

Bar-Ilan University

Memory Management Extension to Kernel Mode
Programming

Eliad Lubovsky

Submitted in partial fulfillment of the requirement for the
Master's Degree in the Department of Computer Science
Bar-Ilan University

Ramat Gan, Israel

2005

Acknowledgments

I would like to thank Dr. Joel Isaacson who was my supervisor. His patience and ongoing guidance is what ensured the completion of this thesis. Thank you for being my mentor, guide and friend.

Table of Contents

| | | |
|-----------|---|----|
| 1 | Abstract | 7 |
| 2 | Acronym | 9 |
| 3 | Introduction | 10 |
| 3.1 | GNU/Linux Operating System | 10 |
| 3.1.1 | Linux Software Architecture | 10 |
| 3.1.2 | Memory Management | 11 |
| 3.1.2.1 | Memory Addressing | 11 |
| 3.1.2.2 | Segmentation In Hardware | 11 |
| 3.1.2.3 | Physical Memory | 12 |
| 3.1.2.4 | Page Table Management | 13 |
| 3.1.2.4.1 | Page Directory | 14 |
| 3.1.2.4.2 | Page Table Entry | 15 |
| 3.1.2.4.3 | Page Table Handling | 15 |
| 3.1.2.4.4 | Kernel Page Table | 15 |
| 3.1.2.4.5 | Translation Lookaside Buffers (TLB) | 16 |
| 3.1.2.5 | Physical Page Allocation | 17 |
| 3.1.2.5.1 | Data Structure | 17 |
| 3.1.2.5.2 | Requesting and Releasing Page Frames | 17 |
| 3.1.2.6 | Noncontiguous Memory Area Management | 18 |
| 3.1.2.6.1 | Virtual Memory Areas | 18 |
| 3.1.2.6.2 | Allocating and Freeing A Noncontiguous Area | 19 |
| 3.1.3 | Processes | 19 |
| 3.1.3.1 | Process Descriptor | 20 |
| 3.1.3.2 | Process Address Space | 21 |
| 3.1.3.3 | The Page Fault Handler | 22 |
| 3.1.3.4 | Kernel Threads | 22 |
| 3.1.3.5 | Task Management and the Task State Segment | 22 |

| |
|-------------------|
| Table of Contents |
|-------------------|

| | |
|--|----|
| 3.1.3.5.1 Task Gate Descriptor | 24 |
| 3.1.3.5.2 Task Switching | 24 |
| 3.1.4 Interrupts and Exceptions | 24 |
| 3.1.4.1 Exception Classification | 25 |
| 3.1.4.2 Exception Types | 25 |
| 3.1.4.3 Task Restart | 26 |
| 3.1.4.4 Interrupt Descriptor Table | 26 |
| 3.1.4.5 Handling Interrupt and Exceptions | 27 |
| 3.1.4.5.1 Exception or Interrupt Handlers Procedures | 27 |
| 3.1.4.5.2 Interrupt Tasks | 28 |
| 3.1.5 System Calls | 30 |
| 3.2 Linux as A Real Time Operating Systems | 31 |
| 3.2.1 Kernel Response Time | 31 |
| 3.2.2 Real Time Scheduling | 31 |
| 3.2.3 Techniques To Improve Kernel Response Time | 32 |
| 3.3 Restricted Memory Management in Kernel Environment | 33 |
| 3.3.1 The Kernel Mode Stack Problem | 34 |
| 3.3.2 Micro Kernels | 34 |
| 4 Body of Work | 36 |
| 4.1 Design | 36 |
| 4.2 Resources | 40 |
| 4.3 Implementation | 40 |
| 4.4 Testing Methods | 52 |
| 4.4.1 Functionality Test | 52 |
| 4.4.2 Benchmark | 53 |
| 5 Discussion and Conclusions | 54 |
| 5.1 The Importance of The Research | 54 |

| |
|-------------------|
| Table of Contents |
|-------------------|

| | |
|---|----|
| 5.1.1 System Stability | 54 |
| 5.1.2 Powerful Kernel Entities | 55 |
| 5.1.3 Efficient Physical Memory Handling | 55 |
| 5.2 Comparison To Previous Research and Tools | 55 |
| 5.2.1 The Kernel Mode Linux Project | 55 |
| 5.2.2 Micro Kernels | 56 |
| 5.3 Future Directions | 57 |
| 5.3.1 Dynamic Kernel Mode Stack Size | 57 |
| 5.3.2 SMP Support | 57 |
| 5.3.3 Fixed Virtual Address Interval | 57 |
| 5.3.4 Comprehensive Paging in The Kernel | 58 |
| 6 Appendix | 59 |
| 6.1 Appendix A: Benchmark Results | 59 |
| 7 References | 65 |

Table of Figures

| | |
|---|----|
| Figure 1: Two-Level Page Table | 14 |
| Figure 2: Process Descriptor and Kernel Stack | 20 |
| Figure 3: Task Structure | 24 |
| Figure 4: System Architecture | 29 |
| Figure 5: Option A. New Kernel Mode Stack (with two physical pages) | 38 |
| Figure 6: Option B. New Kernel Mode Stack (with one physical page) | 39 |
| Figure 7: A Printout of The Faulting Virtual Address and a Page Table Trace of The vm_area Space That Belongs to The Stack (in a 16KB stack) ... | 53 |

Table of Source Code List

| | |
|--|----|
| Code 1: Allocate and Free The thread_info and The Kernel Mode Stack Area ... | 41 |
| Code 2: vmalloc_thread_info Implementation | 42 |
| Code 3: Map The Thread Info Area To The Page Table Entries | 44 |
| Code 4: Kernel Page Fault TSS | 46 |
| Code 5: Initial TSS update | 47 |
| Code 6: Find The Thread Info Virtual Area | 49 |
| Code 7: Expanding The Kernel Mode Stack | 50 |
| Code 8: The Kernel Page Fault Handler | 51 |

1 Abstract

This research deals with memory management in the Linux kernel. Memory which is one of the fundamental resources in the system architecture, is divided to fixed sized frames. Frames are sequentially numbered and enable the CPU to reference a specific address. This address is called the physical address. Another type of address used by the operating system is a virtual address. A virtual address specifies additional memory area to the physical one. Running on IA-32, the virtual address space is split into two parts, the user space part and the kernel space part. The split is determined by the value of `PAGE_OFFSET` which is at 3GB (on the x86). The first virtual part that is from 0 to 3GB is available for user space applications while the remaining 1 GB is mapped to the kernel. Contrary to the kernel mode where the virtual address space remains constant, in user mode, each process has its own virtual address space and the offset between the virtual addresses to the physical addresses is not predefined. Converting a virtual address to a physical one is done by using page tables with two or three conversion levels.

The operating system allocates memory for every new process in kernel space and in user space. In user space the virtual memory is not satisfied immediately. The virtual address space is reserved for the process and physically allocated when the CPU triggers a page fault exception. The page fault handler checks the legality of the faulting address and if it is in order, a new physical page is allocated to satisfy the request. As opposed to user space, memory allocations in the kernel are satisfied immediately and are globally visible in the kernel address space.

The IA-32 hardware architecture provides protected rings from zero to three. The rings provide different protection levels for instructions running on the CPU, while ring three is the most protected level and ring zero is the most privilege level enabling to run all the CPU instruction set. Linux uses ring 0 for kernel environment and ring 3 for user space.

The memory area in the user space and in the kernel space is divided into segments which altogether provide the total memory area for the process. The segments include: code segment, data segment, stack segment and additional system segments.

The IA-32 hardware architecture demands a stack segment for each protection ring used by the operating system. In the user space, the stack is part of the virtual area belonging to the process. The stack can grow according to the memory requirements of a process, using the page fault exception mechanism. In the kernel space the memory that is allocated for the stack is small sized and fixed (in kernel version 2.6 it is set to 4 or 8KB at compile time). As opposed to user space, memory corruptions in the kernel stack are not supervised and may cause system failure.

The design of Linux is based on the Unix operating system that is a non real time kernel, however, there is a need to run applications that require deterministic response time. A better predictive response time may be achieved while running in the kernel control path. Running in kernel mode is more difficult and restrictive than in user space. Some projects try to exploit the preferable response time in the kernel environment and the benefits of the user space environment regarding memory management and the simplicity of programming. For example, the kernel mode Linux project (KML) enables to run user processes in kernel mode and the RTLinux is based on a micro kernel.

The importance of this research is apparent in two main issues: The first is stabilizing the operating system by preventing memory corruption in the fixed sized kernel mode stack. The second issue is to provide a better response time for applications, by running in kernel space while using fundamental normal processes facilities.

The solution purposed in this research is to add the page fault exception mechanism to the kernel mode stack. The Linux kernel is an unstructured monolithic kernel (a kernel that packs all the system utilities into a single kernel). The main problem when handling page fault exceptions in a monolithic kernel is the stack starvation problem. The solution found is based on the IA-32 hardware task management facility that is provided to support process management for kernels.

2 Acronym

GDT: Global Descriptor Table, memory management table that contains segment descriptors.

IDT: Interrupt Descriptor Table, a system table that contains information used in handling interrupts. The table associates each interrupt or exception vector with the corresponding handler address.

LDT: Local Descriptor Table, memory management table that contains segment descriptors. Each process may have its own LDT.

LDTD: Local Descriptor Table Descriptor, segment descriptor containing an LDT.

MMU: Memory Management Unit, a hardware device used to perform virtual to physical address translation.

PGD, PMD and PTE: Page Global Directory, Page Middle Directory and Page Table Entry respectively, forms the page table entries.

PSE: Page Size Extension, IA32 system flag, enables large page size up to four MB of pages.

SMP: Symmetric Multi Processor/Processing.

TLB: Table Lookaside Buffer, on-chip cache that stores the most recently used virtual to physical address translations.

TSS: Task State Segment, a system segment that saves the processor state information needed to restore a task.

TSSD: Task State Segment Descriptor, indicate the segment descriptor refers to a TSS. Used only in the GDT.

UP: Uniprocessor (single CPU) system.

3 Introduction

3.1 GNU/Linux Operating System

The GNU Project was launched in 1984 to develop a complete UNIX style operating system which is free software: the GNU system [GNU]. Variants of the GNU operating system widely used, use Linux as the kernel of the operating system. The Linux kernel was originally created by Linus Torvalds with the assistance of developers around the world. Linus who was a student in the University of Helsinki in Finland, started his work in 1991. The kernel which is the heart of the operating system was released under the GNU General Public License , and its source code is freely available to everyone.

3.1.1 Linux Software Architecture

In addition to all programs included in the GNU/Linux operating system, the main system is the kernel. The Linux kernel is a large system that is responsible for processes, memory and hardware device management. The kernel is divided into seven major subsystems as follows [BHB99]:

- The Process Scheduler – enables the system to support multitasking.
- The Memory Manager – manages physical and virtual memory.
- The File System – provides access to hardware devices.
- The Network Interface – manages network protocols to communicate with other computers using different types of network hardware.
- The Inter-Process Communication (IPC) – a subsystem that allows users processes to communicate with other processes on the same computer.
- The Initialization – responsible to initialize the system with user configuration settings.
- The Library – a subsystem that contains routines which are used throughout the kernel, including: string manipulations, kernel command line and more.

3.1.2 Memory Management

This chapter discusses how the Linux kernel handles memory using available IA-32 hardware features, it describes addressing techniques, caching and memory allocation interfaces.

3.1.2.1 Memory Addressing

There are three kinds of addresses commonly used by the operating system and the IA-32 hardware, in memory addressing. These include: logical, linear and physical addresses. A logical address is part of the machine language instruction. It specifies the address of an operand and or of an instruction, and consists of a segment identifier and an offset. A linear address is a single 32-bit unsigned integer used to address up to 4GB of memory cells. A physical address is also represented as 32-bit unsigned integer, it is used to address memory cells included in memory chips and is actually written to address lines on the bus. The CPU control unit converts the logical address to a linear address which is then converted by the paging unit to a physical address [BC03, IAB97].

