In-Field Debugging: Diagnosing Software Problems While Maintaining System Availability

Paul Leroux, Technology Analyst, QNX Software Systems
paull@qnx.com

Abstract

Software bugs that make it to market not only cause incorrect system behavior and low system availability, but also result in unhappy (and fewer) customers. Unfortunately, conventional debugging methods can themselves interfere with the availability, performance, and correct behavior of the affected system. Consequently, this paper examines debug and information-gathering techniques that can maintain system availability while generating artifacts that help diagnose and resolve software failure.

Introduction

A modern embedded system may employ hundreds of software tasks, all of them sharing system resources and interacting in complex ways. This complexity can undermine reliability, for the simple reason that the more code a system contains, the greater the probability that coding errors will make their way into the field. (By some estimates, a million lines of code will ship with at least 1000 bugs, even if the code is methodically developed and tested.) Coding errors can also compromise security, since they often serve as entry points for malicious hackers.

No amount of testing can fully eliminate these bugs and security holes, as no test suite can anticipate every scenario that a complex software system may encounter. Consequently, system designers and software developers must adopt a “mission-critical mindset” and employ software architectures that can contain software errors and recover from them quickly. Just as important, developers must employ tools and debugging techniques that help maintain system integrity during the problem-solving process.

The tools can’t introduce changes that adversely or unpredictably affect system behavior, particularly if the system is actively provid-ing service to users. And once the developer has fixed any software component, the tools and underlying operating system should make it easy to upload and monitor the fixed version, again without affecting overall system behavior and availability.

Gaining visibility through system tracing

When a complex software system performs slowly or incorrectly, the sheer number of system interactions can make pinpointing the cause a frustrating, if not monumental, task. For instance, if dozens or hundreds of threads are

running and interacting in a multi-processor or multi-core system, and one thread blocks unexpectedly, which event or interaction led to the problem? Without tools that provide a system-wide view, the cause may appear to be located in one part of the system when, in fact, it is located somewhere else.

To complicate matters, conventional tools are typically invasive, changing the behavior of the system being diagnosed. For instance, by halting only the program being debugged and not the whole system, a source debugger can change the order in which the system's operations occur. This phenomenon—often called the probe effect—can temporarily mask race conditions and introduce "errors" that occur only when debugging is performed.

Of course, conventional tools for source debugging and application profiling are still important in today's complex, multiprocessor, multilanguage systems. But they're only useful once the developer has determined which component, or set of components, to fix. To do that, the devel-oper must first understand how the system behaves as a whole. For instance, in a multi-core system, the developer must be able to determine which cores are exchanging messages, and in what order. The developer must also identify which processes or threads are involved in each inter-core transaction and trace the execution path from one core to another.

A key way to gain visibility into system behavior is through system tracing. System tracing can refer to a range of tools and techniques, including:

- *printf()* calls that annotate a program's progress
- system information tools (for instance, the Unix `top` command) that monitor task creation and track resource usage
- compiler-driven instrumentation techniques that enable application profiling and code coverage
- memory tracing tools that analyze a program's history of memory usage and that diagnose problems such as memory leaks and excessive memory fragmentation
- kernel-level instrumentation techniques that reveal events at an operating-system level, providing accurate timing traces and displaying complex interactions among multiple processes and threads

In most cases, system tracing tools can present system activity as a linear sequence of events, allowing the developer to quickly determine which events caused what outcome to occur. Depending on the style of tracing used, the developer can extract additional timing or task-interrupt information and then incorporate that information into the analysis of the system's performance or as part of the debugging process.

Problem	Technique
IPC bottleneck	Watch the flow of messages from one thread to another.
Resource contention	Watch threads as they change states.

Problem	Technique
Slow overall performance	View CPU usage to identify the processes or threads that consume the most CPU cycles.
Large interrupt latency	Search for user events to identify which thread is causing the delay, then insert custom events into that thread to pinpoint the problem.
Excessive thread migration in a multicore system	Watch threads as they migrate from one CPU to another.

Table 1. Some common problems and describes how a developer can use a system profiler to diagnose them.

