Developing A Multi-threaded Kernel From Scratch

Thesis submitted in partial fulfilment

for the Award of

MSC COMPUTING POST GRADUATE DEGREE

in

COMPUTING

by

DANIEL MCCARTHY

DATED: SEPTEMBER 21, 2021

CARDIFF MET MANAGEMENT BUILDING

CERTIFICATE

It is certified that the work contained in the thesis titled **Developing A Multi-threaded Kernel From Scratch** by **Daniel McCarthy** has been carried out under my/our supervision and that this work has not been submitted elsewhere for a degree.

It is further certified that the student has fulfilled all the requirements of Comprehensive Examination, Candidacy and SOTA for the award of **Msc Computing Post Graduate Degree**.

Supervisor
Bakhsh, Sheikh Tahir
Cardiff Met Management Building

DECLARATION BY THE CANDIDATE

I, **Daniel McCarthy**, certify that the work embodied in this thesis is my own bona fide work and carried out by me under the supervision of **Bakhsh, Sheikh Tahir** from **May 2021** to **Sept 2021**, at the **Cardiff Met Management Building**, Cardiff Metropolitan University. The matter embodied in this thesis has not been submitted for the award of any other degree/diploma. I declare that I have faithfully acknowledged and given credits to the research workers wherever their works have been cited in my work in this thesis. I further declare that I have not willfully copied any other's work, paragraphs, text, data, results, *etc.*, reported in journals, books, magazines, reports dissertations, theses, *etc.*, or available at websites and have not included them in this thesis and have not cited as my own work.

Date: September 21, 2021
Place: Wales, Cardiff

Signature of the Student
Daniel McCarthy

ACKNOWLEDGEMENT

I would like to acknowledge myself for all the hard work I have done since 11 years old to be at the point I am at that I am able to develop kernels

Abstract

In this paper you will learn how kernels work and how filesystems work. Your also shown imporant protection mechnisms inside of a processor such as protection rings and paging and how they can be used with your kernel to protect user programs and the kernel its self. Developing a kernel can be difficult, this paper was created to represent a project that was created called COS32(Computer Operating System 32 bit) which is a 32 bit multi-threaded kernel. The full URL to the project source code can be found on Github here:

https://github.com/nibblebits/COS32

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Kernel development is a rare and required skill set. Without kernel developers you would not have Windows, Linux or Mac. This paper is based on a project that was created to help teach kernel development to students around the world, it was created as a basis for students to base their work off and was also created to answer some important questions in kernel development such as:

1. Why is kernel development important?
2. What is paging used for and how can it increase security in an operating system?
3. What are the components of a kernel?
4. How are user programs isolated from other user programs, what role does the kernel play in this?
5. How did kernels from the 1980s and early 1990s work and how they are they different from the kernels of today?
6. What are the restrictions of 16 bit kernels in Intel architectures.
7. How does a BIOS boot your computer?

A small custom protected mode multi-tasking kernel has been developed from scratch to help illustrate what has been obtained and learned from this study. The kernel is available

freely under the GPL (General Public License) so that all software engineers worldwide will be able to use and learn from this work without restrictions.

1.1 Why is kernel development important?

Kernel development is mandatory in modern computing and without it you would not have Windows, Linux, Mac, Android, Apple phones and much more. The kernel is responsible for ensuring user programs run correctly, have access to the resources that they need and to allow a running program to work independently by its self without knowledge of any other running program. The kernel must keep user programs memory separate from one another, the kernel does this by providing an illusion to a user program that it has access to memory that it does not and that it owns most of the address space. In reality the memory addresses that a user program accesses may not even be the physical memory addresses that the program has been loaded into. This concept is called paging and is heavily used in modern computing to give the illusion of more memory than a system has, and its used for security purposes to hide processes memory from each other. [6]

Chapter 2

Literature Review

In this literature review we will discuss different papers and compare the papers that have been found.

2.1 Kernel History

Before we had operating systems and kernels computers would execute single programs, they had no layer that would manage processes and tasks, therefore processes and tasks did not exist. Single programs could be run through the machine and executed. These programs had no lower layer to execute and would have had to have been completely self-efficient. Per Brinch Hanson describes this in detail in his paper "The Evolution Of Operating Systems". Per Brinch Hanson explains how in 1954 computers had no operating systems and computers such as the IBM 701 were operated completely manually by their users, he explains how this led to a lot of time being wasted and that the cost of wasted computer time was $146,000 per month [7]. Hansen further explains how the solution to this problem is batch processing. Batch processing is the idea that computers are able to schedule their own workload by the means of software and be able to run programs without user interaction. [7]. Batch processing back in the 1950s and 1960s was achieved by using punch cards and then submitting them into the computer to be executed. Hansen has shown the importance of operating systems and kernels as without them users would

be forced to execute one program at a time and wait for the result, while hoping that their program is error and bug free and that they have not wasted other people's time. Dale Fisk from Columbia university wrote about how he wrote programs on punch cards in 1973 and he did it by using a program somebody else wrote that was a compiler responsible for producing new punch cards from a source deck. The source deck would get processed and compiled and out would come the compiled program on a new punch card [5]. Dale Fisk's experience with writing and executing programs on punch cards in 1973 supports Hansen's claims that batch processing made use of punch cards that got executed on a computer.

