Using Resource Partitioning to Build Secure, Survivable Embedded Systems

Paul N. Leroux and Kerry Johnson  
QNX Software Systems  
pauln@qnx.com

Abstract
This paper explores how resource partitioning can improve the security, availability, and dynamic upgradeability of industrial automation systems, automotive infotainment devices, and other embedded systems. Using this technique, developers can place programs into virtual compartments, called partitions, and allocate a guaranteed amount of memory or CPU time to each partition. These resource guarantees can safeguard the system against denial-of-service (DoS) attacks; prevent poorly written or malicious tasks from monopolizing resources needed by other tasks; ensure that lower-priority functions always have the CPU cycles they require; and allow the system to dynamically support new services while ensuring that existing services still have sufficient computing resources.

Introduction
Virtually every embedded system today is connected, either physically or wirelessly, to the outside world. This network connectivity creates numerous possibilities for remote monitoring and control, and allows systems to download new applications or content on the fly. However, it also makes systems vulnerable to infiltration by a growing cadre of cyber terrorists and cyber extortionists. In fact, malicious hackers have already compromised a variety of SCADA systems, HVAC control systems, nuclear safety systems, and mobile devices, using viruses, denial-of-service (DoS) attacks, and other networked-based exploits.

Given these threats, embedded developers must pay serious consideration to security when designing any new system. However, given the huge number of embedded systems already deployed, developers need techniques that can also make legacy designs resistant to cyber attack, preferably without software redesign or recoding. Moreover, these techniques must work within the strict CPU and memory constraints that characterize the majority of embedded designs, both old and new. Security may be key, but minimizing resource consumption remains as important as ever.
Achieving secure operation while satisfying these constraints represents a significant challenge — a challenge made all the more difficult by the growing complexity of embedded software. 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. Coding errors can also compromise security, since they often serve as entry points for malicious hackers.

Unfortunately, no amount of testing can fully eliminate bugs and security holes, as no test suite can possibly anticipate every scenario that a complex software system may encounter. Systems designers and software developers must, as a result, adopt techniques that enable their products to identify and contain software faults and other threats, and to recover from them quickly. The real challenge, however, is implementing techniques that consume a minimum of development effort and computing resources.

**Secure compartments**

To address these reliability and security issues, some designs place virtual compartments, or partitions, around groups of software processes and allocate a predetermined set of resources, including CPU time, to each partition; see Figure 1. The system can thus prevent processes in any partition from erroneously or maliciously monopolizing resources needed by processes in other partitions. In the defense and aerospace industry, for example, many partitioned systems comply with the ARINC 653 specification, which provides a well-known though somewhat rigid and inefficient approach to resource partitioning.

Among other things, partitions can provide memory protection, where the OS uses the memory management unit (MMU) to control all memory access. A microkernel operating system, for instance, can partition applications, device drivers, protocol stacks, and file systems into separate, memory-protected processes. If any process, such as a device driver, attempts to access memory outside of its process container, the MMU will notify the OS, which can then terminate and restart the process.

This approach offers an immediate improvement to system reliability and availability. It also reduces development time. For instance, if any process attempts a memory-access violation during development and testing, the microkernel can identify the process responsible, at the exact instruction. It can also generate a process dump file that provides all the information a debugger needs to identify the source line that caused the problem, along with diagnostic information such as the contents of data items and a history of function calls. As a result, developers no longer have to spend days or weeks hunting down hard-to-reproduce memory errors.
Avoiding process starvation

Nonetheless, building a reliable system involves more than partitioning functionality into separate memory domains. For many systems, ensuring resource availability is also critical. If a key subsystem becomes deprived of memory or CPU cycles, the services provided by that subsystem will become unavailable to users. In a DoS attack, for instance, an external system could bombard a networked control application with requests that need to be handled by a high-priority process. That process will then overload the CPU and starve other processes of CPU cycles, making the system unavailable to users.

