CREATING SOFTWARE SEPARATION FOR MIXED CRITICALITY SYSTEMS

ANDREW CAPLES, NUCLEUS PRODUCT MARKETING MANAGER
WAQAR SADIQ, TECHNICAL MARKETING ENGINEER

INTRODUCTION

The introduction of powerful embedded processors which have been driving system consolidation for non-safety embedded devices is now driving system consolidation for safety-critical devices. While software is becoming more complex as more features merge onto a single system-on-chip (SoC), device manufacturers are facing increased scrutiny from regulatory agencies over the safety of their devices. In a safe system, the safety-critical software must have guaranteed and predictable access to compute and other system resources. In a mixed-criticality system guaranteed and predictable access must exist for the safety-critical components.

This inter-mixture of safety-critical and non-safety critical software is possible on today's modern processors, but adds to the overall design complexity. In order for guaranteed resource access by the safety application, there must be isolation from non-safety code to prevent any interference; and thus, several key areas must be considered including memory partitioning and managing access to system resources.

MEMORY REGIONS

Generally speaking, system memory is partitioned in embedded devices as static and dynamic storage (Figure 1). Static storage is where 'text' and 'data' reside; while dynamic storage contains memory for the stack (the RTOS and each RTOS thread will have its own stack) and the heap. For devices architected with a linear memory map, reliability issues occur when software subsystems reach beyond their designated memory range into memory designated to *other modules*. This overreach can transcend the memory allocated to the application to include memory space allocated to the RTOS kernel. Because the entire memory space is seen by all software modules, a subsystem through an errant memory write can potentially take down the entire device.

Therefore, an important consideration in safety devices is memory isolation which ensures that each subsystem is confined to its own designated memory container.

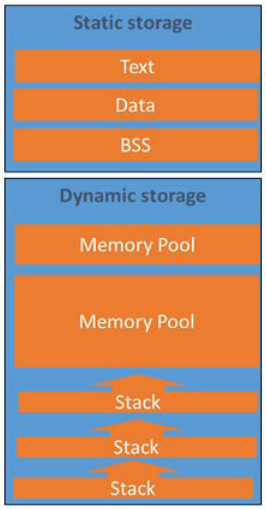


Figure 1: The difference between static and dynamic storage. Regardless of storage type, it's imperative to properly isolate memory into each respective subsystem.

MEMORY PARTITIONING

The ability to partition memory into protected regions isolated from other software subsystems establishes a foundation to build mixed criticality systems comprised of safety and non-safety software modules. Memory partitioning (Figure 2) creates space domains with defined access and read/write privileges which prevents a subsystem (or process) from accessing memory other than what has been specifically allocated to it. For example, errant pointers in a non-safe process are contained to their respective memory domain and cannot impact the memory assigned to a safety process. As a result, space partitioning increases system reliability by containing faults to the process which protects other processes and the system at large. Recovery from a fault can thus be reduced from having to restart the device to something more manageable such as simply restarting or reloading a process module.

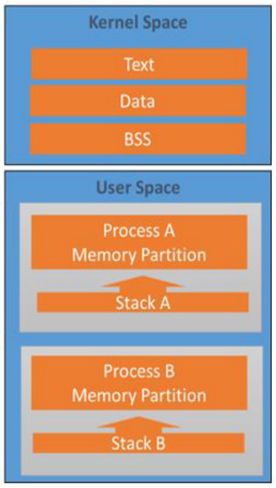


Figure 2: Memory partitioning with processes to create memory protected regions.

One solution is a RTOS with a lightweight process model (such as Mentor's Nucleus RTOS) which avoids overhead to maintain real-time system performance, and can be used throughout the spectrum of modern embedded processors from MCUs (microcontrollers) to MPUs (microprocessors).

SYSTEM ACCESS

The isolation provided by a lightweight process model provides spatial separation between safety-critical and non-safety critical processes, but spatial separation by itself does not go far enough as there is no temporal separation. In this case, both the safety and non-safety processes have full access to kernel resources. As an example, consider a non-safety process that attempts to continuously transmit data over a Local Area Network (LAN) and the effect on system resources which may be required by the safety process (Figure 3).