Microsoft MS-DOS was first released on August 12, 1981. The MS-DOS kernel was responsible for loading COM and EXE executable files and providing an interface for them to talk with the system. In MS-DOS programs talk with the kernel by using CPU interrupts 0x20 to 0x3f. When invoking an interrupt, the processor would find the corresponding interrupt handler and it would be executed essentially running the interrupt handler responsible for the interrupt number inside the MS-DOS kernel. [3]. The services provided in the MS-DOS kernel are access to the filesystem, memory management facilities, a device and input control api, process control (ability to start and exit programs) and peripheral support [3]. When you first turn on your computer the bootloader is loaded by the BIOS which then loads MS-DOS's "MSDOS.sys" file and the "IO.sys" file into memory. These files are responsible for making up the MS-DOS kernel [3]. The boot sector is always at the beginning of a partition, for MS-DOS systems it should contain a BIOS parameter block which contains information about the device and the number of sectors that are in the disk along with a loader routine responsible for loading the kernel. The BPB makes up part of the FAT filesystem [3]. MS-DOS can recognize two types of devices, block devices such as floppy disks or fixed disk drives: and character devices such as keyboard, display, printer, and communication ports [3]. Robert W. Oliver also clarifies many of the points Chugh has made. He says that when he learned about the release of Microsoft's MS-DOS source code, he was very ecstatic. He studied the source code of MS-DOS and found that the MS-DOS kernel has a hardware abstraction input/output layer called IO.sys

that is pared with the MS-DOS kernel MSDOS.sys. Both of these two files make up the core of the system. [12]

2.2 Kernel Security And Vulnerabilities

Testing the Linux kernel is difficult and problematic, ensuring that a kernel has no vulnerabilities is proving to be challenging. Kernel developers attempt to find as many security issues as possible. Security issues can be reduced by finding bugs quickly to avoid too many people being exposed to the bug. [26] You must also ensure that the code base is kept clean and you must avoid allowing bad code into the code base as bad code can lead to further security problems [26] Security holes in a kernel can allow you to bypass kernel security mechanisms and gain root access. [26]

Kernel vulnerabilities may lead to information disclosure, privilege escalation, rootkit planting and more. Kernel vulnerability analysis is critical in system security so that you can identify software vulnerabilities before they are exploited by hackers in the wild. [10]

For example vulnerabilities that cause a kernel crash would kill all the processes in the operating system. Valuable unsaved work could be lost. [10]

It is also possible a hacker could access private data through by using pointers from kernel space to bypass security systems and execute malicious code. Privilege escalation could allow an attacker to get root privilege's. Compared to user-space software kernels are most vulnerable. [10]

Both Zaidenberg's [26] paper "Detecting Kernel Vulnerabilities during the Development Phase" and Lu's [10] paper "Kernel vulnerability analysis: A survey" share similar points of view around privilege escalation and gaining root privileges. The dangers of kernel development if done improperly are clearly shown a malicious user-space program may gain unrestricted access if it successfully exploits the kernel.

A paper titled "Kernel Data Attack Is a Realistic Security Threat" was written by Jidong Xiao, Hai Haung, and Haining Wang. They are from College of William and Mary, IBM

T.J. Watson Research Center, and university of Delaware. In their paper they describe kernel data attacks, a kernel data attack is a pacific attack towards a kernel where kernel structures can be modified in memory causing the kernel to act inappropriately. This can provide a way for a malicious individual or organization to access the system in ways that should not be possible [24]. The authors of the paper explained how they disabled some security features in the Linux kernel to examine the impact they could have with these security features disabled. The authors stated that there is around 380 thousand global function pointers and global variables in the Linux kernel and much of this memory is subject to change at runtime. The authors explain how by altering this pointer and variable memory. Kernel data attackers can get around the defence mechanisms in place in the Linux kernel. The authors of the paper demonstrated this by exploiting the Linux AppArmor security module and bypassing the NULL Pointer deference mitigation on the victim's machine [24] The authors of the paper also demonstrated a second attack, they exploited the Linux kernel "proc" filesystem which acts as a bridge between kernel and user space. They then changed one of the pointer variables and pointed it to a "tty" buffer allowing them to keylog a user and pipe out confidential information that is typed on a user's keyboard [24]. Alibaba Tech from HackerNoon explains how you can exploit the Linux kernel using a kernel space mirroring attack. They explain how modern-day kernels run at a higher privilege level than user-mode programs and that the kernel memory space is invisible and not directly accessible to user programs for security reasons.

They explain that for ARM processors there are special hardware features that can be used to break this memory protection and allow ordinary user programs to directly access the kernel memory space in user mode which breaks the isolation of kernel space and user space [20].The kernel exploit shown is done by exploiting the "inotify_handle_event" function in the Linux kernel. A filename can be passed to the function causing the length to be noted and a buffer allocated that can store the full filename. The file can then be renamed on the code path to cause a buffer overflow in the kernel heap. This allows an attacker to overwrite special kernel objects that are also on the heap, by modifying these objects an attacker can exploit a system by changing data structures in the kernel causing

behaviour to change. [20]. The ARM processor design was changed and a PXN(Privileged Execute Never) mitigation was added to the processor architecture, this helped prevent some vulnerability exploits that took advantage of the way ARM processors were designed. This update prevented user-mode code from being executed from kernel land [20]. Alibaba Tech has shown that security in processors is even more important than security in the kernels themselves, as hardware flaws are harder to solve.