3.1.2.2 Segmentation In Hardware

The IA-32 provides two different ways to perform address translation called real mode and protected mode. The protected mode is the normal mode of operation where as the real mode is mainly used for system bootstrap [BC03]. While operating in protected mode, all memory access pass through either the global descriptor table (GDT) or the local descriptor table (LDT). These tables contain entries called segment descriptors. The segment descriptor represents segments describing the base address of the segment, access rights, type, and usage information. The segment descriptor types widely used are: code segment descriptor, data segment descriptor, task state segment descriptor (TSSD) and local descriptor table descriptor (LDTD). Each segment descriptor has a segment selector associated with it providing an index into the GDT or LDT. The processor

provides segmentation registers to hold the segment selectors called cs, ss, ds, fs, and gs (segment selector is a 16bit field which forms the segment identifier in the logical address.) [BC03, IAB97].

The code, data and stack segments make up the execution environment of a program or a procedure. The system architecture also defines two system segments: the task state segment (TSS) and the LDT. In addition, the system architecture defines a set of special descriptors called gates. The gates provide protected gateways to system procedures and handlers that operate at different privilege levels [IAB97].

3.1.2.3 Physical Memory

The system's memory is broken up into fixed sized chunks called page frames. The kernel treats these physical page frames as the basic unit of memory management. The MMU which is the hardware unit that performs virtual to physical address translations, manages the system's page table using a page-sized granularity. Although the processors smallest addressable unit is usually a word, in terms of virtual memory, pages are the smallest unit that matter [BC03, Gor04]. The kernel represents each physical page frame on the system with a struct page structure, and all the structures are kept in a global mem_map array. The page struct is used to keep track of a page frame status. It is declared in <linux/mm.h> as:

```
struct page {
    page_flags_t flags;
    atomic_t _count;
    atomic_t _mapcount;
    unsigned long private;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
    void *virtual;
};
```

The relevant fields will be described:

The `flags` field describes the status of the page and includes whether the page is dirty or locked in memory. All flags are declared in `<linux/page-flags.h>`. The `_count` field stores the reference count to the page. If `_count` drops to zero no process is using the page and it may be freed. The `virtual` field is the page's virtual address.

The kernel uses the `page` structure to keep track of all the pages in the system and it can determine whether pages are freed or not. If a page is not free, the kernel needs to recognize who owns the page. Each frame is hardware protected through the flags included in the Page Table Entry (PTE) that points to it. When a page fault exception occurs, the kernel can directly determine the memory allocation unit associated with the page.

3.1.2.4 Page Table Management

Although applications operate on virtual memory addresses that are mapped to physical addresses, processors operate directly on those physical addresses [BHB99]. The conversion from linear to physical addresses is performed by using the page tables. Linux layers the machine independent/dependent layer using the concept of a three-level page table in the architecture independent code, even if the underlying architecture does not support it. If this occurs, one of the page table levels folds back onto the greater level. For example, on the x86, only two page table levels are available. The Page Middle Directory (PMD) is defined to be of size one and “folds back” directly onto the Page Global Directory (PGD) which is optimized out at compile time [Gor04]. In most architectures (including the Intel x86 processors) the page table lookups are handled by hardware using a device called MMU.

3.1.2.4.1 Page Directory

Each process including the kernel has its own Page Global Directory (PGD) which is a physical page frame containing an array of `pgd_t` type, which is usually an unsigned

integer (defined in <asm/page.h>). The active entries in the PGD table points to a page frame containing a second level of the page table called Page Middle Directory (PMD) of type `pmd_t`. The entries in the PMD point to a third level of page frames containing the Page Table Entries (PTE) of type `pte_t`, which finally points to page frames containing the actual user data. Any given linear address may be broken up into parts to yield offsets within these three page table levels and an offset within the actual page [Gor04]. A number of macros are provided in order to help break up the linear address into its component parts for each page table level, and to specify the length in bits of each level. In the following figure (figure 1) it is possible to see how the linear address splits into three parts in a two level page table.

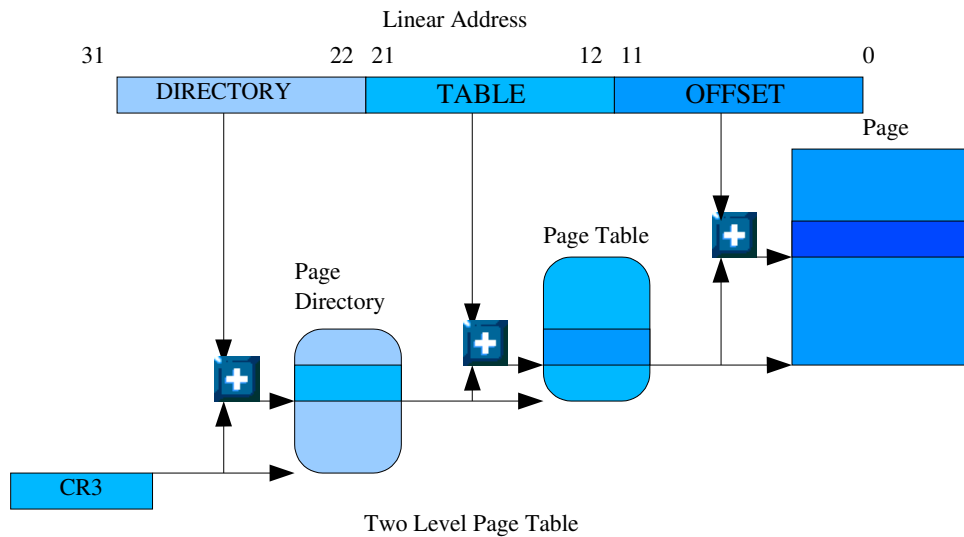


Figure 1: Two Level Page Table

3.1.2.4.2 Page Table Entry

The PTE entry is described by a structure type `pte_t` and consists of two parts. The first part points to the page frames containing the actual user data, and the second part is used for status bits. The status and protection bits describe various properties such as: if the page is resident in memory or swapped out, if the page is write or read enabled or whether it can be accessed from user space.

3.1.2.4.3 Page Table Handling

Linux provides a few macros and functions to handle the page tables. The first set of functions are used for the navigation and examination of the page table entries. For example, `pgd_offset()`, `pmd_offset()` and `pte_offset()` enable to 'walk' the page table and reach the physical page pointed by a specific virtual address. The second set of functions are used to translate and set the page table entries as `mk_pte()` and `set_pte()`. These functions update the page table entries and form a new entry. The last set of functions deal with the allocation and freeing of page table using functions as `pgd_alloc()` and `pgd_free()`.

3.1.2.4.4 Kernel Page Table

The kernel initializes its own page tables using two phases of initialization. In the first phase the kernel creates page tables for 8MB which is used to install itself in RAM. The second phase initializes the rest of the page tables. The PGD which is contained in a static array, is defined at compile time and is called the `swapper_pg_dir`. The page entries are established by two variables `pg0` and `pg1`. The two entries point to the 8MB by exploiting the extended paging feature (of the Intel Pentium processors) which enables 4MB page frames instead of 4KB.

3.1.2.4.5 Translation Lookaside Buffers (TLB)

The IA-32 processor stores the most recently used page-directory and page-table entries in on-chip caches called translation lookaside buffers or TLBs. Most paging is performed using the contents of the TLBs. Bus cycles to the page directory and page tables in memory are performed only when the TLBs do not contain the translation information for a requested page. The TLBs are inaccessible to application programs and tasks (privilege level greater than 0); that is, they cannot invalidate TLBs. Only the operating system or executive procedures running at privilege level of 0 can invalidate TLBs or selected TLB entries. Whenever a page-directory or page-table entry is changed (including when the present flag is set to zero), the operating-system must immediately invalidate the corresponding entry in the TLB so that it can be updated the next time the entry is referenced [IAP99].

All (non global) TLBs are automatically invalidated any time the CR3 register is loaded. The CR3 register is usually loaded by executing a task switch, which automatically changes the contents of the CR3 register. It is also possible to invalidate a specific page-table entry in the TLB using INVLPG instruction. Normally, this instruction invalidates only an individual TLB entry; however, in some cases, it may invalidate more than the selected entry and may even invalidate all of the TLBs. The page global enable (PGE) flag in register CR4 and the global (G) flag of a page-directory or page-table entry (bit 8) can be used to prevent frequently used pages from being automatically invalidated in the TLBs on a task switch or a load of register CR3. When the processor loads a page-directory or page-table entry for a global page into a TLB, the entry will remain in the TLB indefinitely. The only way to deterministically invalidate global page entries is to clear the PGE flag and then invalidate the TLBs or to use the INVLPG instruction to invalidate individual page-directory or page-table entries in the TLBs [IAP99].

3.1.2.5 Physical Page Allocation

Linux uses a strategy called the buddy system algorithm in order to allocate contiguous physical page frames. This technique tries to keep track of existing blocks of free contiguous page frames trying to avoid external fragmentation. The advantages of using contiguous page frames are gaining performance with less average memory access times and the ability to handle hardware devices that use DMA buffers.

3.1.2.5.1 Data Structure

The Buddy system algorithm relies on the following data structure: the `mem_map` array, the `free_area_t` structure and a binary bitmap array. The allocator maintains blocks of free pages where each block is a power of two number of pages. The exponent for the power of two sized blocks is referred to as the order. An array of `free_area_t` structures are maintained for each order that points to a linked list of blocks of pages that are free.

3.1.2.5.2 Requesting and Releasing Page Frames

The following API functions are available for allocating physical pages. All API functions use the core `__alloc_pages()` function and are declared in `include/linux/gfp.h`.

`alloc_page(unsigned int gfp_mask)` - Allocate a single page and return a struct address.

`alloc_pages(unsigned int gfp_mask, unsigned int order)` - Allocate 2^{order} number of pages and return a struct page.

`get_free_page(unsigned int gfp_mask)` - Allocate a single page, zero it and return a virtual address.

`__get_free_page(unsigned int gfp_mask)` - Allocate a single page and return a virtual address.

`__get_free_pages(unsigned int gfp_mask, unsigned int order)` - Allocate 2^{order} number of pages and return a virtual address.

`__get_dma_pages(unsigned int gfp_mask, unsigned int order)` - Allocate 2^{order} number of pages from the DMA zone and return a struct page.

The parameter `gfp_mask` is a set of flags that determine how the allocator will behave. Some of the flags available include `GFP_ATOMIC`, `GFP_KERNEL`, `GFP_USER`, `GFP_KSWAPD` and `GFP_HIGH_USER`. The second parameter is the `order` - allocations are always for a specified order, 0 in the case where a single page is required.

As mentioned, all API functions use the `alloc_pages()` (`mm/page_alloc.c`) which is the heart of the allocator. This function checks the number of available pages and if necessary wakes the `kswapd` thread to begin freeing pages to a free storage. Freeing pages is done via a simple API providing functions to free allocated pages. The API include `__free_pages()`, `__free_page()` and `free_page()`, the first two functions takes a struct page as a parameter while the last takes a virtual address.

3.1.2.6 Noncontiguous memory Area Management

Although it is preferable to map memory areas into sets of contiguous page frames both for cache related and memory access latency issues, it may cause a problem of external fragmentation. In addition to the buddy allocator algorithm which is used to allocate continues physical page frames, Linux provides a mechanism to allocate a non-contiguous physically memory that is contiguous in virtual memory [Gor04]. The noncontiguous area is allocated using the `vmalloc()` mechanism that uses the normal physical page allocator. The size of the allocation aligned to the hardware page size and the virtual address space allocated is between `VMALLOC_START` and `VMALLOC_END`.

3.1.2.6.1 Virtual Memory Areas

Each noncontiguous memory area is associated with a descriptor of type struct

`vm_struct` declared in `<linux/vmalloc.h>` as follows:

```
struct vm_struct {
    void                *addr;
    unsigned long       size;
    unsigned long       flags;
    struct page         **pages;
    unsigned int        nr_pages;
    unsigned long       phys_addr;
    struct vm_struct    *next;
};
```

All area descriptors are linked together in a linked list via the `next` field. The address of the first element of the list is stored in the global `vmlist` variable. The fields in the `vm_struct` structure are used to track the virtual memory area. The two basic fields are the `addr` and `size` which is the starting address of the memory block, and the size of the virtual area in bytes respectively.