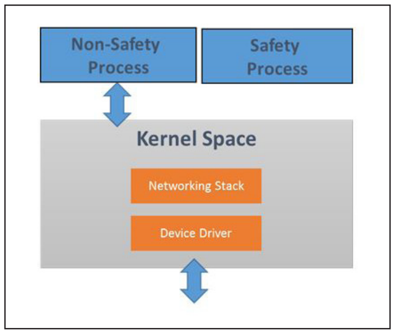
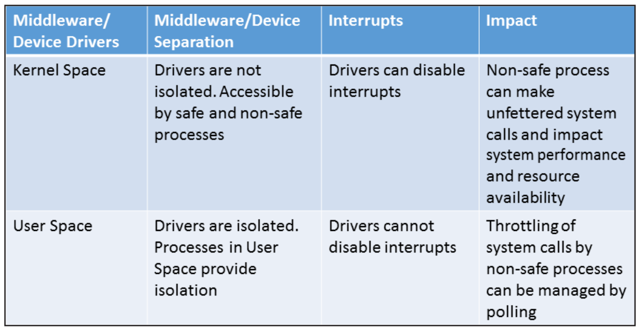


Figure 3: A non-safety process that continuously accesses kernel space, but shares kernel sevices with safety process.

As Figure 3 depicts, even though the higher priority safety process can preempt the non-safety process, a network device driver could be written such that there are periods during packet transmission in which interrupts are disabled for critical sections of code, or non-reentrant code, which inherently impedes the safety process from accessing system resources. The net result is it would be possible for the non-safety process to block a safety-related process thereby impacting the sanctity of a deterministic response. With no built-in mechanism to throttle back system access, system resources could continue to be consumed by a non-safety process. For safety-critical devices, any impediment to system resources by non-safety critical code is unacceptable. In order to ensure the safety-related code executes in a predictable manner managing system resource access is required.

Fortunately, with a lightweight process model, there are solutions to prevent unfettered access to system resources: such as moving the device drivers to user space, and moving the safety critical code to kernel space.

USER SPACE MIDDLEWARE AND DEVICE DRIVERS



Middleware/ Device Drivers	Middleware/Device Separation	Interrupts	Impact
Kernel Space	Drivers are not isolated. Accessible by safe and non-safe processes	Drivers can disable interrupts	Non-safe process can make unfettered system calls and impact system performance and resource availability
User Space	Drivers are isolated. Processes in User Space provide isolation	Drivers cannot disable interrupts	Throttling of system calls by non-safe processes can be managed by polling

Figure 4: Understanding the behaviors of kernel and user space.

For mixed criticality systems, there is a need to ensure that non-safety code cannot impact the deterministic response of the safety process. Because device drivers and middleware commonly reside in kernel space, they have access to kernel APIs, including those APIs which can be used to disable interrupts that can block safety-critical code.

To circumvent this, Nucleus Process Model makes it possible to move device drivers and middleware into protected regions in user space. Code executing in user space will have access to only a subset of kernel APIs. And as a result, the code will be unable alter the temporal domain. Thus, safety processes, when ready to execute, will be able to access system resources in a predictable and deterministic manner. If a non-safety process attempts to consume system resources for compute intensive I/O tasks (through the use of polling calls) the kernel can manage the amount of resource utilization provided (Figure 4). During periods in which safety code is idle, the kernel can use the available cycles to poll for I/O requests. Because the safety process will have the highest priority, any system resource request will take precedence over the non-safety code to ensure a deterministic response.

SAFETY-CRITICAL CODE IN KERNEL SPACE

To provide greater spatially and temporal separation, the safety-critical code can be moved to kernel space. This software architecture can be referred to as "foreground/background" mode. The safety process is considered the foreground mode executing in kernel space with the highest priority; while the background consists of user-mode processes comprised of non-safety code. The non-safe processes operating in background mode are not able to affect the safety processes in foreground mode – neither spatially (because of the memory protection provided by the process model) nor temporally (by restricting access to kernel APIs that are used for interrupt disablement).