Modern day processors such as Intel and AMD provide protection rings. You have ring 0 to ring 3 with ring 0 being the most privileged ring. These protection rings are built into the processor and are used by the operating system to ensure user programs cannot affect kernel space or perform illegal operations. At any given time x86 processors are running in a given protection ring that determines what the code that is running can and cannot do. Most modern kernels only use ring 0 and ring 3 and rings 1 and 2 are ignored [4]. About fifteen machine instructions can only be executed while the processor is in ring 0. Many other machine instructions have limits on their operands. Attempting to run a ring 0 restricted CPU instruction whilst the processor is not in a ring 0 state will result in general protection fault exception interrupt to be executed [4]. The CPU privilege level has nothing to do with operating system users such as root privileges. Whether you are root, guest, or administrator it does not matter, they are not related to the CPU privilege level. All user programs and user code run on ring 3 meaning the processor is in an unprivileged ring and any dangerous instructions run from a user program will result in a general protection fault. All kernel code is run on protection rings zero, one and two [4]. These protection rings exist in modern day processors to protect the kernel from user programs. Access to memory and I/O can be restricted and user mode (code run in ring 3) can do nothing outside of its self without first calling the kernel and asking it for assistance. User programs cannot open files, send network packets, print to the screen or allocate memory. User processors run in a sandboxed environment setup by the kernel. The modern kernels make full use of the protection modern processors provide in the hardware. It is impossible by design for a processor to leak memory beyond its existence or leave open files after it exists. All the data structures that control memory, open files

cannot be touched directly by user code. Once a process is terminated it is unloaded by the kernel [4]. Windows 95 and Windows 98 crashed so much because important data structures were left accessible to user mode(ring 3) code for compatibility reasons [4]. The CPU protects memory at two crucial points, when a segment selector is loaded and when a page of memory is accessed with a linear address. Segmentation and paging models are both involved [4]. The Intel programmer's manual confirms what Durate has written. In protected mode, the Intel architectures provide a protection mechanism that protects both at the segment and page level. This protection mechanism provides the ability to limit access to certain segments or pages based on privilege levels. No CPU instruction may write into a data segment if it is not writable, also no instruction may read an executable segment unless the readable flag is set. The processors segment-protection mechanism has 4 privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges [9]. The operating system kernel should be run in level zero, whilst operating system services should be run in level 1 and level 2. Finally, user applications should be run in Level 3 [9]. The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level. The CPL is treated slightly differently when accessing conforming code segments. Conforming code segments can be accessed from any privilege level that is equal to or numerically greater (less privileged) than the DPL of the conforming code segment. Also, the CPL is not changed when the processor accesses a conforming code segment that has a different privilege level than the CPL [9]. All protection violation's result in an exception being generated that should be handled by the corresponding interrupt handler setup by a kernel [9].

2.3 Kernel Life Cycle

When you first turn on your computer the BIOS(Basic Input Output System) begins to initialize the screen and keyboard and test the main system memory. At this stage the

machine does not have any access to mass storage media, the information about the current date and time are loaded from the CMOS values. When the hard disk and its geometry are identified the system passes control from the BIOS to the boot loader. [18]

The boot loader is the first physical 512-byte data sector of the hard disk. It is loaded into memory. As the bootloader occupies the first sector(512 bytes) of hard disk storage it is referred to as the MBR(Master Boot Record). The bootloader its self is what passes control to the operating system. [18] In most cases the bootloader will start loading the kernel fundamentals and then pass control to the kernel for further loading. The BIOS will load the bootloader, the bootloader will load part of the kernel and the kernel will load the rest of the operating system.

Chapter 3

Methodology

A kernel was created for this paper to answer important research questions and demonstrate how a kernel works, the source code for the kernel can be found here: https://github.com/nibblebits/COS32. The kernel created for this paper is called COS32 which stands for COMPUTER OPERATING SYSTEM 32 bit. COS32 is a simple multitasking kernel that can run multiple processes and is written in the C programming language.

COS32 shares similar design patterns that are found in the Linux kernel. In COS32 we follow a similar design pattern as the Linux kernel when using file operations. The Linux kernel implements a virtual filesystem layer which allows filesystems to be installed during operating system runtime, the filesystem modules do not have to be compiled statically. COS32 also has a similar interface. Virtual file system layers written in the C Programming language are achieved through the use of function pointers and abstraction.

3.1 Intel processor overview

Before understanding how kernels work, some basic understanding of how the target processor works is essential. Kernel development involves understanding hardware and the processor.

3.1.1 Registers and internals

The Intel processor has several general purpose registers RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI these registers are 64-bit's in size and are used for general numeric operations such as adding two numbers together. The registers can hold up to 64-bit values and each have their own special purpose. The registers are made up of sub registers, of sub registers. RAX(64-bit) for example is split into EAX(32-bit), AX(16-bit), AH(8-bit) and AL(8-bit). AH and AL make up AX. EAX is made up of 16 bits and AX. RAX is made up of 32-bits and EAX. This design means if we change the AL register we are actually changing the lower 8 bits of the RAX or EAX or AX register. Since the RAX register is made up of all these sub registers. The same design is applied for each general purpose register. RCX is made up of ECX, CH, and CL. RDX is made up of EDX, DX, DH and DL.

3.1.2 Intel processor modes

When you first boot your computer it is put into what is known as real mode, this is a compatibility mode designed so you are still able to run 16-bit software on your Intel processor. In real mode we have access to the AX, CX, DX, BX, SP, BP, SI AND DI general purpose registers. When we are in real mode we can then put the processor into protected mode which puts the processor into a 32 bit state, allowing us to work with 32-bit CPU registers, from protected mode we can then put the processor into long mode which is a 64 bit state allowing us to use the CPU's 64-bit registers and instructions.