3.1.2.6.2 Allocating and Freeing a Noncontiguous Area

The `vmalloc()` function is provided to allocate a noncontiguous memory area in the kernel virtual address space. It takes a single parameter `size` which is rounded up to the page frame size and returns a linear address for the new allocated area. In the first step `vmalloc()` uses the `get_vm_area()` function which searches the `vmlist` linked list to find a region large enough to satisfy the request. The next step is allocating the physical pages according to the requested size. The last step is allocating and mapping the virtual area and the physical pages to the kernel page table. Freeing the virtual area is satisfied using the function `vfree()` which is doing the opposite of `vmalloc()`. It linearly searches the list of `vm_struct` to free the virtual region, unmaps it from the kernel page tables and then deallocates the corresponding physical pages.

3.1.3 Processes

A process is one of the fundamental abstractions in operating systems usually defined as

an instance of a program in execution. A process includes a code and data sections, a set of resources as open files, an address space and threads of execution.

3.1.3.1 Process Descriptor

In order to manage processes Linux stores all process descriptors in a circular doubly linked list called the task list. The descriptor that is of `task_struct` type contains all the information related to a single process. The process descriptor is pointed by a structure of `thread_info` struct type. The `thread_info` is stored at the bottom of the kernel stack of each process as shown in the following figure (figure 2). This allows obtaining the process descriptor of the process currently running on the CPU from the value of the stack pointer, without any memory referencing. This is done by the

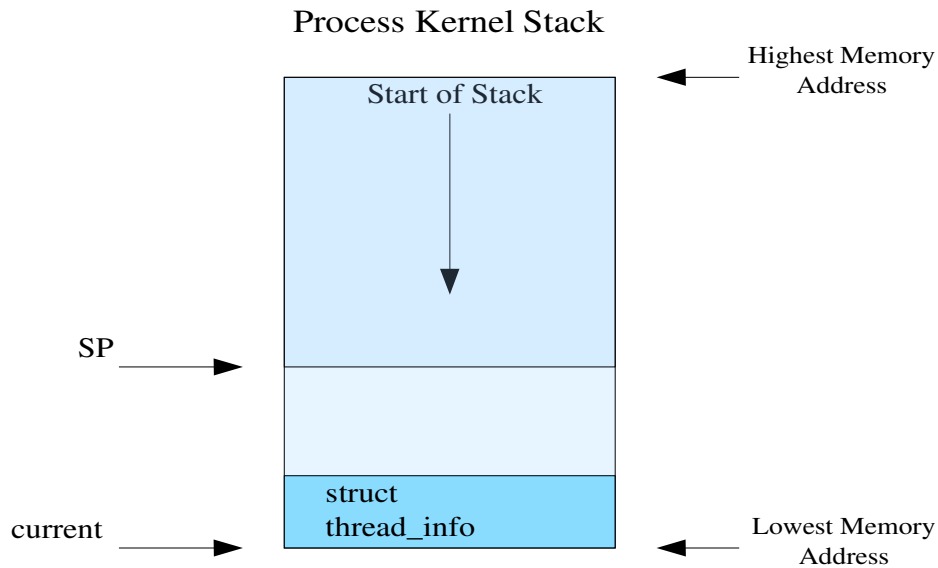


Figure 2: Process Descriptor and Kernel Stack

current macro which produces some assembly instruction as follows:

```
__asm__("andl %%esp,%0; ":"=r" (ti) : "0" (~(THREAD_SIZE - 1)));
```

After executing `current`, `ti` contains a pointer to the process descriptor of the currently process running on the CPU. The size of the kernel stack which is determined by the value of `THREAD_SIZE` is fixed and configured at compile time to 4 or 8 KB.

All processes in Linux are created by invoking the `do_fork()` system call defined in `kernel/fork.c`. the `do_fork()` function calls `copy_process()` to create a new process as a copy of the old one and then starts the process running. `copy_process()` is responsible for allocating the new area to serve the kernel mode stack and the `thread_info` structure by invoking the `alloc_task_struct()` macro. After invoking all stages a complete child process in the runnable state is obtained. Now it is up to the scheduler to decide when to give the CPU to this child.

3.1.3.2 Process Address Space

Linux as a virtual memory operating system manages the process address space by enabling each process to have its own virtual address space, and presents the range of addresses within this space the process allows to use. The process can only access a memory in a valid area, described by intervals of legal addresses called memory areas.

The kernel manages the linear address space by splitting it into two parts, the user space portion that is up to `PAGE_OFFSET` (the default is `0xc0000000`) and changes every context switch, and the kernel portion which is the remaining memory (default 1GB) that remains constant [Gor04].

The kernel stores all the information related to a process address space with a data structure called the memory descriptor represented by a struct `mm_struct` which is defined in `<linux/sched.h>`. Some of the information contained in the `mm_struct` fields includes the number of processes using the address space and a reference to `mmap`. `mmap` is a head of a linked list of all virtual regions belonging to the address space. The type is

of `struct vm_area_struct` defined in `<linux/mm.h>`. The memory areas stored in the `vm_area_struct` are often called virtual memory areas or VMA's in the kernel. The VMAs represent multiple types of memory areas, for example memory-mapped files or the process's user-space stack.

3.1.3.3 The Page Fault Handler

In user mode memory allocations are not satisfied immediately as in the kernel mode, and pages related to a process address space may be swapped out to a backing storage. This means that the process linear address space is not necessarily resident in memory but just reserved with the `vm_area_struct`. When a page fault is triggered it is handled by the `do_page_fault()` handler function, which basically compares the linear address that caused the page fault against the memory regions of the current process. If the exception handler has decided that it is a valid page fault, it allocates a new page, if necessary retrieves its data from a backing storage and resumes the execution for the process. The parameters passed to the handler include the processor registers and exception error codes. Using the input parameters (on the IA-32) the handler can retrieve the faulting linear address (saved in the CR2 control register) and determine exception details as if it occurred in user mode or kernel mode.

3.1.3.4 Linux Kernel Threads

Kernel threads are standard processes that exist solely in kernel space and are used to perform background operations as flushing disk caches, swapping out unused page frames and so forth. The differences between kernel threads and normal processes are in the following ways: each kernel thread executes a single specific kernel function, they run only in kernel mode and they do not have an address space (their pointer to the active `mm_struct` is `NULL`).

3.1.3.5 Task Management and The Task State Segment

When operating in the IA-32 protected mode, all processor execution takes place from within a task. A task is made up of two parts: a task execution space and a task-state segment (TSS). The task execution space consists of a code segment, a stack segment and one or more data segments. If using the processor's privilege-level protection mechanism, the task execution space also provides a separate stack for each privilege level. The TSS which is a hardware architecture segment, specifies the segments that make up the task execution space and provides a storage place for task state information (store hardware context, refer to figure 3). If paging is implemented for the task, the base address of the page directory used by the task is loaded into CR3 control register.

Using the TSS the hardware provides a mechanism for saving the state of a task, for dispatching tasks for execution, and for switching from one task to another [IAP99].

The TSS has static and dynamic fields. The static fields include for example the LDT segment selector, the CR3 register and the three protection ring levels (0,1 and 2) as can be seen in the following figure (figure 3). The dynamic fields include fields that may change every context switch for example the EIP, the general purposed registers and the previous task link field [IAP99].

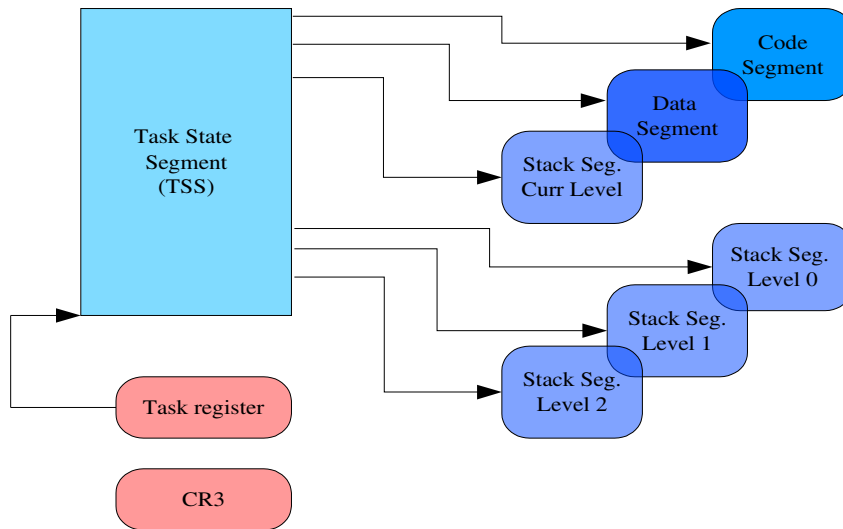


Figure 3: IA-32 Task Structure

3.1.3.5.1 Task Gate Descriptor

A task-gate descriptor provides a protected reference to a task and can be placed in the GDT, an LDT, or the IDT. The TSS segment selector field in a task-gate descriptor points to a TSS descriptor in the GDT. A task can be accessed through a task-gate descriptor which provides the ability for an interrupt or exception to be handled by an independent task. When an interrupt or exception vector points to a task gate, the processor switches to the specified task.

3.1.3.5.2 Task Switching

There are several cases in which the processor transfers execution to another task. One of the cases is of an interrupt or exception vector that points to a task-gate descriptor in the IDT. Some of the operations performed when switching to a new task are: obtaining the

TSS segment selector for the new task (in case using a task gate descriptor in the IDT the TSS found from the task gate), checking security, saving current task state, loading the task register with the segment selector and descriptor for the new task's TSS, loading the new task's state from its TSS into the processor and starting execution of the task.

3.1.4 Interrupts and Exceptions

Interrupts are special electrical signals alerts a sequence of instructions executed by a processor. The signals are generated by hardware circuits both inside and outside the CPU. Both interrupts and exceptions are forced transfers of execution from the currently running program or task to a special procedure or task called a handler. Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero, protection violations, page faults, and internal machine faults. [BC03, IAP99] Interrupts provide a way to divert the processor to code outside the normal flow of control. When an interrupt signal arrives, the CPU must stop what it is currently doing and switch to a new activity. It does this by saving the current value of the program counter in the kernel mode stack and by placing an address related to the interrupt type into the program counter. This chapter describes interrupts and exceptions and will focus on the exceptions-handling mechanism.

3.1.4.1 Exception Classification

Exceptions are generated when the CPU detects an anomalous condition while executing an instruction. The exceptions are divided into three groups depending on the way they report and whether the instruction that caused the exception can be restarted. The exceptions types are:

Faults - An exception that can generally be corrected and allows the program to be restarted with no loss of continuity. The EIP value of the instruction that caused the fault is saved in the kernel mode stack and can be resumed when the handler terminates.

Traps – Mainly used for debugging purposes, triggered when there is no need to re-execute the instruction that was terminated. The return address for the trap handler points to the instruction to be executed after the trapping instruction.

Aborts – Serious errors caused by hardware failures or by invalid values in system tables. Aborts cause the the effected process to terminate, the precise location of the faulting instruction is not always reported.

3.1.4.2 Exception Types

Depending upon the specific CPU model there are roughly twenty different exceptions. The kernel provides an exception handler for each one. The more relevant exceptions are listed according to their type, number and a brief description:

- Double fault: abort type exception #8, can happen when the CPU detects a second exception while trying to call the handler for a prior exception. If the CPU cannot handle the two exceptions serially it raises this exception.
- Invalid TSS: fault type exception #10, the CPU has attempted a context switch to a process having an invalid Task State Segment.
- Stack segment: fault type exception #12, caused when the instruction attempts to exceed the stack segment limit.
- General protection: fault type exception #13, one of the protection rules that has been violated.
- Page fault: fault type exception #14, as described, caused when the addressed page is not present in memory.

3.1.4.3 Task Restart

To allow the restart of a program or a task following the handling of an exception or an interrupt, all exceptions except aborts are guaranteed to report the exception on a precise instruction boundary, and all interrupts are guaranteed to be taken on an instruction

boundary. For fault-class exceptions, the return instruction pointer that the processor saves when it generates the exception, points to the faulting instruction. Therefore, when a program or task is restarted following the handling of a fault, the faulting instruction is restarted (re-executed). Restarting the faulting instruction is commonly used to handle exceptions that are generated when access to an operand is blocked. The most common example of a fault is a page-fault exception (#PF) that occurs when a program or task references an operand in a page that is not in memory. When a page-fault exception occurs, the exception handler can load the page into memory and resume execution of the program or task by restarting the faulting instruction. To insure that this instruction restart is handled transparently to the currently executing program or task, the processor saves the necessary registers and stack pointers to allow it to restore itself to its state prior to the execution of the faulting instruction.