Consider, for example, the system profiler, a visualization tool that forms part of the QNX® Momentics® IDE. Like a debugger that can trace the flow of control from one thread to another within a single program, this tool allows the develop to "see" how the various components in a system interact, whether they all run on a single processor or across multiple processor cores. If something goes wrong, the tool can help pinpoint when the event occurred, which software components were involved, what those components were doing, and, importantly, how to interpret the event.

Gaining insight while maintaining availability

A good system profiler is noninvasive; it can provide insight without requiring code modifications and has minimal impact on system behavior. Properly implemented, it will allow the developer to diagnose a live system without interrupting or unduly degrading the services provided by that system.

To ensure this "noninvasiveness," a system profiler typically uses fast, selective logging of system events, including messages, kernel calls, thread-state changes, and interrupts. User-written code doesn't have to be modified, since this event logging can be performed by an instrumented kernel.

Take, for example, the instrumented kernel for the QNX Neutrino® RTOS, which is simply the standard QNX Neutrino microkernel with the addition of a small, event-gathering module. When triggered, this module intercepts information about what the kernel is doing, generating time- and CPU-stamped events that are copied to a set of buffers grouped in a circular linked list. Once the number of events inside a buffer reaches a high-water mark, a logging utility either writes the data to a storage location (for instance, battery-backed SRAM) on the target or streams the data directly to the development host—the latter approach eliminates the need for extra storage on the target.

Properly designed, an instrumented kernel can run at virtually same speed as a standard, non-instrumented microkernel. Performance is affected only when events are being collected. But, even then, the kernel can provide a variety of mechanisms to ensure minimal intrusion. For instance, the kernel could allow the developer to trigger event logging only when certain conditions occur. It could also provide user-definable filters so that the logging process only collects events of interest—developers can log as many or as few events as they need.