A security breach isn’t the only cause of process starvation. In many cases, adding software functionality to a system can push it “over the brink” and starve existing applications of CPU time. Historically, the only solution was to either retrofit hardware or redesign software.

To address these problems, systems designers need a partitioning scheme that enforces CPU budgets, either through hardware or software, to prevent processes or threads from monopolizing CPU cycles needed by other processes or threads. An embedded operating system (OS) is a good candidate to enforce CPU partition budgets, since it already provides centralized access to the CPU, memory, and other computing resources.

Conventionally, embedded OSs have used priority-based preemptive scheduling to determine which process or thread gets control of the CPU. While this approach provides an easy, well-known method to define the scheduling priority of every thread, it can lead to process starvation. For instance, let’s say you have two threads, A and B, where A has a slightly higher priority than B. If A becomes swamped with work, it will lock out B (as well as any other lower-priority thread) from accessing the CPU. See Figure 2.

In other words, priority-based scheduling offers no guarantee that lower-priority threads will access at least a fraction of the CPU. Services provided by lower-priority threads — including diagnostic and fault-recovery services that protect the system from software errors or DoS attacks — can be starved of CPU cycles for unbounded periods of time, thereby compromising system availability. These issues become more acute as system complexity, and the number of threads, increases.
Figure 2 — Priority scheduling ensures that critical threads gain access to the CPU, but it can also cause problems when a high-priority thread inadvertently or maliciously consumes all available CPU cycles. In this example, Thread A prevents all other threads from accessing the CPU after T4.

Fixed partition schedulers

To address these problems, some operating systems offer a fixed-cycle partition scheduler that allows the system designer to group processes into partitions and to allocate a percentage of CPU time to each partition. With this approach, no process in any given partition can consume more than the partition’s statically defined percentage of CPU time. For instance, let’s say a partition is allocated 30% of the CPU. If a process in that partition subsequently becomes the target of a DoS attack, it will consume no more than 30% of the total available CPU time. Other partitions can continue to access the remaining CPU cycles.

Fixed-cycle schedulers have their drawbacks, however. Since the scheduling algorithm is fixed, partitions that aren’t busy consume their allocated CPU cycles in an idle state. Meanwhile, other partitions can’t access those unused cycles, even if those partitions are busy and could benefit from the extra processing time. This approach squanders valuable (and available) CPU cycles and prevents the system from handling bursty demands. Because of this “use it or lose it” approach, fixed partition schedulers can achieve only 70% CPU utilization.

This cap on CPU utilization presents several undesirable choices to the system designer: use a faster, hotter, more-expensive processor; limit the amount of software functionality that the system can handle; or simply tolerate slower performance.
Adaptive partitioning schedulers

Another approach, called adaptive partitioning, addresses these drawbacks by providing a more dynamic scheduling algorithm. Like fixed-cycle partitioning, adaptive partitioning allows the system designer to reserve CPU cycles for a process or group of processes. The designer can thus guarantee that the load on one software subsystem won’t affect the availability of other subsystems.

Unlike fixed approaches, however, adaptive partitioning recognizes that CPU utilization is sporadic and that one or more partitions can often have idle time available. Consequently, an adaptive partitioning scheduler will dynamically reallocate those idle CPU cycles to partitions that can benefit from the extra processing time. This approach, which was pioneered by QNX Software Systems, offers the best of both worlds: it can enforce CPU guarantees when the system runs out of excess cycles (for guaranteed availability of applications and services) and can dispense free CPU cycles when they become available (for maximum CPU utilization and performance).

In Figure 3, for example, Partition 3 consumes no more 20% of CPU cycles when the system is running at capacity. But it can consume more than 20% whenever other partitions require less than their allocated CPU budget.