3.1.4.4 Interrupt Descriptor Table

The interrupt descriptor table (IDT) is a system table which associates each exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt. The IDT format is similar to that of the GDT or LDT, each entry in the table corresponds to an interrupt or an exception and consists of an 8-byte descriptor. To form an index into the IDT, the processor scales the exception or interrupt vector by eight. The table may contain up to 256 descriptors, although descriptors are required only for the interrupt and exception vectors that may occur. The IDT contains three types of descriptors: Interrupt gate descriptors, Task gate descriptors and Trap gate descriptors.

3.1.4.5 Handling Interrupt and Exceptions

After executing an instruction, the CPU registers contain the address of the next instruction. Before dealing with that instruction the control unit checks whether an

interrupt or exception has occurred. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the IDT. If the index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler. If index points to a task gate, the processor executes a task switch to the exception- or interrupt-handler task.

3.1.4.5.1 Exception or Interrupt Handlers Procedures

An interrupt gate or trap gate references an exception- or interrupt-handler procedure that runs in the context of the currently executing task. When the processor performs a call to the exception- or interrupt-handler procedure, it saves the current registers state on the stack. The registers include the CS and the EIP which provide the return instruction pointer for the handler. If an exception causes an error code to be saved, it is pushed on the stack after the EIP value. If the handler procedure is going to be executed at the same privilege level as the interrupted procedure, the handler uses the current stack. If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When a stack switch occurs, a stack pointer for the stack to be returned to is also saved on the stack. The segment selector and stack pointer for the stack to be used by the handler is obtained from the TSS for the currently executing task. The processor copies relevant registers, and error code information from the interrupted procedure's stack to the handler's stack. The processor does not permit transfer of execution to an exception- or interrupt-handler procedure in a less privileged code segment (numerically greater privilege level) than the CPL. An attempt to violate this rule results in a general-protection exception (#GP).

3.1.4.5.2 Interrupt Tasks

When an exception or interrupt handler is accessed through a task gate in the IDT, a task switch results. Handling an exception or interrupt with a separate task offers several

advantages:

- The entire context of the interrupted program or task is saved automatically.
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. If an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack. In the following figure (figure 4) the red line describes a task gate path from the IDT table to a separate TSS. In contrary, when using an interrupt gate we can see that the system uses the current TSS.

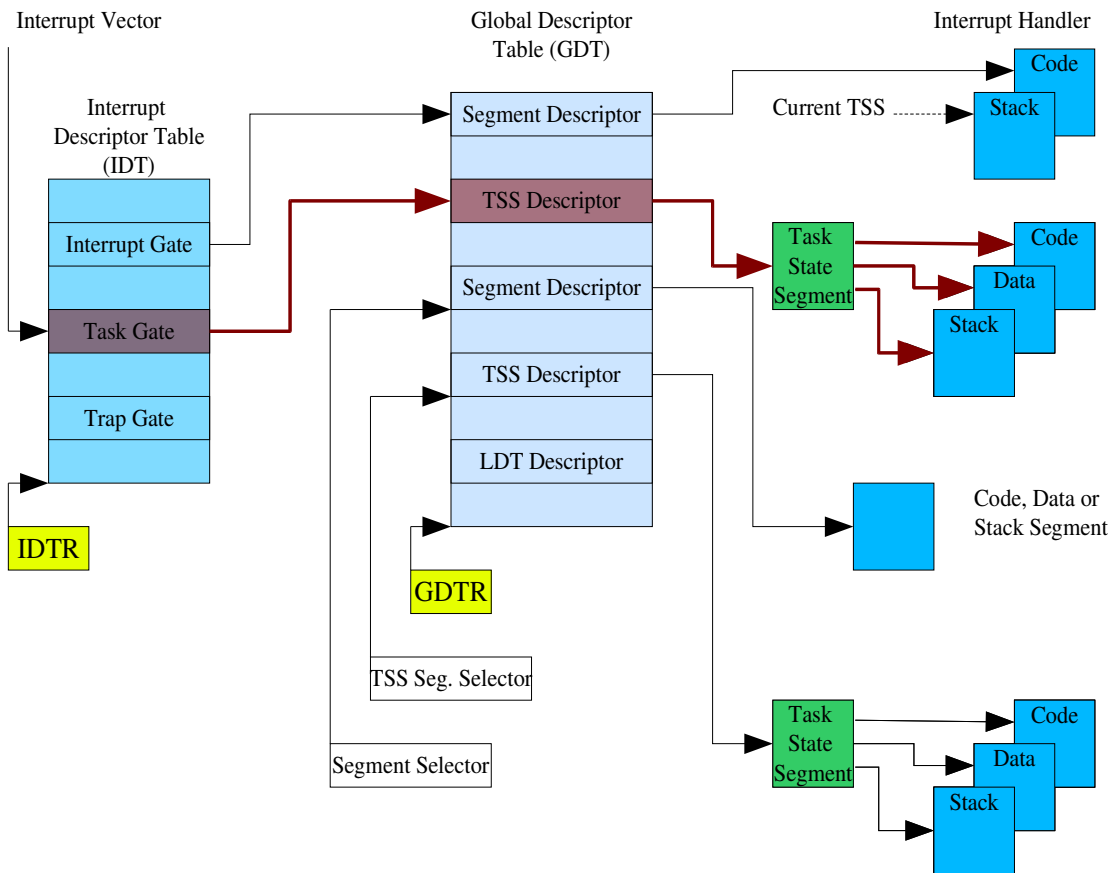


Figure 4: IA-32 System Architecture

The disadvantage of handling an interrupt with a separate task is that the amount of machine state that must be saved on a task switch makes it slower than using an interrupt gate, resulting in increased interrupt latency. A switch to the handler task is handled in the same manner as an ordinary task switch. The link back to the interrupted task is stored in the previous task link field of the handler task's TSS. If an exception caused an error code to be generated, this error code is copied to the stack of the new task. When exception- or interrupt-handler tasks are used in an operating system, there are actually two mechanisms that can be used to dispatch tasks: the software scheduler (part of the operating system) and the hardware scheduler (part of the processor's interrupt mechanism).

3.1.5 System Calls

Linux provides a set of interfaces for interacting between user space applications and the kernel by means of system calls. This is the only way for user space applications to divert execution to the kernel. All system calls are invoked by executing the `int $80` assembly instruction, and by passing a system call number (using the EAX register) to identify the system call. The kernel handle system calls by a system call handler performing the following operation: saving the relevant registers on the kernel mode stack, invoking the corresponding service routine and exiting the handler by means of `ret_from_sys_call()` function.

3.2 Linux as A Real Time Operating System

Linux is designed as a non-preemptive kernel, therefore, by its nature, it is not well suited for real-time applications that require deterministic response time. A better predictive response time can be achieved while running in kernel context (the kernel is the highest priority component in the system) using the Linux facilities of interrupt handlers, Softirq, Takslets, Work Queues and Kernel Threads. Although gaining critical response time, running code in kernel mode is difficult and restrictive [RL03, BC03].

3.2.1 Kernel Response Time

Kernel response time is the time elapsing from when an event occurs to the moment the process responds. The response time is related to the nature of the operating system. Multitasking operating systems either can be cooperative multitasking or preemptive multitasking. Linux provides for full preemptive multitasking in user space since its early days. Kernel preemptive multitasking (as of the last kernel series (2.6.x)) is partially possible, so long as the kernel is in a state in which it is safe to reschedule.

3.2.2 Real Time Scheduling

The Linux user processes scheduler provides the POSIX 1003.13 standard of real-time extension. The POSIX standard defines a “Multi-User Real-Time System Profile” which ensures that “Real-time” processes are always executed in a predictable order. Although Linux meets this standard the resultant task cannot be defined as real-time task, since kernel activity can block it. The real time scheduling uses the SCHED_FIFO (first in first out) and SCHED_RR (round robin) with fixed priorities. Gaining the CPU, SCHED_FIFO will never be preempted unless explicitly yielding the CPU, or a higher priority task becomes ready. The Linux real-time scheduling preempts tasks using SCHED_OTHER, which is the normal and default task priority [RL03, MB99, MBDPH00, CW02].

The lack of full kernel preemption in Linux presently means that long system calls can delay execution of a user process with high priority for relatively a long period of time. Not responding quickly to I/O events is not acceptable for applications that require predictable and low response times such as audio applications and software modem. For example, the DSP algorithms of software ADSL modem require quick response to I/O events, large memory areas, and high peak CPU consumption. The response time can be critical to the modem in order to keep a stable connection. A delay in receiving or transmitting physical layer samples may result in samples buffer overrun or under-run and may cause bad connections or modem retrains (physical disconnections). In addition, periodical hardware commands for controlling the VCXO (Voltage-Controlled Crystal Oscillator) clock deviation require accurate and as short a response time as possible to gain performance and avoid retrains [CW02].

3.2.3 Techniques To Improve Kernel Response Time

There are some techniques to improve the kernel response time. For hard-real-time applications it is possible to use RTLinux or RTAI. These systems use a nano kernel that runs Linux as its lowest priority execution thread. This thread is completely preemptive so real-time tasks are never delayed by non-real-time operations. Mechanisms for communication with the Linux kernel make use of all the powerful, non-real-time services of Linux [MBDPH99, KD00]. RTLinux and RTAI as solutions for hard-real-time applications are not by their nature a general solution for all problems. For the less restricted real-time applications, two patches of the vanilla Linux kernel were introduced in the last kernel series (2.4.x which was later incorporated into the current 2.6.x kernel.). The two kernel modifications are the low latency patch and the preemptive patch. The basic idea of these patches is to run the scheduler more often by finding places in the kernel code that are safe to preempt. These improvements are known as soft real time solutions meaning, the system does not meet each and every desired response time.

[CW02, AG00, BK04, RL00]

Another known solution to improve the kernel-programming environment is the KML (Kernel Mode Linux) project. KML has a mechanism that makes it possible to execute user programs in kernel mode. These programs are executed as user processes but have the privilege level of kernel mode. The benefit of the KML mechanism is a direct access to the kernel address space. Therefore, the overhead of system calls are eliminated while scheduling and paging remain as usual. There are two main problems that occur when applying the KML technology. The first problem is security, one needs to prevent illegal memory access and illegal code execution. KML uses an execution mechanism called TAL (Typed Assembly Language) to safely run in the kernel mode in an attempt to avoid the aforementioned problem [TMkml, TM02]. The second problem deals with the concept of taking user level applications and running them contrary to their natural environment. This can easily cause system failure.

A more native solution to minimize the applications responsiveness is to run kernel threads that are using fundamental normal processes facilities.

3.3 Restricted Memory Management in Kernel Environment

As widely explained in the memory management section, kernel space is more restrictive than in user space. For example, memory allocation in the kernel is satisfied immediately. While in user space, it is simply reserved in the linear address space by pointing a page table entry which references a read-only globally visible page filled with zeros. Writing that page triggers a page fault exception which results in allocating a new physical page mapped to the process page table [Gor04]. The restricted memory management in the kernel, forces system architectures and kernel programmers to take further precautions while designing and programming in the kernel environment.

3.3.1 The Kernel Mode Stack Problem

User space allocations are transparent with a large and dynamically growing stack. In the kernel environment the stack is small sized and fixed. It is possible to determine the stack size (as from version 2.6) at compile time choosing between 4 to 8KB. Each process that is executing in the kernel mode, receives its own kernel stack. The entire call chain of a process executing inside the kernel must be capable of fitting on the stack. In an 8KB stack size configuration, interrupt handlers use the stack of the process they interrupt. This means that the kernel stack size might need to be shared by a multiple function deep call chain and an interrupt handler. In a 4KB stack size configuration, interrupts have a separate stack making the exception mechanism more complicated and slower. Memory corruption caused by a stack overflow may occur and cause the system to be in an undefined state [RL03, BC03]. The kernel makes no effort to manage the stack. However it is possible to compile the kernel with special flags as the `CONFIG_DEBUG_STACKOVERFLOW` and the `CONFIG_DEBUG_STACK_USAGE`, to warn the user upon a stack overflow that may lead to a memory corruption.

