as well as to test execution. Automated test generation can be applied to new software as it is written, and new tests can then be integrated into the project's automated test execution system that runs all tests on all configurations and notifies the appropriate engineers when something fails.

Static source code analyzers perform a full program analysis, finding bugs caused by complex interactions between pieces of code that may not even be in the same source file. Static analyzers find defects, such as NULL pointer references, resource leaks, and buffer overflows that can elude testing, code reviews, and normal in-field operation. Yet these flaws represent lurking safety and security vulnerabilities. For example, Green Hills DoubleCheck static analyser has found defects in the Apache web server and Linux, two of the most heavily used and tested open source software systems. Despite the promise of static analysis tools, a recent survey found that less than 5% of engineers make regular use of them. Engineers cited several barriers to adoption, including poor development environment integration and prohibitive execution time.

Commercial static analysers traditionally run as separate tools, distinct from the tool chain used to develop and build application software. Configuration can be a time consuming and recurring aggravation. With an integrated static analyser (ISA), analysis is performed within the same compiler used to build software. This approach brings with it the obvious advantage of eliminating the configuration nuisance. In addition to web page reports, an integrated analyser can generate its warnings or errors interleaved with the other standard diagnostics output by the compiler. Furthermore, integration between the project management tools and analyser enhances usability: when a defect is reported during the build process, the user can hyperlink from the build output window back to the source code quickly, rectify the error, and then return to rebuilding the program.

Commercial static analysers typically require orders of magnitude more execution time than a regular compile. Large software projects may require hours or even days of analysis time. Static analysis, if used at all, will be employed

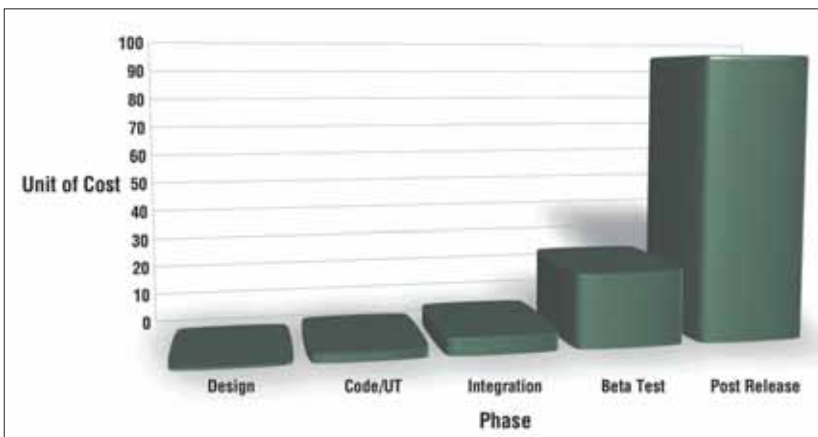


Figure 3. Increasing cost of defects

periodically or during test phases. Knowing that software development cost and time to market decrease when flaws are detected earlier in the project cycle (figure 3), it follows that static analysis tools would be more effective if execution time can be reduced to a level that encourages constant use. Developers can detect flaws while software is written and before it is ever committed to a configuration management

system - certainly prior to quality assurance testing that may occur months after the software was first created. Here again is where the ISA approach proves beneficial. The time to build and analyze software is reduced since the tool uses a single parsing pass of the code and takes advantage of highly efficient compiler dataflow algorithms for both optimization and defect detection. An ISA also takes advantage of

the IDE's existing distributed build mechanism. The parsing pass for the project's source code is distributed across available workstation assets on the user's network, dramatically reducing the total analysis time.

ISAs improve the safety of software while reducing time to market by enabling defects to be fixed earlier and more efficiently. ■