Of course, it's always possible that the overhead of event logging, no matter

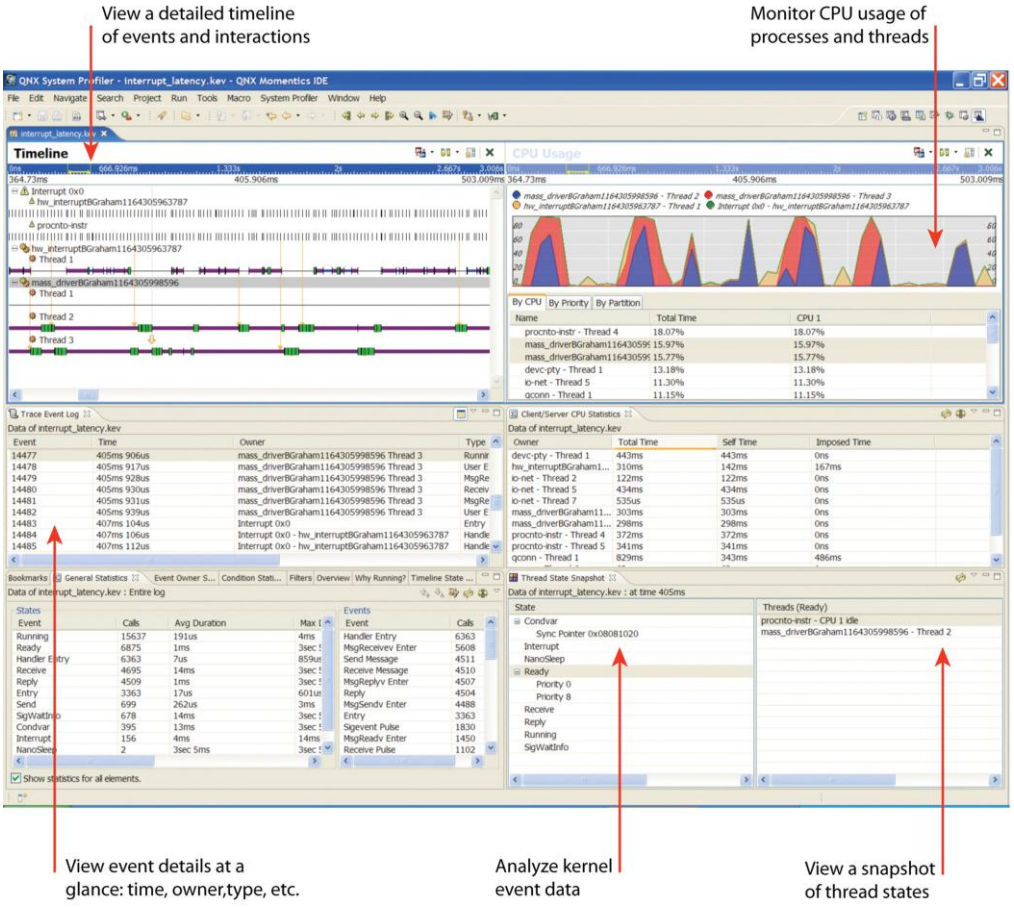

Figure 1. A system profiler can collect and analyze data for kernel calls, hardware interrupts, thread-state changes, interprocess communications, and other system-level events, allowing the developer to pinpoint deadlocks, logic flaws, and other performance-degrading conditions.

how small, will have a miniscule effect on system timing. To help the developer determine whether this problem is occurring, the instrumented kernel should be able to log all event types, including those generated by any event-logging operation. The instrumented kernel should also be fully preemptible; that way, high-priority tasks with hard deadlines can preempt event-logging operations.

Even when runtime event-filtering is applied, the events log from an instrumented kernel may contain data for many thousands of individual events. Thus, the system profiler should let the developer apply additional filters during analysis. That way, the developer can reduce the volume of data, making it easier to "zoom in" on events of interest.

Injecting user-defined events

It's often useful to determine the timing of specific events in an application. Thus, the instrumented kernel should also allow the developer to insert user-defined trace events into the system. For instance, to trace the time a packet takes to be processed, the developer could create an event when the packet arrives in the system and when the packet leaves the system.

Sometimes, the data provided by an instrumented kernel can be so detailed that it becomes difficult to understand what, exactly, the code in question is doing. User-defined events can help address the problem. By placing user-defined events throughout application code or an interrupt handler, the developer can construct an event sequence that shows which actions the program is reacting to and which sections of code were involved.

Isolating memory access violations

If designed correctly, an RTOS can simplify the task of isolating and resolving a variety of errors in a live system, including memory access violations. For instance, in a microkernel RTOS, only a small core of fundamental objects (for instance, signals, timers, scheduling) are implemented in the kernel itself. All other components—device drivers, file systems, protocol stacks, user applications—run outside the kernel as separate, memory-protected processes. See Figure 2.

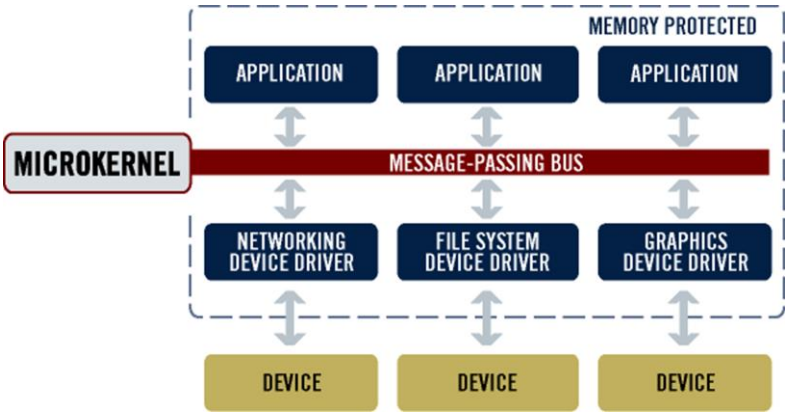

Figure 2. In a microkernel OS, virtually any component can fail and be restarted, without damaging the kernel or rebooting the entire system.

This approach offers fine-grained fault isolation, preventing any component from corrupting any other component. It also helps isolate a memory or logic error down to the component that caused it. For instance, if a driver tries to access memory outside its process container, the OS can immediately terminate the driver and reclaim the resources that the driver was using; the OS can also indicate the location of the errant instruction or position a symbolic debugger directly at that line of source code. Meanwhile, the rest of the system can continue to run, allowing the developer to diagnose the problem and to focus on resolving it.