Adaptive partitioning offers several advantages, including the ability to:

- use realtime, priority-based scheduling when the system is lightly loaded: this allows systems to use the same scheduling behavior that they do today
- overlay the partitioning scheduler onto existing systems without code changes: users can launch existing POSIX-based applications in a partition, and the scheduler will ensure that partitions receive their allocated budget
- achieve 100% CPU utilization: this allows integrators to realize the benefits of time partitioning, but without the need for faster, more expensive processors
- guarantee that fault-detection and recovery operations have the CPU cycles they need to repair software faults, thereby improving mean time to repair (MTTR)
- allow operators to remotely monitor, troubleshoot, or upgrade the system, without interrupting the availability of critical applications
- ensure rapid, predictable response to user actions (e.g. button push, remote console command, voice command), no matter how busy the system may become
- prevent malware and DoS attacks from stealing CPU time needed by legitimate programs

While adaptive partitioning offers greater flexibility, fixed-cycle scheduling may be desirable in some situations. To address this requirement, an implementation of adaptive partitioning should allow the system designer to configure a system with fixed partition budgets and no CPU time “borrowing.” This approach allows system designers to choose the most appropriate scheduling behavior for their application requirements.
Figure 3 — Adaptive partitioning enforces CPU budgets when the system is loaded and dispenses free CPU cycles during periods of lower processor utilization. For instance, processes in Partition 3 can use CPU cycles allocated to the other three partitions when those partitions don’t consume all of their CPU budget.

Partition inheritance

To further enforce CPU budgets, a partitioning scheduler can provide partition inheritance. The principle behind this mechanism is simple: Whenever a server process (for instance, a networking driver or a file system) executes requests on behalf of a client application, the client is billed for the server’s time. As a result, runaway clients can’t monopolize drivers and services needed by other applications. Moreover, server processes can run with a minimal CPU budget (i.e. only the budget required to perform background tasks). Consequently, systems designers don’t have to reengineer the server budget when more client applications are added. See Figure 4.

Figure 4 — With partition inheritance, any work that a file system (or other system service) performs for a client is charged to the client’s budget. That way, a runaway client can’t monopolize system services needed by other applications.
**Increased system availability**

When a hardware or software subsystem fails in a high availability embedded system, automated recovery functions must return the system to a proper operating state. An example is a "software watchdog" that automatically detects and restarts failed processes. The faster such recovery functions execute, the lower the mean time to repair (MTTR) and the greater the overall system availability. An approach such as adaptive partitioning can help by ensuring that these functions have the CPU time they require.

**Scaling partitions on multi-core processors**

To date, partitioning has been used almost exclusively in single-processor environments. However, with the growing proliferation of multi-core processors, developers now need a way to implement partitions across two, four, eight, or more processing cores.

Therein lies a challenge. In a single-processor environment, the RTOS scheduler allocates CPU capacity to each partition. Ideally, the RTOS scheduler can simply extend this concept across all the processing cores in a multi-core system. Unfortunately, many RTOSs don’t provide this capability, largely because of their limited multiprocessing capabilities.

An RTOS may support one or more of the following multiprocessing models:

- **Asymmetric multiprocessing (AMP)** — Treats each core as a discrete CPU. A separate instance of the RTOS runs on each core, forcing the developer to statically configure memory, interrupts, and other shared system resources. Applications running on a given core can use only the resources that have been statically configured for that core.

- **Symmetric multiprocessing (SMP)** — A single instance of the RTOS manages all of the cores. The RTOS transparently manages shared system resources, allowing them to be used by any application running on any core. In addition, the RTOS can dynamically schedule any process on any core, enabling full CPU utilization. This approach not only simplifies development, but also offers greater flexibility when using secure partitions in a multi-core environment.

- **Bound multiprocessing (BMP)** — Extends SMP by allowing the developer to bind any process (and all of its associated threads) to a specific core. This approach simplifies migration by allowing legacy applications to run unmodified on a multi-core system, while enabling newer applications to take advantage of the parallelism offered by SMP.