An empirical study of operating system errors is found by automatic, static, compiler analysis applied to the Linux kernel. The study compares errors in different subsections of the kernel, discovers how bugs are distributed and generated, calculates how long bugs live in the kernel, clusters bugs according to errors types and compares the Linux kernel against the OpenBSD. The data used in the study comes from snapshots of the Linux kernel spanning over seven years (until 2.4.1 kernel series). After gathering the results, a total of 1025 bugs have been reported, 102 of them were because of large stack variables on the fixed-size kernel stack [YHE01].

3.3.2 Micro Kernels

In the late 1980s, a new concept of an operating system was introduced: the micro kernels. The idea of micro kernel is to minimize the kernel and to implement servers

outside the kernel. All servers run in user mode and have the protection of separated memory addresses to each application (as do normal processes in user mode). In contrast to the concept of the micro kernels, the traditional monolithic kernels pack all kernel utilities such as scheduling, networking, file systems, device drivers, memory management and more into a single kernel [JL96].

The micro kernels idea that became widely accepted by operating system designers has many obvious technology advantages such as: system flexibility and extensibility (can easily adopt to new hardware), server malfunction is as isolated as normal application, clean kernel interface, interdependences between the various parts of the system and more [JL96].

An interesting concept implemented in micro kernel is an external pager. The kernel continues to manage physical and virtual memory, as done in traditional monolithic kernels but forwards page faults to user level tasks [JL96].

Although the micro kernel has many theoretical advantages, it was a disappointment by means of performance, efficiency and flexibility. Today, operating system designers are moving toward a second generation of micro kernels trying to avoid old mistakes, while making micro kernels as the basis for all types of operating systems [JL96].

4 Body of Work

The main method while implementing the research is to support demand paging for the kernel mode stack. The solution that is proposed integrates the kernel source code and is a part of the kernel. The solution is configuration dependent and can be enabled or disabled using the kernel configuration tools. In the following sections the design implementation and testing utilities will be described.

4.1 Design

The main concept in implementing demand paging is to allocate a virtual address space with no corresponding physical pages. In kernel mode stack the `thread_info` and the start of the stack pointer are located in a predefined offset (that is `THREAD_SIZE` declared in `include/asm-i386/thread_info.h`). The main reason is to gain performance. The stack pointer address is used frequently and is stored in a special purpose register. In order to get a reference for the current process descriptor, one can ask for the stack pointer and quickly figure out the reference. This is done via the `current` macro that is implemented in the x86 architecture (3.1.3.1). In order to leave the current mechanism untouched, two alternatives are introduced to implement the solution. In the first option, two noncontiguous physical pages and a virtual address space that is aligned to `THREAD_SIZE` are allocated. The physical pages are mapped to the lowest virtual address and to the highest virtual address, in the kernel page table. The lowest virtual address will be used for the `thread_info` structure and the highest address will be used for the start of the stack pointer as can be seen in the following figure (figure 5). In the second alternative one physical page and a virtual address space that is aligned to `THREAD_SIZE` are allocated. The physical page is mapped to the highest virtual address. The `thread_info` is set to the top of the highest virtual address less than the size of the `thread_info` structure. The stack pointer starts from beneath the `thread_info` as shown in figure number 6. Although the `current` macro has to be updated with the new location of the `thread_info`, it is still based on the stack pointer register. In both options additional physical pages will be allocated when the CPU

triggers a page fault exception, by the exception handler. The size of the virtual area allocated may be of any size. In our implementation we used a virtual area of 16KB as can be seen in the following figures (figures 5,6).

The main problem of implementing demand paging in the kernel mode stack is stack starvation. When a program is executed in kernel control path and causes a page fault exception on the stack, the IA-32 CPU tries to push several registers (EIP, CS, and more) to the same stack. The CPU doesn't switch stack to a lower privilege level stack because the kernel runs at the lowest privileged level. Therefore, the IA-32 CPU cannot push the registers and generate a double fault exception which is an unrecoverable exception in the kernel. To solve the stack starvation problem, the IA-32 hardware task management mechanism is used. The IA-32 task management facility is provided to support process management for kernels and can be used to handle interrupts and exceptions via the IDT table. If an interrupt occurs and a task gate is assigned in the IDT, the CPU first saves the execution context of the interrupted program to a TSS data structure of the program instead of to the memory stacks. Then, the CPU restores the context from the TSS data structure specified in the IDT. In the implementation, only page faults exception that occur in the kernel mode stack are handled by the task management facility. The IDT page fault entry is set with a task gate entry as long as running in kernel control path. While the original interrupt gate entry is resumed for user mode applications. The task gate entry references a new TSS to provide a new execution context, and therefore a valid stack to save the CPU registers.

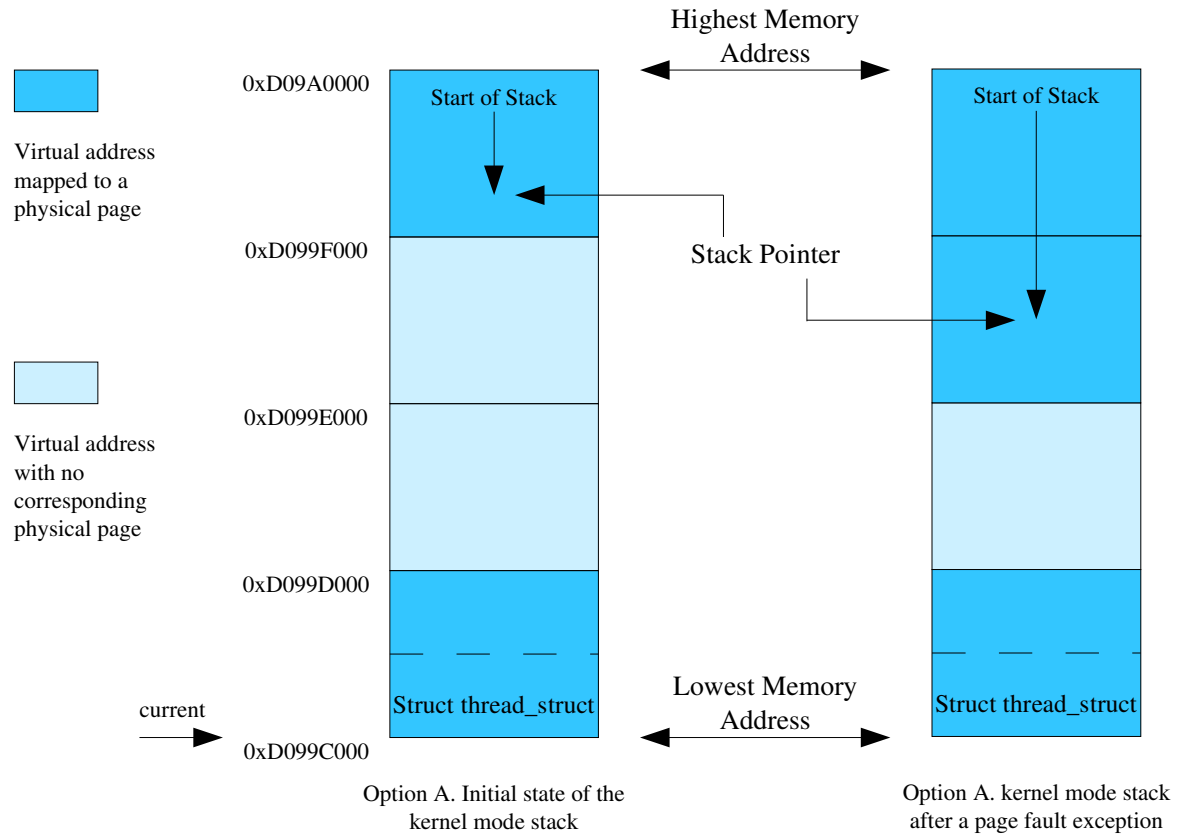


Figure 5: Option A.

New Kernel Mode Stack (size 16KB, with two physical pages)

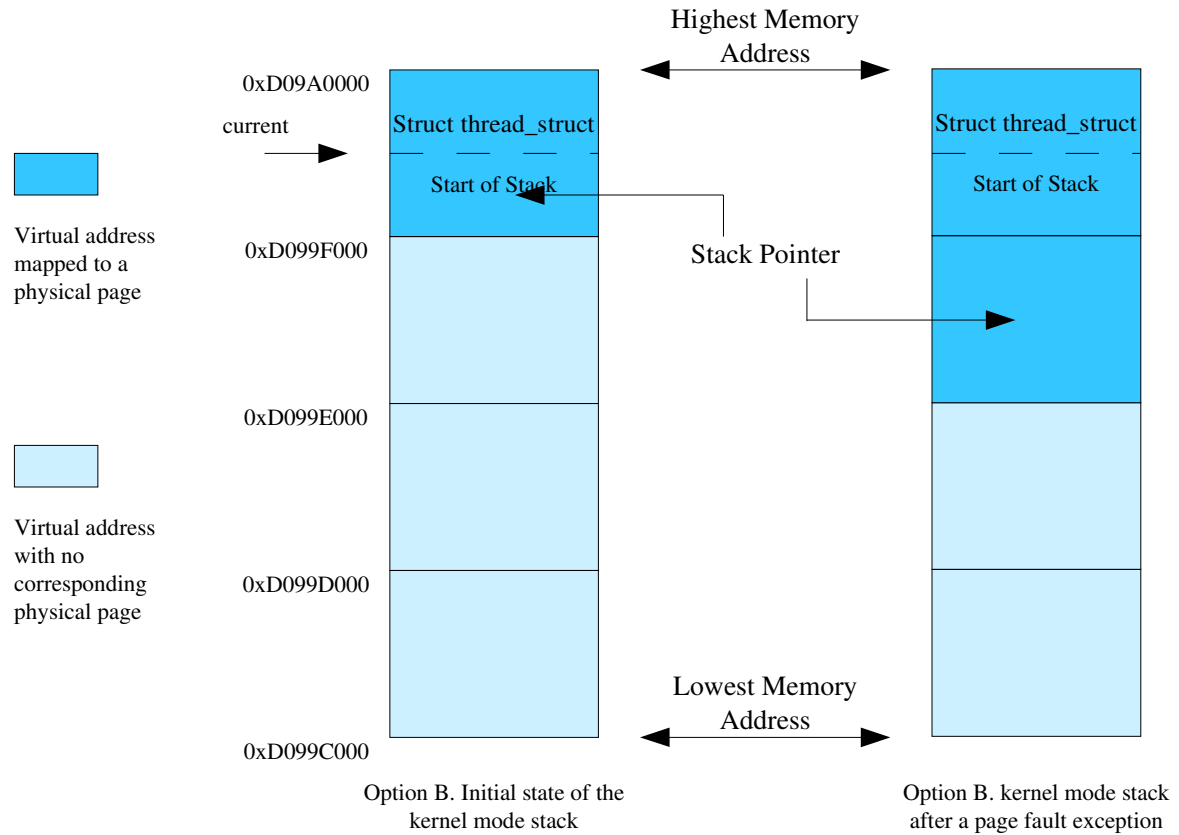


Figure 6: Option B.

New Kernel Mode Stack (size 16KB, with one physical page)

4.2 Resources

The resources necessary for this proposal are:

Implementing the relevant changes in the Linux kernel source code using the Linux kernel version 2.6.9. Red Hat Fedora core2 will be used as the operating system, running on Intel Pentium4 Celeron 2.4MHz processor and on IBM Thinkpad laptop with 1.7MHz Centrino processor. During implementation of the work proposed here, we will measure the performance of the new mechanism, so as not to introduce unnecessary performance degradations. To measure performance, we will use the BYTE UNIX Benchmarks (Version 3.11) tools suite, in order to test the kernel patch on both computers.

4.3 Implementation

Handling page faults exception for the kernel mode stack is implemented in the following stages 1-4 for option A, and 5 for option B:

Stage 1: Physical and virtual address allocations for Kernel Mode Stack.