3.1.3 Real mode

This compatibility mode gives access to only the 16-bit registers of the Intel CPU this is the mode that MS-DOS would be able to run on. Real mode exists to be a compatibility layer to allow older software made for the older versions of Intel processors to still be able to be run. In real mode we can access only up to 1MB of memory in total. Since real mode uses only 16-bit registers if you wish to access more than 65KB of ram you are required

to use the legacy segmentation memory model that was used frequently in the MS-DOS era [8].

The segmentation memory model takes a segment and an offset to compute an absolute address, in real mode we have several segment registers, CS(Code Segment), DS(Data Segment), ES(Extra Segment) and SS(Stack Segment). Depending on the CPU instruction you are using will determine which segment register is used. If you are executing instruction 0x100 the CPU will actually be executing the physical address CS * 16 + 0x100. So if the CS(Code segment) is equal to 0x200 and your IP(Program Counter) register is set to 0x100 you would actually be executing the physical address 0x200 * 16 + 0x100 = 0x2100 in memory.

For using the Intel's "mov" instruction for moving data from a CPU register to memory the DS(Data segment) register would be used. If we imagine the instruction "mov word[0x100], AX" we are not moving the value of the "AX" register into address 0x100 we are moving the value of the "AX" register into address (DS*16)+0x100.

3.1.4 Protected Mode

Protected mode in modern Intel processors allows us to use 32-bit general purpose registers ECX, EDX, EBX, EAX, ESP EBP, ESI, and EDI. This mode also allows us to use the 16 bit registers available in real mode. EAX for example is made up of 16 bits and the AX register as explained in the subsection above named "Registers and internals".

When in protected mode we can no longer use the segmentation memory model provided by real mode, the segment registers become selectors. A selector points to a Global Descriptor Table which explains how we are able to access memory if our segment register is equal to that particular selector. With a GDT(Global Descriptor Table) entry we are able to tell the processor if a particular region of memory is readable or executable. We are also able to specify which security ring level is required to access this memory [13]. See section 3.2 for more information on the protection mechanisms provided by Intel.

With protected mode we also have access to paging which allows us to map virtual memory

addresses to physical memory addresses, this feature is essential for modern day security and also allows us to create the illusion of memory that does not exist. Read more on section 3.3.

Protected mode is a 32 bit mode and allows us to access up to a maximum of 4GB of installed RAM, we also have virtual addressing up to 4GB [15].

3.1.5 Long Mode

Long mode is a 64-bit processor mode, when we are in long mode we have access to the general purpose registers RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI and all sub-registers. Whilst in 64-bit long mode we also have access to far more memory than the 4GB memory we can access in 32-bit protected mode. We have access to a maximum of 16-exabytes of addressable memory, this is the maximum value a 64-bit register can hold. Although many Windows operating systems will only work with 128GB of memory, and a maximum of 2TB of memory for Windows 10 Pro 64-Bit [2].

Current Intel 64-bit processors support 48-bit virtual addressing and 256-TiB virtual address spaces. So although in a 64-bit processor we can write values as high as 16-exabytes due to the lower addressing scheme for paging being only 48-bit's we are only able to access 256-TIB virtual address spaces. Intel has written documentation for an extension that will allow for 57-bit addressing which will allow you to address up to a maximum of 128-PiB [15].

3.2 Kernel protection mechanisms

Protection mechanisms need to be provided by a kernel to protect its self and protect user programs from damaging each-other in memory. In older processors such as the Intel 8086 little hardware protection was available for kernels to protect themselves from user programs other than interpreting the program rather than executing it directly on the processor. An MS-DOS user program could be executed and modify the kernel machine code in memory and little could be done about it. The lack of security in older processors

became a huge problem which is why in 1982 Intel's protected mode was first introduced with the release of Intel's 80286 processor a 16 bit microprocessor with memory protection capability. [1]

3.2.1 How do you protect a kernel?

Modern Intel processors have several protection mechanisms, which range from protection rings to preventing access to certain memory by using the paging mechanisms in modern processors. In modern Intel processors you have four protection rings. Ring 0 provides unrestricted access you can call any CPU instruction without worrying about a General Protection Fault. Rings 1 and 2 are mostly used for device drivers and this ring offers some protection but not as much protection as ring 3. Ring 3 is for user programs, this is a highly restricted protection ring, access to supervisor memory is disallowed, access to sensitive CPU instructions that can modify the page table, or other processor states is disallowed [17]. Attempting to break these security rules for example by executing a privileged instruction in ring 3 will result in the processor throwing a General Protection Fault exception. This would then invoke the kernel interrupt handler for dealing with general protection faults. A well made kernel would then terminate the offending program. The responsibility of using these protection mechanisms lies with the kernel. The kernel must use the privileged instructions provided by the processor to setup paging and lower the protection ring to ring 3 when executing unprivileged code. When we talk about Ring 3 another term used is "user land". Another term used to describe rings 0-2 is "kernel land". User land is commonly used to describe a processor that is in a ring 3 state and executing a user program. Where as kernel land generally means we are in the a less restricted state and are performing kernel operations.