By its very nature, AMP cannot address processing requirements that extend across multiple cores. Thus, any partition in AMP is limited to a portion of a single core, up to 100%. In SMP and BMP, on the other hand, the RTOS has an overall system view, allowing it to use the entire CPU capacity of the multi-core processor (i.e. all the cores) for partitioning. System designers can, as a result, flexibly map partitions onto a number of cores in whatever manner is dictated by system requirements — and independently of processor boundaries. For instance, in Figure 5, Secure Partition 1 spans across two CPU cores while the other partitions run on single cores.
This flexibility allows system designers to accommodate platform evolution, in both software and hardware. For example, introducing new software features to an application may stress a partition to the point where a single CPU core can no longer handle the processing load. If so, the partition can easily be expanded to encompass two cores. Migrating to a new hardware platform may also require changes to partition budgets, again creating the need to include multiple cores in a partition.

If a partition requires a dedicated CPU core, the system designer can use bound multiprocess-ing to ensure that only one partition executes on that core.

Figure 5 — In the symmetric and bound multiprocessing models, partitions can flexibly span across multiple cores. For instance, Secure Partition 1 in the above diagram spans across cores 1 and 2. This approach accommodates future growth in software features and makes it easier to migrate to new hardware platforms.

Pain-free software integration

Partitioning can also avoid the performance and reliability problems that inevitably arise when integrating multiple software subsystems. For example, an OEM may need to integrate application programs from one vendor, protocol stacks from another, multimedia middleware from yet another, and an embedded database from still another. The manufacturer must then combine those components with multiple software subsystems developed inhouse, each written by a separate development group.

Given the parallel development paths, performance problems invariably arise at the integration phase, when, for the first time, the various subsystems begin vying with one other for CPU time and other system resources. Subsystems that worked well in isolation now respond slowly, if at all.

Solving such problems is inherently time-consuming. Once the cause has been identified, system designers must juggle task priorities, possibly change thread behavior across the system, and then retest and refine their modifications. This iterative process can easily take several calendar weeks, resulting in increased costs and delayed product.
Adaptive partitioning can head off this problem at the design stage, by allowing systems designers to allocate a predetermined CPU budget for each partition. This approach eliminates the need to implement and enforce global priority schemes, and allows developers to define a separate (and optimal) priority scheme for each subsystem. As a result, the design teams can proceed in parallel with their design and implementation.

When subsequently testing processes in their partition, developers can launch a simple, CPU-intensive program into other partitions to simulate a heavily loaded CPU. They can then test the operation and performance of their partition under a simulated worse-case condition. As a result, developers can resolve potential performance and CPU-contention issues well before the integration phase.

**Conflicting demands**

Embedded software is becoming so complex that, without some form of partitioning, system designers and software engineers will be hard-pressed to satisfy the conflicting demands for reliability, performance, security, time to market, and new features. Resource partitioning goes a long way towards addressing these requirements by providing each subsystem with a guaranteed portion of memory and CPU cycles, while still delivering the deterministic, realtime response that embedded systems require. As a result, developers can:

- contain the effects of denial-of-service (DoS) attacks
- prevent poorly written or malicious tasks from monopolizing resources needed by other tasks
- ensure that lower-priority functions, including programs that monitor and repair system faults, always have the CPU cycles they require
- allow the system to dynamically support new system services and applications, while ensuring that existing services still have the resources they need to function correctly
- simplify the integration of software subsystems from multiple development teams and suppliers

Properly implemented, a partitioning solution will also:

- work with existing designs and standard APIs
- allow system designers to dynamically control the configuration and resource budgets of every partition
- provide CPU guarantees while still enabling 100% CPU utilization
- support partitioning on multi-core processors; for instance, by enabling partitions that span multiple cores

© 2009 QNX Software Systems GmbH & Co. KG. All rights reserved. QNX, Momentics, and Neutrino are trademarks of QNX Software Systems GmbH & Co. KG, which are registered trademarks in certain jurisdictions and used under license by QNX Software Systems International Corporation. All other trademarks and trade names belong to their respective owners.