By providing fine-grained fault isolation, a microkernel offers much faster Mean Time to Repair (MTTR) than conventional OS architectures. For instance, when a driver faults, the OS can quickly terminate and restart the driver, often within a few milliseconds. This is orders of magnitude faster than the conventional solution, which is to reboot the entire system.

Using software watchdogs to maintain high availability

Many embedded systems employ a hardware watchdog timer to detect if the software or hardware has gone “insane.” Typically, some component of the system software checks for system integrity and then strobes the timer hardware to indicate that the system is functioning normally. If the timer isn't

strobed regularly, it expires and forces a processor reset. The good news is that the system recovers from the software or hardware lockup; the bad news is that the system must completely restart, which, for some systems, can result in significant down-time and loss of state information.

Now let's look at what happens in a memory-protected system. If an intermittent software error occurs, the OS can catch the event and pass control to a user-space process called a *software watchdog*. This process can then make an intelligent decision about how best to recover from the failure. Rather than force a full reset—which is what a hardware watchdog would do—the software watchdog could:

- abort the process that failed due to a memory-access violation, then simply restart that process without shutting down the rest of the system

OR

- terminate the failed process and any related processes, initialize the hardware to a “safe” state, and then restart the terminated processes in a coordinated manner

OR

- if the failure is very critical, perform a controlled shutdown of the entire system and sound an alarm to system operators

Unlike its hardware counterpart, the software watchdog allows the developer to retain intelligent, programmed control of the embedded system, even though several processes within the control software may have failed for various reasons. A hardware watchdog timer can still help a system recover from hardware latch-ups, but for software failures the software watchdog offers much better control.

Better still, a software watchdog can monitor for system events that are invisible to a conventional hardware watchdog. For example, a hardware watchdog can ensure that a driver is servicing the hardware, but may have a hard time detecting whether other programs are talking to that driver correctly. A software watchdog can cover this hole and take action before the driver itself shows any problems.

Create core dump files for offline analysis

While performing a partial restart, the software watchdog can also collect information about the nature of the software failure. For example, if the system contains, or has access to, mass storage (flash memory, hard drive, a network link to another computer with storage), the soft-ware watchdog can invoke a dumper utility that generates a chronological archive of core dump files.

