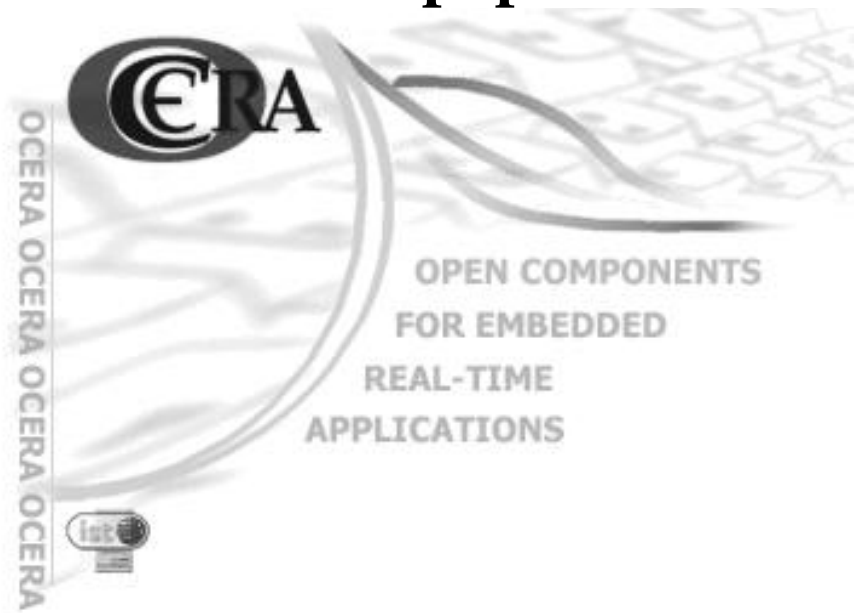
White paper



Memory protection in a threaded system

OCERA Legal Status.
by Alfons Crespo, Ismael Ripoll, Miguel Masmano, Vicente Esteve.

Published 23. April 2004

Table of Contents

Document Presentation

Project Coordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14, 46022 Valencia, Spain
Phone:	+34 963877576
Fax:	+34 963877576
Email:	alfons@disca.upv.es

Participant List

Role	Id.	Participant Name	Acronym	Country
CO	1	Universidad Politécnica de Valencia	UPVLC	E
CR	2	Scuola Superiore Santa Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA/DRT/LIST/DTSI	CEA	FR
CR	5	Unicontrols	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	Visual Tools S.A.	VT	E

Document version

Release	Date	Reason of change
1_0	03/03/04	First release

Chapter 1. Introduction

Chapter 2. What can be protected?

The application

The main difference between a conventional concurrent application based on heavy processes or threads is that all threads share the same memory space, as well as most of the system internal data structures (as file descriptors, process attributes, etc).

Threads were designed to implement concurrent (or parallel) applications in an efficient way. Although it is possible to write concurrent applications using heavy process (via the fork() system call), the strong isolation and memory protection enforced by the operating system, introduces a considerable overhead. In fact, most of the POSIX real-time extensions has been designed to work for the threading execution model (priority inheritance protocols, timed functions, etc.).

On the other hand, the flexibility and small overhead of the single execution space (with no memory protection) of a threading application highly increments the consequences of a programming bug. For example, a wild pointer may overwrite data used by other thread causing it to operate incorrectly.

The POSIX 1003.13 standard defines the Profile 51 (real-time embedded systems) as a thread model. POSIX thread standard define a thread as:

"*A single flow of control within a process. [· · ·] Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via malloc(), directly addressable storage obtained through implementation defined functions, and automatic variables, are accessible to all threads in the same process.*"

The only memory space that can be protected in this class of systems is the executive code and data, as well as the memory already not allocated to the application.

The executive

The operating system kernel itself is a large program that may also be organised in smaller functional units (hardware drivers, networking stack, filesystems, etc.). Depending on the internal operating system design, these functional units may be isolated and stand-alone enough to implement memory protection among them or not.

In the microkernel and nanokernel approaches, the kernel is structured in a highly modular collection of independent services that communicate between them using microkernel primitives. The microkernel is executed in supervisor mode, while the rest of the tasks (servers) that are part of the OS are executed in its own address space, protected from each other.