All processes in Linux are created using the `do_fork()` system call (3.1.3.1). `do_fork()` is responsible for allocating a place for the `thread_info` and the kernel mode stack. The allocation is done via the `kmalloc()` interface which allocates 4 or 8KB of contiguous physical pages (3.1.2.5). In order to allocate noncontiguous physical pages a new mechanism is implemented. The new mechanism uses a new function called `vmalloc_thread_info()` and changes the `do_fork()` system call to work with the new function. `do_fork()` calls to `copy_process()` which then calls to `dup_task_struct()`. `dup_task_struct()` actually allocates the new area using the `alloc_thread_info()` macro, which was replaced with a call to the new function. Freeing the allocated area also has a new `vfree()` implementation. Changes to `fork.c` can be seen in the following `diff` code list (code list 1):

Body of Work

```
--- kernel/fork.c.orig 2005-07-17 14:53:49.963041384 +0300
+++ kernel/fork.c      2005-07-17 14:54:59.908408072 +0300
@@ -79,7 +79,7 @@ static kmem_cache_t *task_struct_cache;

void free_task(struct task_struct *tsk)
{
-   free_thread_info(tsk->thread_info);
+   vfree_thread_info(tsk->thread_info);
   free_task_struct(tsk);
}
EXPORT_SYMBOL(free_task);
@@ -264,7 +264,8 @@ static struct task_struct *dup_task_stru
    if (!tsk)
        return NULL;

-   ti = alloc_thread_info(tsk);
+   ti = vmalloc_thread_info();
    if (!ti) {
        free_task_struct(tsk);
        return NULL;
    }
}
```

Code 1: Allocate and Free The `thread_info` and The Kernel Mode Stack Area
(kernel/fork.c)

The new function accepts no parameters. The size it needs to allocate is fixed and equal to the `THREAD_SIZE` defined in `include/asm-i386/thread_info.h`. `vmalloc_thread_info()` calls to `__vmalloc_thread_info()` that accepts three parameters describing the allocation size and the behavior of the allocator. The `__vmalloc_thread_info()` is responsible for allocating a contiguous virtual address space (3.1.2.6), two noncontiguous physical pages and to map the new allocated area to the global kernel page table. The two physical pages are mapped to the base and to the highest virtual address providing physical area for the `thread_info` and the stack respectively, as can be seen in the following implementation (code list 2):

Body of Work

```
void *__vmalloc_thread_info(unsigned long size, int gfp_mask,
pgprot_t prot)
{
    struct vm_struct *area;
    struct page **pages;
    unsigned int nr_pages, array_size;
    size = PAGE_ALIGN(size);
    if (!size || (size >> PAGE_SHIFT) > num_physpages)
        return NULL;
    area = get_thread_info_vm_area(size, VM_ALLOC);
    if (!area)
        return NULL;
    nr_pages = size >> PAGE_SHIFT;
    array_size = (nr_pages * sizeof(struct page *));
    area->nr_pages = nr_pages;
    area->pages = pages = kmalloc(array_size, (gfp_mask &
~__GFP_HIGHMEM));
    if (!area->pages) {
        remove_vm_area(area->addr);
        kfree(area);
        return NULL;
    }
    memset(area->pages, 0, array_size);
    /* allocate only two pages, at the bottom and at the top of
the vm area, the pages will be used for the thread info and SP
respectively. */
    area->pages[0] = alloc_page(gfp_mask);
    if (unlikely(!area->pages[0])) {
        area->nr_pages = THREAD_SIZE/PAGE_SIZE;
        goto fail;
    }
    area->pages[THREAD_SIZE/PAGE_SIZE-1] = alloc_page
(gfp_mask);
    if (unlikely(!area->pages[THREAD_SIZE/PAGE_SIZE-1])) {
        area->nr_pages = THREAD_SIZE/PAGE_SIZE;
        goto fail;
    }
    if (map_thread_info_vm_area(area, prot, &pages))
        goto fail;
    area->nr_pages = THREAD_SIZE/PAGE_SIZE;
    return area->addr;
fail:
    area->nr_pages = THREAD_SIZE/PAGE_SIZE;
    vfree_thread_info(area->addr);
    return NULL;
}
```

Code 2: vmalloc_thread_info Implementation

The virtual area used for the `thread_info` and the stack is allocated with a new `get_thread_info_vm_area()` function. The original kernel function for virtual area allocation uses the alignment of one which means that the allocation size is rounded up to the next page alignment (3.1.2.6.2). In our implementation the alignment of `THREAD_SIZE` is used so the current mechanism of extracting the `thread_info` from the stack pointer continues to work (3.1.3.1).

The physical area is allocated with a new mechanism that works just the same as the normal physical page allocator (3.1.2.5.2), except that references to the current process are eliminated. This is because the kernel page fault handler works in a special context that doesn't have a normal process to reference.

Mapping the virtual and physical addresses to the kernel page table is done similarly to the current mapping implementation. In the page table entries the physical pages are mapped to the lowest and the highest virtual addresses as shown in the following implementation (code list 3):

```
static int map_thread_info_area_pte(pte_t *pte, unsigned long
address,
                                unsigned long size, pgprot_t prot,
                                struct page ***pages)
{
    unsigned long end;
    struct page *page = **pages;

    address &= ~PMD_MASK;
    end = address + size;
    if (end > PMD_SIZE)
        end = PMD_SIZE;

    /* map the first page */
    WARN_ON(!pte_none(*pte));
    if (!page)
        return -ENOMEM;
    set_pte(pte, mk_pte(page, prot));

    /* map the second (top) page */
    pte += (THREAD_SIZE/PAGE_SIZE-1);
```

```

    (*pages) += (THREAD_SIZE/PAGE_SIZE-1);
    page = **pages;
    WARN_ON(!pte_none(*pte));
    if (!page)
        return -ENOMEM;
    set_pte(pte, mk_pte(page, prot));

    return 0;
}

```

Code 3: Map The Thread Info Area To The Page Table Entries

Stage 2: Creating a Task State Segment

As described in section 3.1.3.5 the system architecture provides a hardware data structure called the task state segment (TSS). TSS keeps track of segment selectors, general purpose registers, 4 levels of stack pointers, the instruction pointer, and various permissions and flags. The fields of a TSS are divided into two main categories: dynamic fields and static fields. The CPU updates the dynamic fields upon context switch while the static fields are initialized once when a task is created.

The TSS is declared in `include/asm-i386/processor.h`. The relevant fields will be described in detail:

```

struct tss_struct {
    unsigned short  back_link, __blh;    Contains the segment selector for the
                                         TSS of the previous task. The
                                         processor copies the segment selector for the
                                         current TSS to the previous task link field of
                                         the TSS for the new task. This field permits a
                                         task switch back to the previous task by
                                         using the IRET instruction.

    unsigned long   esp0;                stack pointer and segment selector for
    unsigned short  ss0, __ss0h;        protection ring number 0.

    unsigned long   esp1;                stack pointer and segment selector for
    unsigned short  ss1, __ss1h;        protection ring number 1.
}

```

Body of Work

```
unsigned long    esp2;          stack pointer and segment selector for
unsigned short   ss2, __ss2h;  protection ring number 2.

unsigned long    __cr3;        Contains the base physical address of the
                                page directory. The __cr3 register is not
                                saved upon context switch and needs to be
                                updated.

unsigned long    eip;          Instruction pointer

unsigned long    eflags;       The EFLAGS register contains a group of
                                status flags, a control flag, and a group of
                                system flags. When an interrupt or exception
                                is handled with a task switch, the state of the
                                EFLAGS register is saved in the TSS for the
                                task being suspended.

unsigned long    eax, ecx, edx, ebx;  General purpose registers.

unsigned long    esp;          Stack pointer.

unsigned long    ebp;          General purpose registers.
unsigned long    esi;
unsigned long    edi;

unsigned short   es, __esh;    Segment selectors.
unsigned short   cs, __csh;
unsigned short   ss, __ssh;
unsigned short   ds, __dsh;
unsigned short   fs, __fsh;
unsigned short   gs, __gsh;

unsigned short   ldt, __ldth;  Contains the segment selector for the task's
                                LDT.

unsigned short   trace, io_bitmap_base;
unsigned long    io_bitmap[IO_BITMAP_LONGS + 1];
unsigned long    io_bitmap_max;
struct thread_struct *io_bitmap_owner;
unsigned long    __cacheline_filler[35];  pads the TSS to be cacheline-
                                            aligned (size is 0x100)
```

Body of Work

```
    unsigned long stack[64];          another 0x100 bytes for emergency
                                      kernel stack
} __attribute__((packed));
```

In the implementation, a static TSS for the kernel page fault exceptions is declared and is called the `pagefault_tss`. It is initialized with the relevant fields necessary to perform a hardware task switch as can be seen in the following implementation (code list 4):

```
struct tss_struct kernel_pagefault_tss __cacheline_aligned = {
    .esp0 = STACK_START,              Pointer to a predefined static array of 32bit
                                      unsigned integer for the local variables stack.
    .ss0  = __KERNEL_DS,              The stack segment attributes are as the kernel
                                      data segment.
    .ldt  = 0,
    .io_bitmap_base = INVALID_IO_BITMAP_OFFSET,
    .eip  = (unsigned long) kernel_page_fault, Points to the new
                                      kernel page fault
                                      handler
    .eflags = X86_EFLAGS_SF | 0x2,    0x2 bit is always set
    .esp   = STACK_START,              Initialize the current SP to a
                                      static predefined stack.
    .es    = __USER_DS,                Initialize segment registers.
    .cs    = __KERNEL_CS,
    .ss    = __KERNEL_DS,
    .ds    = __USER_DS,
    .__cr3      = __pa(swapper_pg_dir) Contains the base physical
                                      address of the global kernel
                                      page directory.
};
```

Code 4: Kernel Page Fault TSS

The new TSS resides in a new file located at: `arch/i386/kernel/pagefault.c`

The fields of the TSS should reflect the initial state of the registers when the context switch is started. Not all of the registers need to be filled and can be left to the application

use, however, the following fields must be filled in before starting the task switch (as implemented in code list 4):

- The Stack Pointer (ESP) is set to the top of the stack.
- The Instruction Pointer (EIP) is set to `kernel_pagefault` which is the task handler.
- eFlags should, at the very least, be set to `0x0002`.
- The Page Directory Base Pointer points to the global page table.
- The Segment Registers are set to appropriate values.

Another update is made to the initial macro of the TSS segment which is declared in `/include/asm-i386/processor.h` and can be seen from the following `diff` code (code list 5):

```

--- include/asm-i386/processor.h.orig      2005-07-17
15:59:49.133156040 +0300
+++ include/asm-i386/processor.h          2005-07-17
16:01:53.139304240 +0300
@@ -453,9 +454,10 @@ struct thread_struct {
     .esp0          = sizeof(init_stack) + (long)&init_stack,
 \
     .ss0          = __KERNEL_DS,          \
     .ss1          = __KERNEL_CS,          \
-    .ldt          = GDT_ENTRY_LDT,        \
+    .ldt          = GDT_ENTRY_LDT*8,      \
     .io_bitmap_base = INVALID_IO_BITMAP_OFFSET, \
     .io_bitmap     = { [ 0 ... IO_BITMAP_LONGS] = ~0 }, \
+    .__cr3        = __pa(swapper_pg_dir)  \
 }

```

Code 5: Initial TSS update

Stage 3: Updating the IDT and GDT tables

In order that a page fault exception in the kernel environment will trigger a hardware context switch, the IDT (3.1.4.4) and the GDT tables are updated (3.1.2.2). The current interrupt vector of the page fault exception points to an interrupt gate and is handled by a regular handler (3.1.4.4, 3.1.4.5.1). Changing the IDT entry to a task gate that points to a TSS descriptor in the GDT causes the hardware to context switch (3.1.3.5.1). In order to

use the new entry just in kernel mode, the entry needs to be updated only while in kernel control path. When resuming execution to user space the IDT entry needs to be set back to the original state. Setting the entry is done by the following function:

```
set_task_gate(14, GDT_ENTRY_PAGE_FAULT_TSS);
```

The first parameter to the function is the exception vector (3.1.4.2), the second parameter (`GDT_ENTRY_PAGE_FAULT_TSS`) is a new define of a free entry offset in the GDT table defined in `./include/asm-i386/segment.h`.

