Debugging made easy - Optimizing memory-protection to pinpoint subtle software faults

by Darren Learmonth

While some kernels or executives provide support for memory protection in the development environment, few provide protected memory support for the runtime configuration, citing penalties in memory and performance as reasons. But with memory protection becoming common on embedded processors, the benefits of memory protection far outweigh the very small penalties in performance for enabling it.

The Neutrino microkernel employs powerful MMU techniques providing full protection in runtime systems - with no appreciable degradation of performance. How is this possible? Neutrino has a faster process context switch time than many other thread switch times on similar hardware.

The key advantage gained by adding memory protection to embedded applications, especially for mission-critical systems, is improved robustness.

With memory protection, we can check to see if one of the processes executing in a multitasking environment attempts to access memory that hasn't been explicitly declared or allocated for the type of access attempted. The MMU hardware can notify the kernel, which can then abort the thread at the failing/offending instruction. This protects process address spaces from each other, preventing coding errors in a thread on one process from damaging memory used by threads in other processes or even in the OS. This protection is useful both for development and for the installed runtime system, because it makes postmortem analysis possible.

During development, common coding errors (e.g. stray pointers and indexing beyond array bounds) can result in one process/thread accidentally overwriting the data space of another process. If the overwrite touches memory that isn't referenced again until much later, you can spend hours of debugging - often using in-circuit emulators and logic analyzers - in an attempt to find the guilty party.

With an MMU enabled, the OS can abort the process the instant the memory access violation occurs, providing immediate feedback to the programmer instead of mysteriously crashing the system some time later. The OS can then provide the location of the errant instruction in the failed process, or position a symbolic debugger directly to this instruction.

Using a process that is designed to write out all of the code and data spaces owned by a failing process the developer can easily use this data, in their source code debugger, to identify the offending code path.

With the QNX real-time platform, we provide a process called dumper which helps you achieve this task - writing a file that is designed to be loaded into the debugger, along with the source code of the binary at fault, if available. The file which dumper produces is written into /var/dumps/.

Assuming you have a dump file, you can easily obtain a brief view of the data with the utility, coreinfo. Of course you may need to extract more information from your dump file than coreinfo provides. Assuming you have installed the GNU Compiler Collection, you'll be able to use the GNU Debugger, gdb, to further analyze the dump file. With gdb you can glean a lot of information about your crashed process.

After changing into */var/dumps*, try:

```
coreinfo <dump file>
```

```
gdb <path to your application> <path to your dump file>
```

We have included extensive documentation on gdb in helpviewer under the section "QNX Neutrino/Development Tools/Using GDB". GDB can easily take the dump file information and place you at the faulting instruction, along with full local and global variables, and a stack history.

Moreover, systems deployed in the field that have access to, say, flash storage, can use dumper to generate true runtime problem reports that cannot be easily re-created in the test environment. Using these dump files allows the developer to pinpoint problems very quickly, fix them, and deploy field upgrades in a fraction of the time previously possible - all without resorting to in circuit emulators!