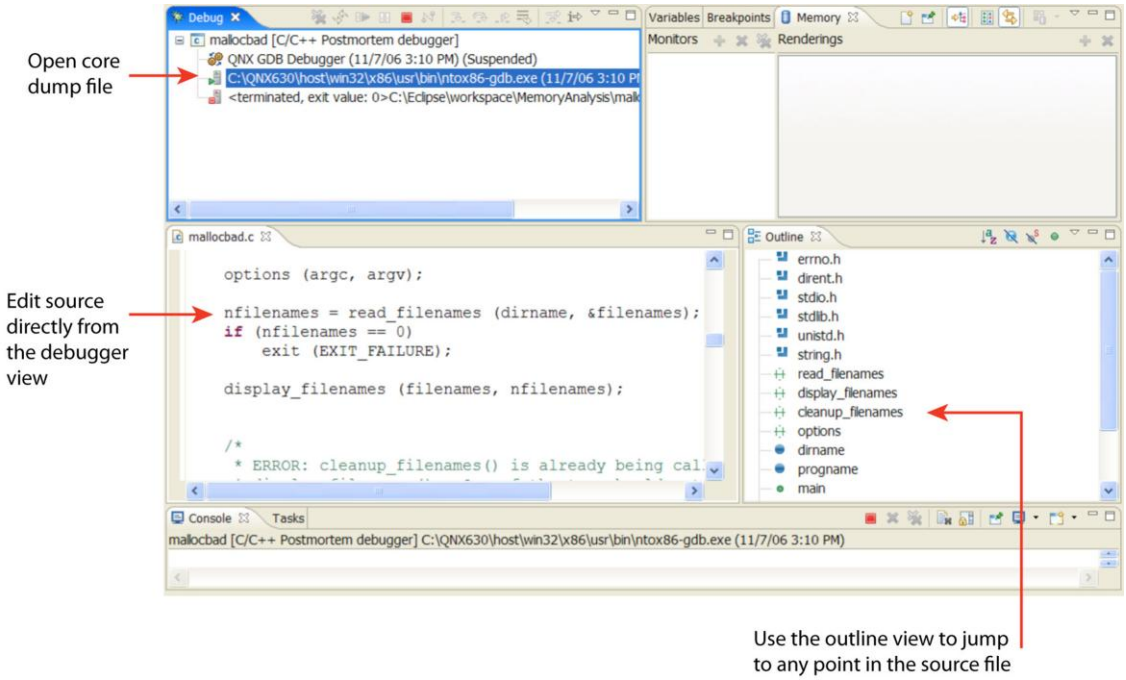

Figure 3. Performing postmortem debugging on a core dump file.

A dump file provides the information needed to identify the source line that caused a process failure, along with a history of function calls, contents of data items, and other diagnostic information. The developer can debug the dump file as if it were a live application on the target system, stepping through call stacks to determine which events led to the problem.

In some cases, the rogue process may have to be restarted as soon as possible. If so, the developer may need to restart the process and *then* complete the core dump. Some software-watchdog implementations, such as the QNX critical process manager, provide this level of control, allowing the developer to change the order of operations, grab kernel traces, and add decision-making capabilities.

Using a variation of the dumper utility (the service responsible for grabbing core information from a dying process), the developer can create a dump file for a particular process even if the processes hasn't yet attempted a memory violation. This utility simply sets a "hold" on the process, duplicates the code and data of the process into buffers, and then unholds the process. The utility then writes the buffers to a dump file. The benefit of all this? If a live sys-tem seems to be running strangely, the developer can take a snapshot for later analysis—without first forcing a shutdown or experiencing any down time.

Uploading the fix

By performing postmortem analysis of core dump files, the developer can debug and fix a problem offline, without having to remove the field-deployed

system from service. However, once the problem has been corrected, there is still the challenge of updating the target system with the new code.

When it comes to upgrading application-level code, most modern operating systems have little problem. In fact, some operating systems even allow new system services, such as drivers and protocols, to be dynamically attached to the kernel. However, because these services then run in kernel space, it's difficult to stop, remove, and replace them with new versions. Upgrading them becomes difficult, if not impossible, unless the system is taken down and rebooted.

To address these problems, an OS should, at a minimum, allow device drivers and other system services to be dynamically unloaded. But even then, there are many cases in which a driver may have to be upgraded without interrupting the service that the driver itself provides.

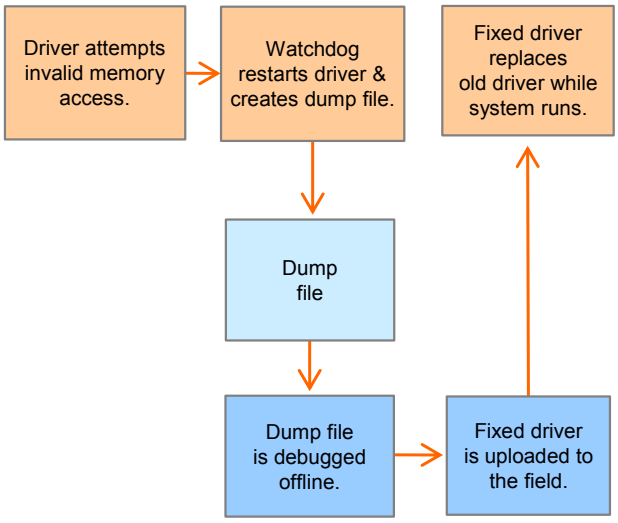

Figure 4. A software watchdog can restart problem components automatically, without downtime or operator intervention. The watchdog can also generate a process dump file for postmortem debugging, allowing developers to engineer a fix that can be uploaded to the field.

As a result, the OS should allow a new version of the driver to start while the old version is still running, and then allow the new version to gracefully take over the existing driver's duties. Once the transition is complete, the OS could terminate the old driver and recover whatever resources it was using.