3.2.2 How do kernels drop to lower privileged ring levels?

When your processor is in ring 0 (an unrestricted state) the kernel can reduce the ring level by faking an interrupt return. When in protected mode and the processor is interrupted, the processor switches back into ring 0 and then pushes the address of the next instruction

before the interruption, the code segment, flags, stack pointer and the stack segment. The kernel routine then begins executing. Now when the kernel routine has finished executing it can execute an ”iret” instruction which means return from interrupt. The data pushed to the stack during the processor interruption is now popped off and restored, we are then running the next instruction in the unprivileged code once again much like returning from a function call. This unprivileged code could be a user program for example. Faking an interrupt return is the equivalent of pushing to the stack, a custom return address (the address you want to be executed), the code segment/selector (ring 3), flags, stack pointer and stack segment. Then you just issue an ”iret” and the processor will come out of ring 0 and go into ring 3 as if it just returned from an interrupt even though it did not. We faked an interrupt return to trick the processor into lowering our privilege level. When an interrupt is invoked again we will be put back into ring 0 where the kernel interrupt handler we setup will being executing again [16].

```
; void task_return(struct registers* regs);
task_return:
    mov ebp, esp
    ; Let's access the structure passed to us
    mov ebx, [ebp+4]
    ; push the data/stack selector
    push dword [ebx+44]
    ; Push the stack pointer
    push dword [ebx+40]
    ; Push the flags
    pushf
    pop eax
    or eax, 0x200
    push eax
    ; Push the code segment
    push dword [ebx+32]
    ; Push the IP to execute
    push dword [ebx+28]
    ; Setup some segment registers
    mov ax, [ebx+44]
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    push dword [ebp+4]
    call restore_general_purpose_registers
    add esp, 4
    ; Let's leave kernel land and execute in user land!
    iretd
```

Figure 3.1: Assembly for getting to user land ring 3

3.3 Paging in kernels

Paging is an essential security and performance component of modern computing. With paging we can create what is known as virtual memory. Virtual memory is memory that may or may not even exist, we map virtual addresses to physical addresses. The virtual address we map must be aligned to the system page size. Page sizes can be 4096 bytes or 4 MB in size for Intel processors whilst in 32 bit protected mode. [22] Let's take the following example and assume the processor is configured to use 4096 byte pages, you could map virtual address 0x1000 to physical address 0x5000 and this would mean that when you access any address between 0x1000 and 0x2000 you would actually be accessing memory in physical address 0x5000 to 0x6000.