Monolithic kernels are compiled as a single big program, where all the tasks (threads, servers) are executed in the same memory space and in the highest processor privilege level. This kind of operating systems are easier to implement (it is possible to access any data

structure from any code) and the kernel is faster because the communication inside the kernel has less overhead (it can be done with simple function calls). On the other hand, monolithic kernels are less robust and are more difficult to maintain and port.

Linux and RTLinux have been designed following the monolithic approach.

Chapter 3. Processor features related to memory protection

Memory protection can only be implemented if the microprocessor (or the memory manager unit) provides some specific facilities to do it. It is not possible to implement memory protection without the help of the hardware.

Historically, memory protection mechanisms has been implemented jointly with the support for virtual memory. Memory protection information is stored in the same data structure used to translate memory address. There was two approaches to virtual memory and memory protection: segmentation and paging.

Currently, segmentation is only used in the IA32 processor family and it is only maintained for compatibility reasons. Paging is implemented in all the processors that support virtual memory (including the IA32 family), also paging is the preferred hardware mechanism used by most (if not all) UNIX systems to implement virtual memory.

Intel AI32 family

Most of these information has been directly taken from the Intel Architecture manuals (“The Intel Architecture Software Developer s Manual, Volume 3: System Programming Guide"

Intel defines four privilege levels (called “rings”). Ring-level 0 is the highest privilege level and ring-level 3 the lowest one. Ring-level 0 is considered “supervisor mode” while ring-levels 1 to 3 are “user mode”.

Segmentation

The IA32 architecture is based on the segmentation model. There is no way to disable segmentation. Also, most of the basic data structures required by the processor to work in advanced mode (protected mode) are stored is specific segments (TS, CS, DS, ES, SS, etc.), which are accessed through the global or local descriptor tables (GDT/LDT). These tables contain entries called segment descriptors.

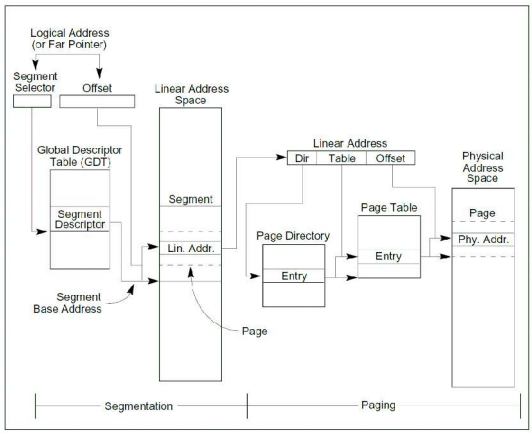


Illustration 1 Address translation

Addresses emitted by a program are called “logical address”. A logical address consists of a segment selector and an offset. The segment selector among other things provides an offset into a descriptor table

(GDT or LDT). The segment descriptor specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a linear address in the processor's linear address space.

Next is the the information stored in the segment descriptor regarding memory protection:

– Descriptor privilege level (DPL): Determines minimum processor privilege level required to access this segment.

– Executable, read/write: For segments that contain code or data.

Paging

When paging is enabled, lineal addresses are translated to physical addresses the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location, see Illustration 1. Page tables is organised in two levels[1]: Page directory (first-level) and page table (second-level).

Protection information for pages are managed by two flags in a page-directory or page-table entry: the read/write flag and the user(ring-level 3)/supervisor(ring-level 0-2)[2] flag.

When the processor is in supervisor-mode and the WP flag in register CR0 is clear (its state following reset initialisation), all pages are both readable and writable (write-protection is ignored). When the processor is in user-mode, it can write only to user-mode pages that are read/write accessible. User-mode pages which are read/write or read-only are readable; supervisor-mode pages are neither readable nor writable from user mode.

When paging is enabled, the processor evaluates segment protection first, then evaluates page protection.

Specific compatibility mode (WP bit)

Write Protect (bit 16 of CR0) inhibits supervisor-level procedures from writing into user-level read-only pages when set; allows supervisor-level procedures to write into user-level read-only pages when clear. This flag facilitates implementation of the copy on write method of creating a new process (forking) used by operating systems such as UNIX.

Changing the state of the WP bit do not cause any collateral overhead (TLB invalidation), therefore, it can be used to implement simple memory protection in supervisor-mode.