Resuming the IDT entry for user space is done using the function:

```
set_intr_gate(14, &page_fault);
```

The second parameter is the page fault handler of the normal page fault mechanism.

When the system starts the GDT entry is set once. A new TSS descriptor that points to a new `do_kernel_pagefault()` handler is added. The function to set the descriptor in the GDT is (the GDT table called `cpu_gdt_table` and is declared in `./arch/i386/kernel/head.S`):

```
__set_tss_desc(cpu, GDT_ENTRY_PAGE_FAULT_TSS, &do_kernel_pagefault);
```

Stage 4: Implementing the kernel page fault handler

The kernel page fault handler is responsible for allocating a new physical page and for mapping it to the kernel page table. The first stage is determining the address that caused the page fault. The address is saved by the hardware exception mechanism to the `cr2` control register and can be read using an assembly instruction. After extracting the faulting address the address is searched in order to find the `vm_area` that includes the address. Finding the virtual area is done by simply searching the global `vmlist` (3.1.2.6.1) using a new function called `find_thread_info_vm_area()` as can be seen in the following implementation (code list 6):

```

struct vm_struct *find_thread_info_vm_area(void *addr)
{
    struct vm_struct **p, *tmp;

    write_lock(&vmlist_lock);
    for (p = &vmlist ; (tmp = *p) != NULL ; p = &tmp->next) {
        if ((unsigned)(tmp->addr) == (unsigned)addr)
            goto found;
    }
    write_unlock(&vmlist_lock);
    return NULL;

found:
    printk("vm_area found address 0x%x\n", (unsigned int)
(tmp->addr));
    write_unlock(&vmlist_lock);
    return tmp;
}

```

Code 6: Find The Thread Info Virtual Area

If the `vm_struct` is found, a new physical page is allocated and mapped in the kernel page table to the faulting virtual address. It is done by the `expand_stack_size()` function as can be seen in the following implementation (code list 7):

```

void expand_stack_size(struct vm_struct *area)
{
    struct page **pages;
    unsigned int expand_address;
    unsigned gfp_mask = GFP_KERNEL | __GFP_HIGHMEM;
    pgprot_t prot = PAGE_KERNEL;
    unsigned int page_idx;

    for (page_idx=THREAD_SIZE/PAGE_SIZE-1; page_idx>=0;
page_idx--) {
        if (area->pages[page_idx])
            continue;
        else
            break;
    }
    expand_address = (unsigned int)((area->addr)
+PAGE_SIZE*page_idx);

    area->pages[page_idx] = alloc_thread_info_page(gfp_mask);
    if (unlikely(!area->pages[page_idx])) {

```

```

/* free all available pages in
vfree_thread_info() */
    area->nr_pages = THREAD_SIZE/PAGE_SIZE;
    printk("Alloc page failed\n");
    goto fail;
}
/* set to max number of pages */
area->nr_pages = THREAD_SIZE/PAGE_SIZE;
pages = &(area->pages [page_idx]);

if (map_expand_stack_vm_area(expand_address, prot,
&pages))
    goto fail;

page_table_trace((unsigned int) (area->addr));
return;
fail:
printk("Failed to expand_stack_size\n");
vfree_thread_info(area->addr);
}

```

Code 7: Expanding The Kernel Mode Stack

Returning from the exception handler and switching back to the previous task is done by executing the CPU IRET instruction. After IRET, the execution context of the page fault handler will be saved to the 'kernel_pagefault_tss'. This means that when the next page fault occurs, the page fault task will start from just after IRET with the stack pointer pointing to some memory location that may differ from 'STACK_START'. In the implementation the last instruction is above the main handler code (implemented with goto) so in the next exception the handler will run as expected (code list 8).

Before executing the return code, the busy bit in the TSS descriptor needs to be cleared. If the task switch is initiated with an exception, the processor sets the busy (B) flag in the new task's TSS descriptor; if initiated with an IRET instruction, the busy (B) flag is left set so it needs to be cleared on return. Although tasks are not reentrant, setting the busy flag in the handler works fine because recursive page faults should never occur. The actual page fault handler can be seen in the following implementation (code list 8):

```

static void do_kernel_pagefault(void)
{
    unsigned int address, aligned_addr;
    unsigned int i=0;
    struct vm_struct *area;

    goto handle_fault;

return_from_fault:
    __asm__("iret");

handle_fault:
    cpu_gdt_table[GDT_ENTRY_PAGE_FAULT_TSS].b &= 0xffffdff;
    __asm__("movl %%cr2,%0":"=r" (address));
    aligned_addr = ((address+PAGE_SIZE)&~(PAGE_SIZE-1));
    /* search for the vm area */
    for(i=0; i<THREAD_SIZE/PAGE_SIZE; i++) {
        area = find_thread_info_vm_area((void*)
(aligned_addr-(i*PAGE_SIZE)));
        if(area) {
            printk("vm area 0x%x, addr 0x%x, address
0x%x\n", (unsigned int)area, (unsigned int)(area->addr),
aligned_addr-(i*4096));
            break;
        }
    }
    /* allocate a new physical page, expand the stack size */
    expand_stack_size(area);
    goto return_from_fault;
}

```

Code 8: The Kernel Page Fault Handler

Stage 5: Relevant only to support option B. and introduced in more general details:

After allocating the virtual area for the `thread_info` and the kernel mode stack, the `thread_info` is set to the top of the highest virtual address less than the size of the `thread_info` structure. The stack pointer starts from beneath the `thread_info` (figure 5). The changes include updating the initial system thread data structures (the first thread that runs in the system has an initial global structures), updates to the `do_fork()` mechanism that creates a new process (setting the correct `thread_info`

and stack pointer offsets) and updates to the `current` macro offsets.

4.4 Testing Methods

Final testings are made to make sure that all the mechanisms implemented in the research work as expected. In order to do so, the new system is tested for its new functionality and then a benchmark is run to make sure that no degradation is found in system performance.

4.4.1 Functionality test

The new mechanism is tested by writing two test utilities. The first is a kernel module called `mm_test` that implements three basic system calls: `open`, `ioctl` and `close`. The module registers itself as a character device and is loaded into the kernel. The `ioctl` system call is used to overload the module stack and to trigger a page fault exception. The second utility written is a user space application called `mm_test_application`. The `mm_test_application` opens a handle to the `mm_test` device and calls to the `ioctl` file operation. Running the application and calling to the `ioctl` system call causes the module to trigger a page fault exception by overloading the kernel mode stack. These utilities were executed on both of the original kernel and the new updated kernel. In the original kernel the CPU triggered a page fault exception and then a double fault exception. The double fault exception is an unrecoverable exception in the kernel. In the updated kernel the CPU triggered a page fault exception, which was handled by the new exception handler resulting in allocating a new physical page to the faulting address. The result can be seen in the following printout (figure 7):

```
Page fault exception address 0xe01c6e88
virtual memory area found at 0xe01c4000
Expand stack:
Allocate physical page
map pmd: start address 0xe01c6000 end 0xe01c7000
map pte: start address 0xe01c6000, size 4096
set page table entry: address 0xe01c6000 pte 0x1e98f163
Page table trace start from virtual address 0xe01c4000
=====
virtual address 0xe01c4000 page table entry 0x18dfa163
-----
virtual address 0xe01c5000 page table entry 0x0
-----
virtual address 0xe01c6000 page table entry 0x1e98f163
-----
virtual address 0xe01c7000 page table entry 0x18d7f163
-----
```

Figure 7: A Printout of The Faulting Virtual Address and a Page Table Trace of The `vm_area` Space That Belongs to The Stack (in a 16KB stack).

4.4.2 Benchmarks

The BYTE Unix benchmark is used to measure system performance for the new mechanism integrated into the Linux kernel. The BYTE Unix benchmark is an open source tools suite. The benchmark consists of numerous groups of programs including: arithmetic, system calls, memory operations, disk operations, dhrystone, database operations, system loading, and miscellaneous. The system test programs that were used check: system calls overhead, pipe throughput, pipe context switching, process creation (spawning of child processes) and `execl` (replacement of the current process by a new process). The benchmark has been tested against the original and the updated kernel. The tests were repeated after rebooting the system. Examination results found no performance degradations in the new mechanism integrated into the Linux kernel. See appendix A for further details.

5 Discussion and Conclusions

This paper suggests a new approach to handle memory management in Linux kernel, using demand paging for kernel mode stack. Linux kernel is developed as an unstructured monolithic kernel with a strong practical emphasis, necessitating us to search for a solution that will not change the characteristics of the kernel and will not cause degradation in kernel performance. The solution found uses the IA-32 hardware task management facility that is provided to support process management for kernels. The solution is implemented and integrated into the kernel source code, and it can be enabled or disabled easily using the traditional kernel configuration tools.

5.1 The Importance of The Research

5.1.1 System Stability

Using demand paging in kernel environment for kernel mode stack reduce unpredicted memory corruption and improves the kernel stability. The Computer Science Laboratory at Stanford University analyzed the Linux kernel source code. The research project known as, Meta compiler Project, lasted for a few years. The analysis was based on a Coverity analysis system developed at the University. In 2001, the research community presented a study of defect trends in the Linux operating system. They found over 1000 defects in the Linux source code version 2.4.1. about 10 percent were because of large stack variables on the fixed-size kernel stack. In 2004, repeated analysis of the Linux kernel was performed. This analysis punctuated by releases of security holes in the 2.6.4 kernel source code [YHE01, CIse, CI04]. The findings can be seen from the following quotation[CI04]:

“In particular, Coverity Prevent™ currently detects several types of buffer overruns in the heap and the stack. Those results marked as "Negative Returns" and "Reverse Negative" may have the most devastating security implications because they are often triggered by rare failure cases that almost never happen in normal system operation. The goal of a malicious attacker, though, is to drive the system into an unexpected state, which can, in some cases, trigger these disastrous buffer overruns.”

5.1.2 Powerful Kernel Entities

Using demand paging in the kernel environment for kernel mode stack, makes kernel entities as kernel threads much more powerful and significant in the system. Kernel threads do not have a memory descriptor of their own, and therefore, cannot use normal processes memory areas. In addition, the stack size is fixed adding another restriction to the aforementioned problem. One of the solutions for the fixed stack size includes the implementation of a private stack. It is used in applications that need a large stack size and are implemented for a specific task and not as a general solution. The private stack mechanism is difficult to implement in Linux because of the stack and the process descriptor architecture (as introduced earlier in the research). Using large and dynamic stack for kernel threads reduces unpredictable memory corruption, and enables programmers to use kernel threads for wider usage of applications and with fewer precautions.

5.1.3 Efficient Physical Memory Handling

The kernel mode stack size can be either 4 or 8KB according to kernel configurations. Although, there are some driver applications (mainly networking drivers) that require 8KB stack for installation and use. For example, the NdisWrapper which is an open source wrapper tool driver (the NdisWrapper program enables the Linux kernel to support wireless cards). That means, that distributions that have the 4KB stack option hardwired, needs to be compiled again with the 8KB stack size option enabled (some drivers used in the kernel are patched for the 8KB stack option along with the patched source code). Using the new kernel page faults mechanism, enables to use a kernel that is configured to 4KB stack size, and yet, to sustain demand drivers.

5.2 Comparison To Previous Research and Tools

5.2.1 The Kernel Mode Linux Project

As mentioned in the introduction, the KML technology enables us to execute user

programs in kernel mode. Applications running in kernel mode can access the kernel address space directly eliminating the overhead of system calls. The kernel mode user processes are ordinary user processes except for their privilege level. Kernel-mode user processes have their own address space. Their memory management works the same as in user space including the paging mechanism [MY03]. Exception as page faults are handled the same way as for ordinary user processes, using the concept of hardware context switch as done in this research. The main problem using KML technology is that from the beginning, user space applications are not written with the same precaution and awareness as in kernel mode. Although security actions are taken, they are a small portion of the entire solution.

