CMP
United Business Media

ELECTRONIC ENGINEERING
EETIMES
The industry newspaper for engineers and technical management

APRIL 22, 2002

Proper RTOS designs can improve device security

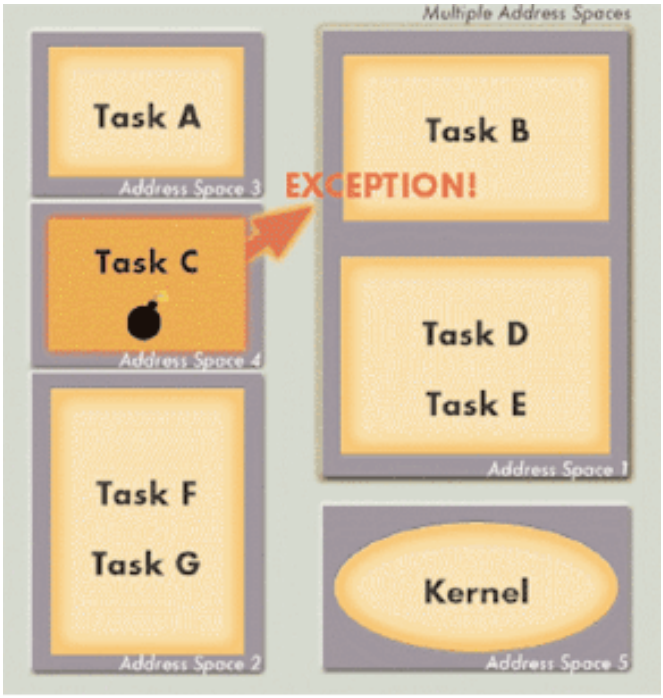
By Dave Kleidermacher, Director of Product Engineering Green Hills Software, Inc., Santa Barbara, Calif., EE Times

How secure is your cell phone? Your wireless PDA? Your TV set-top box? No connected device is immune to attack from hackers or viruses. Network security measures can block most attempts to gain unauthorized access to connected devices, but they are still not 100 percent successful in blocking intrusions. There will always be a hacker who is one step ahead of the network security developer, or the network that just isn't as well protected as it could be. Connected devices must be prepared to deal with such incursions, not just rely on the network to prevent them.

In any situation where damage is inevitable, the best strategy is to control the extent of the damage as much as possible. In addition to damage control and containment, critical system elements must be made invulnerable, lest they fall victim to an attack that could cause the entire system to fail. System integrity must be guaranteed. The good news is that this is possible today.

One part of the solution is found in the processor's Memory Management Unit (MMU). The MMU is an integral part of most 32-bit and 64-bit processors that associates certain pages of physical memory with a virtual "address space" that an individual program, or task, is able to access. A task can only access physical memory that has been "mapped" to its virtual addresses. It is just not possible for a task to access any other area of memory. Using the MMU, the system can prevent one program from overwriting the memory of another program. In this way, each program only has to watch out for itself, and not worry about being "sabotaged" by errors elsewhere in the system.

What's more, the most critical programs in a system can be debugged more extensively than less critical programs. In this way, a given amount of testing time and money can be allocated effectively, with more time devoted to debugging the most critical portions, while less time and money is spent debugging the less critical portions of the system. This is only beneficial, though, if failures in the less-tested programs cannot cause failures in the critical programs. If this can be guaranteed, then selective testing focus can pay big dividends.



In a multiple address space model, each task is able to access only its own address space. The MMU enforces such access, and any attempt to access anything outside of its own space will cause an 'exception', and the access will be denied. This prevents 'innocent' tasks (Task B, in Address Space 1) from being 'attacked' by errant tasks (Task C) in other address spaces (4).

The MMU, plays a key role in assuring system security because it puts different programs into different address spaces, thus isolating each from all others. The MMU does this by providing a hardware mechanism that can establish multiple address spaces, and detect a program's attempts to read/write/execute outside of its assigned address space.

This is the most serious type of bug, because if not

blocked by the MMU, a single bug can easily cause widespread havoc throughout the system, eventually causing catastrophic failure. But before complete failure occurs, it's anyone's guess what damage such a bug can cause along the way.

If critical programs are protected from other programs that might be more exposed to external network access, then at least the critical programs can be spared the effects of failure elsewhere in the system. In addition, if debugging investment is focused on the most important programs in a system, then system reliability will be maximized even as other, less important, programs are fielded with lesser degrees of debugging.

This "compartmentalization" of tasks not only produces a more secure system, but also provides an additional benefit, faster time to market. Consider that a product can only reach market once it has been debugged to a certain market-defined point. Using the right RTOS, only failure-critical components need be "perfect." Lesser degrees of debugging can be tolerated in other areas, since no failure there can cause total system failure. Without the right RTOS, every program must be tested to perfection before the system can be considered failsafe. Thus, the right RTOS enables a product to get to market faster, and to perform reliably once there.

Denial of service is the result of one part of the system "hogging" system resources to such an extent that other parts of the system can't operate properly. For example, suppose one program "hangs" in an endless loop. Other programs might never get CPU time. Or, perhaps one program allocates too much memory, starving other programs' needs for their own memory, preventing them from being able to run as planned. Denial of service can be caused by a programming error or intentionally by a virus or hacker through external access to the system.

Because such intrusions make a system less responsive and deterministic, one way to control such viruses is to make use of an RTOS which has stringent real-time and deterministic response.

A program needs resources to operate. It needs memory for its instructions and data, and it needs CPU cycles to execute those instructions. If a program doesn't get the memory it requires, it can't operate correctly. Perhaps the program will not be able to store the next incoming message, or save the next phone number or name, all of which require memory. If there isn't any memory available because some other program has used it all up, then the requesting program cannot perform its intended function.

Likewise, what happens if a program doesn't get the CPU cycles it requires? Suppose a virus injects a task that starts spawning subtasks without limit. In most systems, each subtask will get CPU time in a round-robin manner, sharing available CPU cycles with all other tasks at that priority level. An innocent task, at the same priority level as the one infected with the virus, will thus be starved for CPU time. Perhaps it can't decompress incoming data, or perhaps it can't find a free frequency for transmission. All because the CPU is tied up doing something unintended by the system designer.

By allocating CPU time for spawned tasks from the budget of the spawning task, rather than from a common system resource, it is impossible for any task to exhaust the time allocated to another task. Instead, all it can do is use up its own CPU time or memory budget, actions that affect only that task and no other innocent tasks. The RTOS can provide every program with guaranteed time (CPU cycles) and space (memory) resources.

A well designed RTOS guarantees that a designer-specified percentage of CPU time will always be available to designer-specified programs and absolutely prohibits other programs from consuming CPU time beyond their assigned limits. The faulty program that spawns unlimited subtasks will certainly fail, but the innocent program will not be impacted as a result of the faulty program's errors.

Similarly, each task is guaranteed a memory budget, so that no other task has access to its memory. All task activities must come from the requesting task's memory budget, making the RTOS impervious to "overspending" by virus-infected, or hacker-compromised tasks.

An appropriate RTOS does this by requiring that each program use its own assigned memory when it requests system action. An errant task simply cannot exhaust the memory of any other program, even accidentally. This guarantees the innocent program access to the system, using its own memory that is protected against theft by other programs.