ARM

This architecture defines paging and a simple memory protection using memory regiones (ranges).

Paging

Page entries are organised in a two level page table directory. Protection information for

1 The original and default configuration. IA32 also defines three level page tables to access more than 4GB.

2 Note that paging mechanisms consider privileged-mode is ring levels 0-2 and user-mode is ring-level 3 only. Unlike segmentation mechanism which considers privileged-mode only ring-level 0 and the rest is user-mode.

pages descriptors has two bits called AP (Access Permission bits), and each page belogns to a “domain” (explained latter) . This AP bits control how page can be used depending on the execution mode (privileged or user):

AP (bits)	Privileged	User
00	Depends on other system config options	
01	Read/Write	No access
10	Read/Write	Read Only
11	Read/Write	Read/Write

Each individual page is assigned to a “domain”. There are 16 different domains. Access to each domain is controlled by a two bit field in the Domain Access Control Register (DACR 32bits register).

- (00) No Access: Any access generates a domain fault.
- (01) Client: Accesses are checked the AP bit in the section or page descriptor bits.
- (11) Manager: Accesses are not checked against the access permission bits in the AP bit in the section or page descriptor bits, so a permission fault cannot be generated.

The WP bit of the IA32 architecture may be simulated changing the corresponding domain bits of the DACR register.

Protection regions

There are eight different regions. Each region defines ranges of physical memory and the protection access permissions:

AP (bits)	Privileged	User
00	No access	No access
01	Read/Write	No access
10	Read/Write	Read Only
11	Read/Write	Read/Write

Chapter 4. Memory protection in Linux

Linux

Linux uses the basic flat model, where the operating system and application programs have access to a continuous, unsegmented address space. All segment registers points to the same segment descriptor. Each segment has full access to the whole memory space.

Virtual memory (and memory protection) is implemented through paging. Each process has its own page directory, but shares the kernel space which is located in the fourth gigabyte. Pages located below 3 gigabyte are user-mode accessible, and pages above the 3 gigabyte are marked as supervisor pages.

Linux enables the WP flag to implement the copy-on-write feature.

RTLinux

No memory protection at all implemented. Any thread has complete access to all address space.

RTLinux Stand-alone

Rtl-SA provides three different memory schemes:

1. Flat memory space with no protection. This is the original RTLinux model, where all the code (RTLinux executive and application) is compiled and linked as any conventional program. There is no overhead when a system call is invoked.

   Paging is not enabled, and segmentation is initialised to the basic flat model.

2. RTLinux executive protection. This model protects the RTLinux executive against write access from application threads. Application threads are not protected among them.

   Paging is enabled. Only one single page directory is initialised, and logical pages are mapped into the same physical pages. Pages that contain the Rtl-SA executive are marked as read-only pages (write is not allowed), and the rest of the pages are marked as read and write. Before RTLinux code is executed the WP flag is cleared (full access) and on return to application code the WP flag is set (executive pages are protected).

3. Context memory protection. The execution model has been augmented to support several protected execution contexts. In each context lives one or more threads.

   This memory protection scheme works as the previous one but with the addition of a page directory per context and the corresponding page directory change on the context switch between threads of different contexts.

In order to keep compatibility with standard RTLinux, applications are always executed in privileged-mode.

Chapter 5. Conclusions

Memory protection is a core operating system feature. The memory protection scheme used (monolithic kernel, memory protection inside the kernel, user application model, etc.) greatly determines the implementation of the operating system.

Changing the memory protection scheme of an already implemented system is not an easy task. There are two ways to change (or add memory protection):

– Rewrite the memory manager code and also some parts (servers, drivers, etc.) of the operating system to adapt them to a new communication mechanism.

– If we are lucky and processor has some facilities not used the current implementation, then it may be possible to use them to add the required mechanisms.

Since standard RTLinux completely delegates memory management to Linux, it has been

possible to implement in the RTLinux Stand-alone version several protection mechanism in a transparent way. RTL-SA Memory management code has been written from scratch.

We think that it is possible to achieve complete memory protection in standard RTLinux (when executed jointly with Linux) using specific IA32 features. Some preliminary results are encouraging.