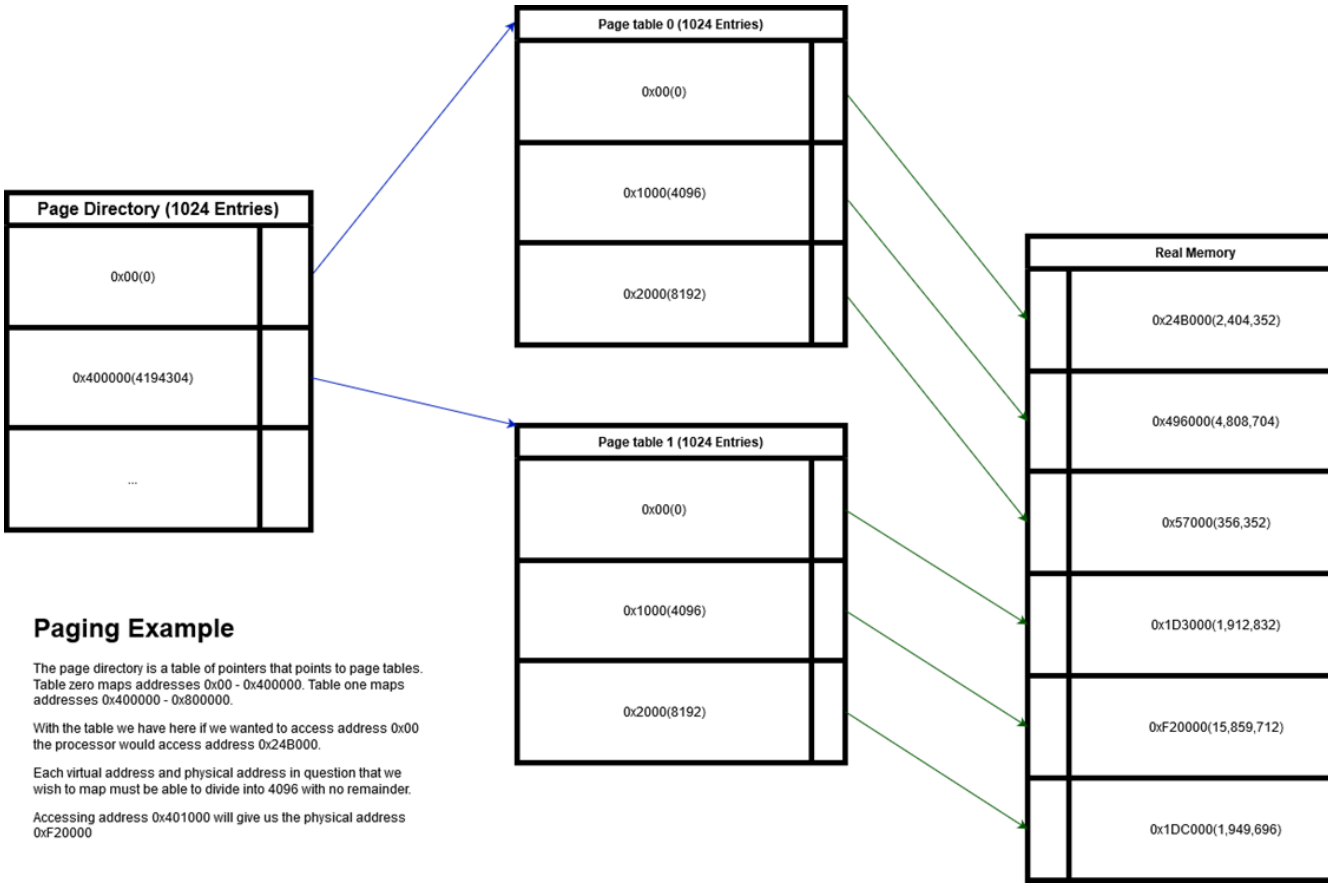


Figure 3.2: Paging Example With Page Directories And Page Tables

Paging in the x86 Intel processor is made possible with two components, page directories and page tables. The page directory contains pointers to page tables. The page tables contain pointers to real memory. Each page table entry represents 1024 pages, the page directory represents 1024 page tables [19].

A page simply represents 4096 bytes or 4MB of memory that has been virtualized. In the illustration Figure 3.2 you can see we map virtual address 0x00-0xFFF to address 0x24B000-24BFFF and virtual address 0x1000:1FFF to physical address 0x496000:496FFF. This means that if we access address 0x120 while paging was enabled we would actually be accessing physical address 0x24B120 not the address 0x120 in RAM. This is because the page tables are setup to point those virtual addresses to different physical addresses as represented in Figure 3.2.

We can only map virtual memory in chunks called pages in the example you have been shown the pages are 4096 bytes in size.