Using time-partitioning to diagnose problems

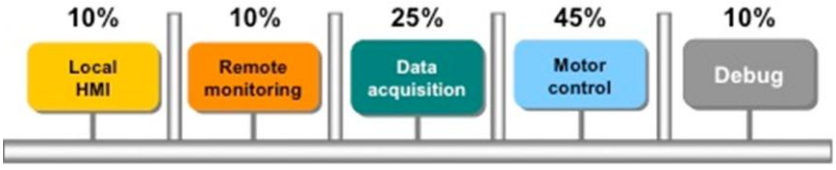

Figure 5. With time partitioning, the system designer can reserve a guaranteed amount of CPU time for each software subsystem, including debug tools.

Time partitioning provides a way to debug systems while ensuring that critical processes have the CPU cycles they need to run in a correct and timely fashion. Using this technique, developers place programs into virtual

compartments, called partitions, and allocate a guaranteed amount of CPU time to each partition. These resource guarantees can:

- contain denial-of-service (DoS) attacks
- prevent poorly written or malicious processes from monopolizing resources needed by other processes
- ensure that lower-priority functions always have the CPU cycles they require
- allow the system to dynamically support new applications and services while ensuring that existing services still have sufficient computing resources

Just as important, partitioning allows a developer to debug a system without starving critical processes of CPU time. For instance, the system designer could reserve 10% of CPU time for the debugger and any associated communications processes; see Figure 5. Because every other subsystem is also guaranteed a portion of CPU time, the cycles consumed by debug-ging operations won't affect the performance or availability of the system's core functions.

Time partitioning can simplify day-to-day testing and debugging, before the system is deployed in the field. For example, in the unit testing phase, code defects can cause runaway conditions that bring debugging to a halt. In these situations, the system appears to be locked and the developer can recover only through a reset—thereby losing useful diagnostic information. To prevent this scenario from occurring, the developer can create a partition that guarantees CPU time for console login and remote debugging. These guarantees allow the developer to continue debugging and to collect the information needed to diagnose the problem.

Allocating unused debug cycles to other processes

Not all partitioning schedulers are created equal. Some implementations strictly enforce CPU budgets at all times, so that each partition will consume its full budget even when it has no work to do. Other implementations take a more flexible approach and dynamically allocate unused CPU cycles to partitions that could benefit from the extra processing time; this approach maximizes overall CPU utilization and allows the system to handle peak demands. For instance, in QNX Neutrino adaptive partitioning, the debug partition consumes its budgeted CPU cycles only when the debugger needs them. If the debugger is idle, the scheduler will allocate the idle cycles to other partitions.

Developers can easily drop adaptive partitioning into an existing software design. It is based on the industry-standard POSIX programming model, so developers don't have to rewrite code or learn special programming techniques. Within a partition, threads are scheduled according to the traditional rules of a preemptive, priority-based scheduler. Scheduling policies such as FIFO, round robin, and sporadic all operate within a partition. In effect, each partition becomes a separate virtual processor.

Partitioning for faster error notification and recovery

Many embedded systems cannot tolerate downtime and must remain continuously available to users. Here is the formal definition of availability:

$$Availability = \frac{MTBF}{MTBF + MTTR}$$

MTBF represents the mean time between failures and MTTR the mean time to repair or resolve a particular problem. Simply put, you can increase availability both by reducing the frequency of failures and by reducing the time needed to recover from those failures.

When a hardware or software subsystem fails in a high availability embedded system, soft-ware watchdogs and other automated recovery functions must return the system to a proper operating state. The faster such recovery functions execute, the lower the mean time to repair (MTTR) and the greater the overall system availability. Time partitioning can help by ensuring that these functions have the CPU time they require.

In systems that typically run at very high CPU utilization, processes that monitor system health and report errors don't get an opportunity to run in a timely manner. The CPU guarantees provided by time partitioning address this problem and ensure that routine diagnostic functions run as intended. These functions can thus detect and report problems before the problems result in hard failures.

In the most severe cases, the operator must intervene to revive a system. To ensure the intervention is timely and effective, the system must quickly notify the operator of the failure and provide some way of diagnosing the problem. Again, partitioning helps by ensuring the system has enough CPU cycles to alert the operator and to provide guaranteed access to the user interface, be it a system console, remote terminal, or other method.