As an example, performance tests to determine the safety-critical response were measured with a commercial operating system comprising a lightweight process model were examined during different load conditions. The environment used was Mentor's Nucleus RTOS (v2017.2) on an NXP i.MX6 SabreLite running at 998Mhz, in which the CPU, kernel, and driver were loaded as follows:

1. **CPU load:** A medium priority task incrementing a counter in a while (1) loop.
2. **Kernel load:** Two medium priority tasks synchronized through a semaphore using while (1) loops in which one tasks releases the semaphore while the other task obtains to fully load the kernel and scheduler.
3. **Driver load:** A medium priority tasks sends UDP packets of 1024 bytes continuously which generates multiple interrupts in the system for Ethernet access.

	Modules Loading the System Executing as User Processes					
CPU Load	Kernel Load	Driver Load	Interrupt Latency		Schedule Latency	
			Average	Range	Average	Range
Disabled	Disabled	Disabled	0.7 µsec	0.6-0.8	1.4 µsec	1.2-1.8
Enabled	Disabled	Disabled	0.7 µsec	0.6-0.8	1.4 µsec	1.2-1.8
Enabled	Enabled	Disabled	0.7 µsec	0.6-0.8	1.5 µsec	1.4-1.8
Enabled	Enabled	Enabled	0.7 µsec	0.6-3.0	2.2 µsec	1.4-15.0

*Figure 5: **Interrupt Latency:** Time from the occurrence of an external line interrupt until the first instruction executed in registered ISR.*
***Schedule Latency:** Time from the interrupt handler until the first instruction is executed in the highest priority waiting task.*

When looking at Figure 5, notice the response times for both interrupt and schedule latency remain consistent during tests in which the CPU and kernel are loaded. However, once the network driver is loaded, interrupt variations increased. Also note the schedule latency, which increases during driver load conditions. On the surface it appears the network driver with access to privileged APIs disables interrupts to degrade response times as it consumes the CPU cycles.

Moving the device driver to a user process with a polling mechanism yields the following results for the same test environment.

	Modules Loading the System Executing as User Processes and the Ethernet Driver Process in Poll Mode					
CPU Load	Kernel Load	Driver Load	Interrupt Latency		Schedule Latency	
			Average	Range	Average	Range
Disabled	Disabled	Disabled	0.7 µsec	0.6-0.8	1.4 µsec	1.2-1.8 usec
Enabled	Disabled	Disabled	0.7 µsec	0.6-0.8	1.4 µsec	1.2-1.8 usec
Enabled	Enabled	Disabled	0.7 µsec	0.6-0.8	1.5 µsec	1.4-1.8 usec
Enabled	Enabled	Enabled	0.7 µsec	0.6-0.8	1.5 µsec	1.4-1.8 usec

*Figure 6: **Interrupt Latency:** Time from the occurrence of an external line interrupt until the first instruction executed in registered ISR.*
***Schedule Latency:** Time from the interrupt handler until the first instruction is executed in the highest priority waiting task.*

In Figure 6, with the driver in a user process, neither the interrupt latency nor the schedule latency are impacted when the non-safety application attempts to continuously transmit UDP packets. This is because the network driver can access kernel services, but does not have access to privileged APIs to disable interrupts. Service is provided at controlled intervals. The net result is, the safety-certified module is not impacted and can continue to meet system requirements.

A PROVEN PROCESS MODEL FOR MIXED CRITICALITY SYSTEMS

Nucleus RTOS includes a lightweight process model (Figure 7) that leverages the memory manage unit (MMU) or memory protection unit (MPU) on the processor to enforce the read/write policies for each memory region. Because it is a lightweight process model, there is no need to virtualize memory. The benefit in doing so is reducing unnecessary overhead that can impact performance. Software developers and system architects can therefore deploy a system with a flat, linear memory map with protected memory regions for both user and kernel space. This type of approach also isolates individual processes in user space, and isolates both the kernel and user space. For safety-critical modules, processes can be used for isolation from non-safety critical processes to create certifiable devices that meet highest level of ISO and IEC safety standards.