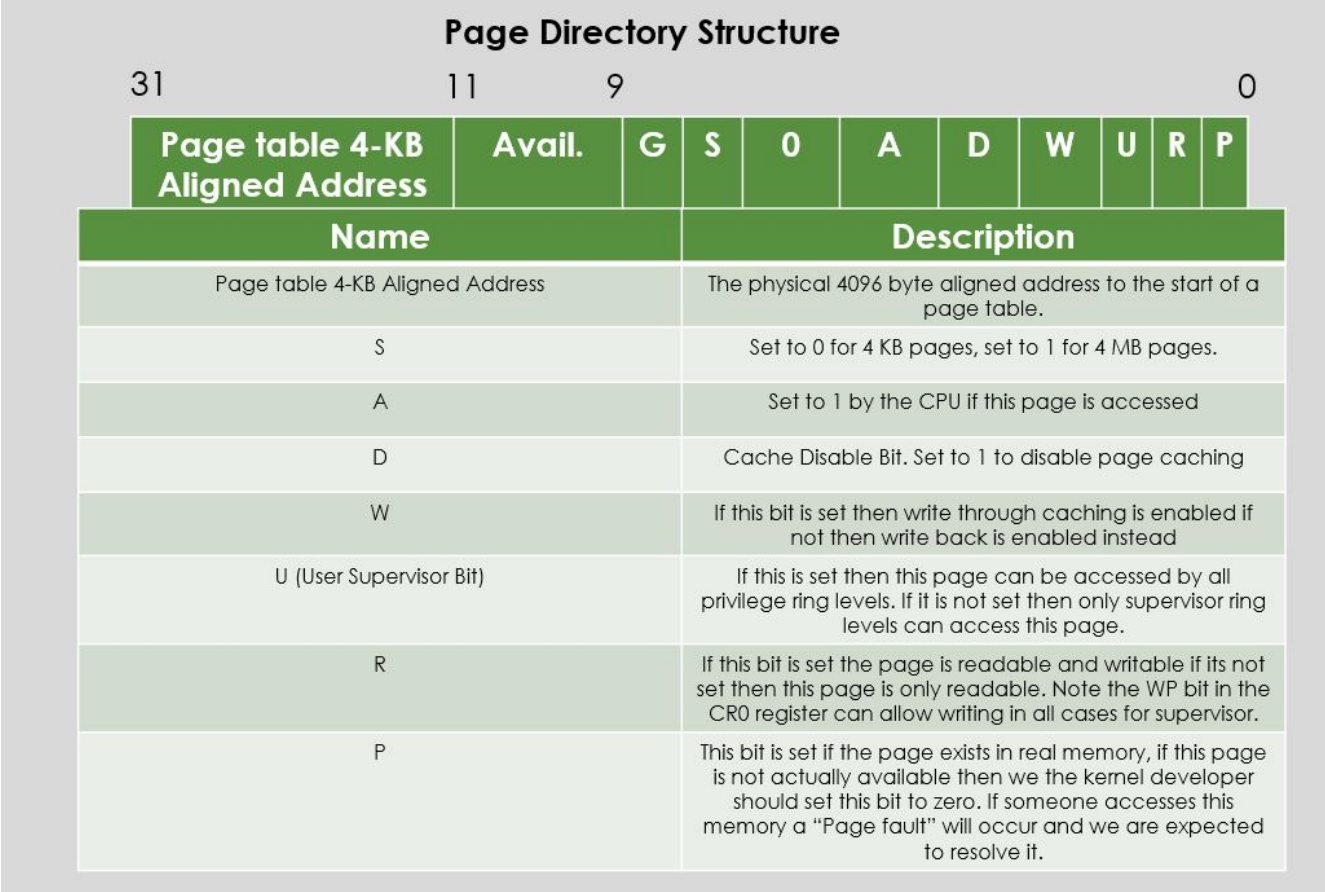
Page Table Entry Structure

31	11	9									0
Physical 4-KB Aligned Address	Avail.	G	0	D	A	C	W	U	R	P	

Name	Description
Page table 4-KB Aligned Address	The physical 4096 byte aligned address to the start of a page table.
G	Set to prevent TLB from updating the address in its cache if the CR3 register is reset.
D	If set this indicates the page has been written to.
A	Set to 1 by the CPU if this page is accessed
C	Cache Disable Bit. Set to 1 to disable page caching
W	If this bit is set then write through caching is enabled if not then write back is enabled instead
U (User Supervisor Bit)	If this is set then this page can be accessed by all privilege ring levels. If it is not set then only supervisor ring levels can access this page.
R	If this bit is set the page is readable and writable if its not set then this page is only readable. Note the WP bit in the CR0 register can allow writing in all cases for supervisor.
P	This bit is set if the page exists in real memory, if this page is not actually available then we the kernel developer should set this bit to zero. If someone accesses this memory a "Page fault" will occur and we are expected to resolve it.

Figure 3.3: Paging Page Table Entry Example

Page table entries also have permissions that control how a page of memory can be accessed, each page table entry in a page directory holds a physical 4096 byte aligned address that signifies where this page should point too in real memory. Other attributes of a page table entry are shown in Figure 3.3. An important part of the page table entry is the "R" bit this specifies weather the page is read only or not. You can also protect kernel code by unsetting the "U" bit which signifies that the page can only be accessed by privileged ring levels.



Page Directory Structure

31	11	9									0
Page table 4-KB Aligned Address	Avail.	G	S	0	A	D	W	U	R	P	

Name	Description
Page table 4-KB Aligned Address	The physical 4096 byte aligned address to the start of a page table.
S	Set to 0 for 4 KB pages, set to 1 for 4 MB pages.
A	Set to 1 by the CPU if this page is accessed
D	Cache Disable Bit. Set to 1 to disable page caching
W	If this bit is set then write through caching is enabled if not then write back is enabled instead
U (User Supervisor Bit)	If this is set then this page can be accessed by all privilege ring levels. If it is not set then only supervisor ring levels can access this page.
R	If this bit is set the page is readable and writable if its not set then this page is only readable. Note the WP bit in the CR0 register can allow writing in all cases for supervisor.
P	This bit is set if the page exists in real memory, if this page is not actually available then we the kernel developer should set this bit to zero. If someone accesses this memory a "Page fault" will occur and we are expected to resolve it.

Figure 3.4: Paging Directory Example

Page directories have similar attributes as the page tables they point too, see Figure 3.4

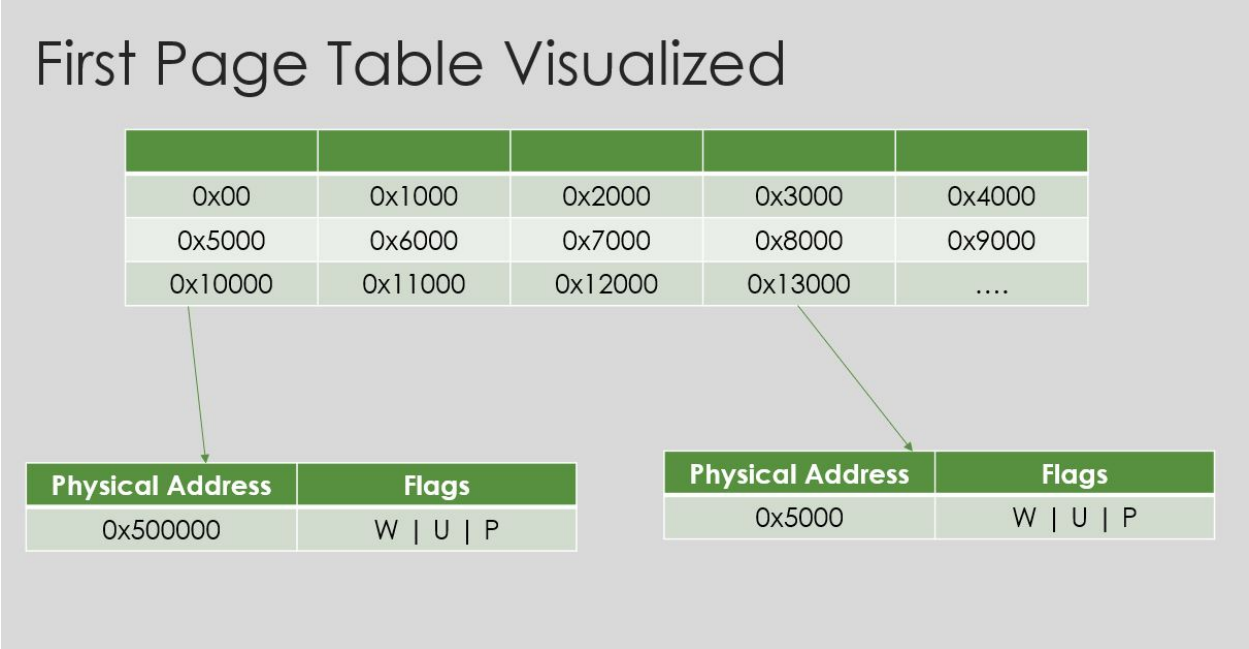


Figure 3.5: Page Table Visualized Example

You can see in Figure 3.5 how an individual page table looks in memory, you can see that each entry in the table represents an additional 4096 byte block. The actual memory for a page table entry in a page table contains the Physical address with the flags bitwise OR'ed on.

```
303  int paging_map(uint32_t *directory, void *virt, void *phys, int flags)
304  {
305      // Virtual address and physical must be page aligned
306      if (((unsigned int)virt % COS32_PAGE_SIZE) || ((unsigned int)phys % COS32_PAGE_SIZE))
307      {
308          return -EINVARG;
309      }
310
311      int res = paging_set(directory, virt, (uint32_t)phys | flags);
312      return res;
313  }
```