5.2.2 Micro Kernels

The main concept of micro kernels is to minimize the kernel and to implement servers outside the kernel that will handle kernel utilities such as device drivers, networking, file system, memory management and more. The Adeos (Adaptive Domain Environment for Operating Systems) project as introduced in the introduction finds the solution for minimizing the kernel by pushing Linux out of ring-zero and into ring-one. Adeos itself runs at ring-zero and handles specific hardware dependent, while Linux or other OS domains are handling all other OS issues including page management. Implementing and handling page faults exception are straightforward in micro kernels. CPU exceptions transfer execution to lower rings having a place to store the current registers state. After returning from the handler the registers are resumed. Transfer of execution to an exception- or interrupt-handler procedure in a less privileged is not permitted by processor results in a general-protection exception (#GP) [IAP99].

5.3 Future Directions

5.3.1 Dynamic Kernel Mode Stack Size

The user process dynamically growing stack differs from the new kernel mode stack by a few main points: the user stack is obtained from the process virtual area space which is larger and not globally visible. In the kernel mode stack the virtual area is part of the global virtual memory that belongs to the kernel. It is limited with size and must be protected.

It is possible to change the current interface to enable users to choose their virtual stack size in the kernel mode stack. A process or a kernel thread that needs a lot of memory space can request the size it needs from the system, while other utilities that don't need a lot of virtual space will not 'waste' unnecessary virtual addresses.

5.3.2 SMP Support

Tasks are not reentrant. A second instance of a page fault handler may overwrite the stack of the first one. Since recursive page faults should never occur on UP machine, there is no need to protect the local variables of the current stack. Although page fault exceptions may work on UP machines, on SMP machines each CPU needs to have its own TSS to handle concurrent faults.

5.3.3 Fixed virtual address interval

In contrary to user space where each process has its own virtual address space, virtual addresses in the kernel are global and shared by all of the kernel control paths. Invoking thousands of processes with a large virtual stack may exhaust the virtual memory space in the kernel. The solution is to allocate one interval of fixed size virtual addresses for all the tasks in the system, and every context switch to update the addresses page table entries. It requires flushing the TLB but only for the specific address interval. The benefits are using huge kernel stack size without exhausting the kernel virtual memory.

5.3.4 Comprehensive Page Fault Handling in The Kernel

Handling page fault exceptions in the kernel mode stack is the first step for handling faults in other parts of the kernel control path. Many kernel utilities that allocate memory in the kernel address space don't necessarily require immediate physical allocation. Moving toward 64 bit system architecture, the concept of virtual memory and demand paging is more significant. In those systems the virtual address space becomes much larger while the physical address space remains almost the same.

6 Appendix

6.1 Appendix A

The BYTE UNIX benchmark results:

Test 1 environment :

=====

Distribution: Fedora core 2/ Linux

CPU/Speed: Celeron P4 2.4GHz

Ram: 256MB

Motherboard: Intel 845

Bus: PCI

Cache: 512k (P4)

Controller: Intel

Disk: Maxtor 40GB

Load: Multi-user, run level 3

Kernel: Linux 2.6.9

Kernel ELF?:

pgms: gcc (GCC) 3.3.3 20040412 (Red Hat Linux 3.3.3-7)

Appendix A

Original kernel benchmark results:

BYTE UNIX Benchmarks (Version 3.11)

System -- Linux localhost.localdomain 2.6.9 #1 Tue Jul 19 11:35:27 IDT

2005 i686 i686 i386 GNU/Linux

Start Benchmark Run: Thu Jul 21 15:03:12 IDT 2005

1 interactive users.

| | | |
|-----------------------------------|---------------|----------------------|
| System Call Overhead Test | 1104116.5 lps | (10 secs, 6 samples) |
| Pipe Throughput Test | 927316.5 lps | (10 secs, 6 samples) |
| Pipe-based Context Switching Test | 164489.2 lps | (10 secs, 6 samples) |
| Process Creation Test | 5301.1 lps | (10 secs, 6 samples) |
| Execl Throughput Test | 1573.9 lps | (9 secs, 6 samples) |

INDEX VALUES

| TEST | BASELINE | RESULT | INDEX |
|-----------------------------------|----------|----------|-------|
| Execl Throughput Test | 16.5 | 1573.9 | 95.4 |
| Pipe-based Context Switching Test | 1318.5 | 164489.2 | 124.8 |
| | | | ===== |
| SUM of 2 items | | | 220.1 |
| AVERAGE | | | 110.1 |

Appendix A

Patched kernel benchmark results:

BYTE UNIX Benchmarks (Version 3.11)

System -- Linux localhost.localdomain 2.6.9pagefault #3 Tue Jul 19 14:13:51 IDT

2005 i686 i686 i386 GNU/Linux

Start Benchmark Run: Thu Jul 21 14:44:47 IDT 2005

1 interactive users.

| | | |
|-----------------------------------|---------------|----------------------|
| System Call Overhead Test | 1102719.3 lps | (10 secs, 6 samples) |
| Pipe Throughput Test | 950919.8 lps | (10 secs, 6 samples) |
| Pipe-based Context Switching Test | 160663.2 lps | (10 secs, 6 samples) |
| Process Creation Test | 4712.9 lps | (10 secs, 6 samples) |
| Execl Throughput Test | 1571.7 lps | (9 secs, 6 samples) |

INDEX VALUES

| TEST | BASELINE | RESULT | INDEX |
|-----------------------------------|----------|----------|-------|
| Execl Throughput Test | 16.5 | 1571.7 | 95.3 |
| Pipe-based Context Switching Test | 1318.5 | 160663.2 | 121.9 |
| | | | ===== |
| SUM of 2 items | | | 217.1 |
| AVERAGE | | | 108.6 |

Appendix A

Test 2 environment :

=====

Distribution: Fedora core 2/ Linux

CPU/Speed: P M 1.7GHz

Ram:512MB

Motherboard:

Bus: PCI

Cache: 2MB

Controller: Intel

Disk: Fujitsu 60GB

Load: Multi-user, run level 3

Kernel: Linux 2.6.9

Kernel ELF?:

pgms:gcc (GCC) 3.3.3 20040412 (Red Hat Linux 3.3.3-7)

Appendix A

Original kernel benchmark results:

BYTE UNIX Benchmarks (Version 3.11)

System -- Linux localhost.localdomain 2.6.9 #1 Thu Jul 21 14:48:22 IDT 2005 i686

i686 i386 GNU/Linux

Start Benchmark Run: Thu Jul 21 15:36:59 IDT 2005

1 interactive users.

| | |
|-----------------------------------|------------------------------------|
| System Call Overhead Test | 1733858.5 lps (10 secs, 6 samples) |
| Pipe Throughput Test | 975096.4 lps (10 secs, 6 samples) |
| Pipe-based Context Switching Test | 379686.3 lps (10 secs, 6 samples) |
| Process Creation Test | 21986.3 lps (10 secs, 6 samples) |
| Execl Throughput Test | 5368.3 lps (9 secs, 6 samples) |

INDEX VALUES

| TEST | BASELINE | RESULT | INDEX |
|-----------------------------------|----------|----------|-------|
| Execl Throughput Test | 16.5 | 5368.3 | 325.4 |
| Pipe-based Context Switching Test | 1318.5 | 379686.3 | 288.0 |
| | | | ===== |
| SUM of 2 items | | | 613.3 |
| AVERAGE | | | 306.7 |

Appendix A

Updated kernel benchmark results:

```
BYTE UNIX Benchmarks (Version 3.11)
System -- Linux localhost.localdomain 2.6.9pagefault #2 Thu Jul 21 15:04:47 IDT
      2005 i686 i686 i386 GNU/Linux
Start Benchmark Run: Thu Jul 21 15:12:07 IDT 2005
1 interactive users.
System Call Overhead Test          1733858.9 lps (10 secs, 6 samples)
Pipe Throughput Test               974959.9 lps (10 secs, 6 samples)
Pipe-based Context Switching Test  380850.5 lps (10 secs, 6 samples)
Process Creation Test              20698.9 lps (10 secs, 6 samples)
Execl Throughput Test               5324.3 lps (9 secs, 6 samples)
```

INDEX VALUES

| TEST | BASELINE | RESULT | INDEX |
|-----------------------------------|----------|----------|-------|
| Execl Throughput Test | 16.5 | 5324.3 | 322.7 |
| Pipe-based Context Switching Test | 1318.5 | 380850.5 | 288.9 |
| | | | ===== |
| SUM of 2 items | | | 611.5 |
| AVERAGE | | | 305.8 |

7 References

- [LKsc] The Linux Kernel Source Code Version 2.6.9
<ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.9.tar.gz>
- [GNU] The GNU Operating System project <http://www.gnu.org>
- [BHB99] Ivan T. Bowman, Richard C. Holt and Neil V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. ICSE'99, Los Angeles, May 16-22, 1999.
- [RL03] Robert Love. Linux Kernel Development, Sams; 1st edition, September 8, 2003
- [MKJ96] Michael K. Johnson, Linux Memory Management, 1996
<http://www.tldp.org/LDP/khg/HyperNews/get/memory/linuxmm.html>
- [BC03] Daniel Pierre. Bovet, Marco Cesati, Understanding the Linux Kernel, O'reilly; 2nd Edition, 2003
- [Gor04] Mel Gorman, Understanding The Linux Virtual Memory Manager, 15th February 2004.
- [IAB97] Intel Architecture Software Developer's Manual Volume 1:Basic Architecture
1997
- [IAI97] Intel Architecture Software Developer's Manual Volume 2:Instruction Set Reference 1997

References

- [IAP99] Intel Architecture Software Developer's Manual Volume 3: System Programming. 1999
- [TMkml] Toshiyuki Maeda. "Kernel Mode Linux: Execute user process in kernel mode" <http://www.yl.is.s.u-tokyo.ac.jp/~tosh/kml>
- [TM02] Toshiyuki Maeda. "Safe Execution of User programs in Kernel Mode Using Typed Assembly Language". Thesis, The University of Tokyo on February 5, 2002.
- [YHE01] A. Chou, J.F. Yang, B. Chelf, S. Hallem, and D. Engler, An Empirical Study of Operating Systems Errors. In Proceedings of the 18th ACM, Symposium on OS Principals (SOSP), pp. 73-88, Lake Louise, Alta. Canada, October 2001.
- [JL96] J. Liedtke, Toward Real Microkernels, Communications of the ACM, Vol. 39 (9), September 1996.
- [KYad] Karim Yaghmour, Adaptive Domain Environment for Operating System. Opersys inc.
- [VY97] Victor Yodaiken, An Introduction to RTLinux, New Mexico Institute of Technology, Oct. 7, 1997.
- [JO89] John Ousterhout, Why Aren't Operating Systems Getting Faster As Fast As Hardware? Western Research Laboratory, Palo Alto, California USA, October, 1989

References

- [MB99] Moshe Bar, “Tangled In The Threads” Byte.com Dec 28, 1999
http://wearcam.org/ece385/process_scheduling_in_linux.htm
- [MBDPH00] P.Mantegazz, E.Bianchi, L.Dozio, S. Papacharalambous, S. Hughes
“RTAI: Real-Time Application Interface” April 2000.
- [CW02] Clark Williams. “Linux Scheduler Latency”, Red Hat, Inc. March 2002
<http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf>
- [KD00] Kevin Dankwardt “Real Time and Linux, Part 3:Sub-Kernels and Benchmarks” May 20, 2001.
<http://www.linuxdevices.com/articles/AT6320079446.html>
- [AG00] George Anzinger and Niget Gamble, “Design of Fully Preemptable Linux Kernel”, MontaVista Software, September 2000.
- [BK04] Bernard Kuhn “The Linux real time interrupt patch”, jan 2004
<http://linuxdevices.com/articles/AT6105045931.html>
- [RL00] Rick Lehrbaum “Real-time Linux – what is it, why do you want it, how do you do it?”, Sept, 14, 2000.
<http://www.linuxdevices.com/articles/AT9837719278.html>
- [TM03] Toshiyuki Maeda. “Kernel Mode Linux”, Linux Journal, May,1, 2003
<http://www.linuxjournal.com/node/6516/print>

References

[CIse] Coverity, Inc. A Software engineering company developing source code analysis systems.

<http://www.coverity.com>

[CI04] Analysis of the Linux kernel, Coverity Corporation 2004.

http://www.coverity.com/datasheets/linux_report.pdf

[SLKR] <http://www.silkroad.com/> Unix/Linux benchmark forums.