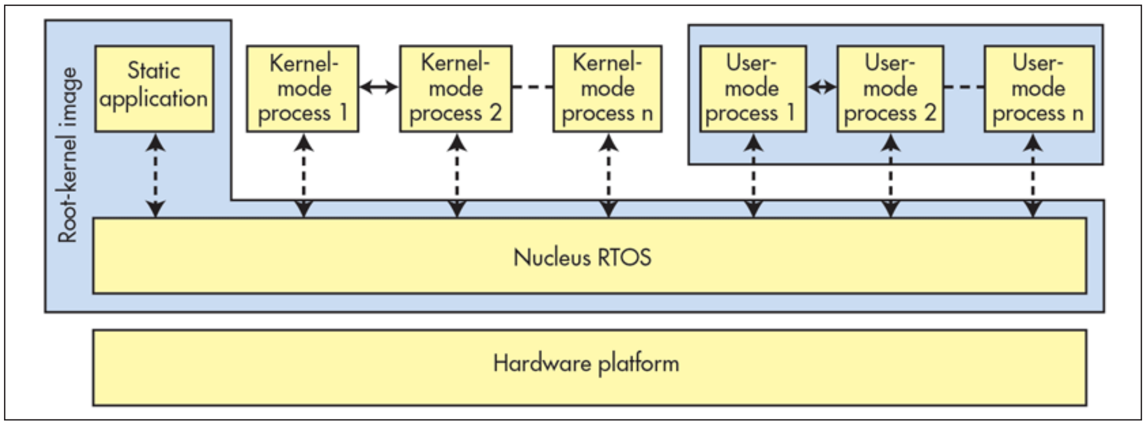


Figure 7: Nucleus process model is a lightweight approach to space partitioning which creates protected memory regions.

CONCLUSION

The use of a lightweight process model for separation can provide the isolation required for mixed criticality designs. Through the use of memory partitioning, spatial separation is achieved to create space domains with assigned access privilege levels that serve to contain faults to individual processes which increases system reliability.

Temporal separation can be achieved through the use of a background/foreground implementation which guarantees system resource access to the safety-critical code executing in kernel space as a foreground process. For many designs, utilizing a lightweight process model can provide the isolation required to meet the highest level of safety certification while retaining the real-time responses necessary to meet the most demanding performance requirements.

Visit Mentor's Nucleus supported processors page to see if Nucleus supports your current processor.

Author biographies:

Andrew Caples is a product marketing manager for the Embedded Systems Division (ESD) at Mentor. He has over 20 years of experience in start-ups and Fortune 500 high-tech companies and has served in a variety of roles ranging from technical marketing to sales management. He has a B.S. in Electrical and Computer Engineering from California Polytechnic University. His current responsibilities include product management for the Nucleus Real-Time Operating System.

Waqar Sadiq is senior technical marketing engineer for the Embedded Systems Division (ESD) at Mentor. He is currently working for Nucleus Real-Time Operating System and other runtime technologies. Waqar has spent over 17 years in the electronic industry, working for both hardware and software design and development. He has held various positions in development and management before moving to his current role at Mentor. Waqar has a B.S. in Electronics and an M.S. in Computer Engineering.

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000
sales_info@mentor.com

Silicon Valley
Mentor Graphics Corporation
46871 Bayside Parkway
Fremont, CA 94538 USA
Phone: 510.354.7400
Fax: 510.354.7467

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
11F, No. 120, Section 2,
Gongdao 5th Road
HsinChu City 300,
Taiwan, ROC
Phone: 886.3.513.1000
Fax: 886.3.573.4734

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Trust Tower
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: +81.3.5488.3033
Fax: +81.3.5488.3004

Mentor®
A Siemens Business

MGC 11-17 TECH16320-w