Figure 3.6: Paging Map Code Example

In Figure 3.6 you can see the C code from the COS32 project that shows how we can map a page in our page table. As can be seen we take our physical address and bitwise OR the flags that are shown in Figure 3.3

3.3.1 Mapping non-existant memory

In a 32-bit computer you can address up to 4GB of memory, when using paging even if your computer only has 1MB of memory or less. This is achieved by taking advantage of the ”P”(Present) bit found in page table entries, if this bit is equal to zero then any access to this page will cause a Page Fault exception. The kernel can then load what data should be in the page that was accessed off of the hard disk into memory and then change the physical address that the virtual page is pointing to so that the page table entry now points to the memory we just loaded. The page can then be marked as present by setting the P bit again. Once we then return from the page fault exception the user program will have access to that memory as if it had always been loaded, the user program had no concept that the memory it accessed was not even real at the time of access and that it was loaded by the kernel when the user program addressed it. This is another advantage of paging and with 64-bit computers the same mechanism can be applied however you are able to map to far higher address ranges, allowing you to map to a total of 256-TiB of memory.

When the kernel detects that it is running out of physical memory it can unload the pages, writing them back to the hard disk and marking the ”P”(Present) bit to zero again, the kernel has now drastically reduced the amount of memory that was loaded and the process

can start all over again, when a page is accessed the memory is loaded back off of the hard disk. This is also what swap files are used for, they are files on a hard disk that the kernel will use for writing and reading these pages [21].

3.4 Filesystems In Kernels

Filesystems are used to make sense of information on a disk. When we write data to a disk we write to the disk using a sector number. A sector represents a region of the disk, usually a 512 byte chunk. Disks have no concept of filesystems we just tell the disk which sector we want to write or read from and the disk controller will allow the kernel to load that sector into memory. Filesystems are simply structures on a hard disk that describe where in the disk you can find file data. Kernels are responsible for reading the disk, making sense of which filesystem is on the disk and then using the filesystem structures stored on the disk to locate existing files and to write new ones.

Virtual filesystems in a kernel are filesystems that make use of an abstraction layer that all virtual filesystems of the kernel must make use of. The kernel and all loaded filesystem modules are aware of this layer and how to communicate on this layer. The kernel and a filesystem use this layer to communicate with eachother, the layer allows the core of the kernel to know nothing about the details of the underlying filesystem and still be able to load and write files due to the abstracted interface they are both aware of. This abstraction acts as a middle man for communication between the kernel and the kernel filesystem modules [11].

```
37  struct disk;
38  typedef void *(*FS_OPEN_FUNCTION)(struct disk* disk, struct path_part* path, FILE_MODE mode);
39  typedef int (*FS_RESOLVE_FUNCTION)(struct disk *disk);
40  typedef int (*FS_CLOSE_FUNCTION)(void *private);
41  typedef int (*FS_SEEK_FUNCTION)(void *private, uint32_t offset, FILE_SEEK_MODE seek_mode);
42
43  /**
44   *
45   * Implementor should cast "descriptor" to its own descriptor private data. The private data should represent a local file des
46   * for its own file in its own filesystem, use size and nmemb to know how many bytes to read back to us in pointer "out"
47   */
48  typedef int (*FS_READ_FUNCTION)(struct disk *disk, void *private, uint32_t size, uint32_t nmemb, char *out);
49
50  /**
51   * Implementor should populate the "stat" variable provided to it, return zero on success
52   * \param private This is the private descritor related to the file that is being stat
53   */
54  typedef int (*FS_STAT_FUNCTION)(struct disk *disk, void *private, struct file_stat* stat);
55
56
57  struct filesystem
58  {
59      // Filesystem should return zero from resolve if the provided disk is using its filesystem
60      FS_RESOLVE_FUNCTION resolve;
61      FS_OPEN_FUNCTION open;
62      FS_CLOSE_FUNCTION close;
63      FS_READ_FUNCTION read;
64      FS_SEEK_FUNCTION seek;
65      FS_STAT_FUNCTION stat;
66      char name[20];
67
68  };
```

Figure 3.7: COS32 Virtual Filesystem Structure

In Figure 3.7 you can see many function pointer type definitions and a filesystem structure that contains variables of those function pointer type definitions. This is an interface that should be implemented by all virtual filesystems that are in the COS32 kernel, this interface is communicated with when opening, writing, and reading files. As the interface always stays the same it allows COS32 to talk with a varity of different filesystems whilst not knowing anything about them individually. All filesystems in COS32 have implemented this filesystem interface due to this abstraction the kernel core does not have to be aware of the internals of the filesystem it wishes to read and write too. When the kernel gets a request to open or read a file the kernel will simply call a function pointer in the filesystem structure of the mounted filesystem. This function pointer would have been set by the underlying file system functionality for a given filesystem during initialization of that filesystem when it was loaded.

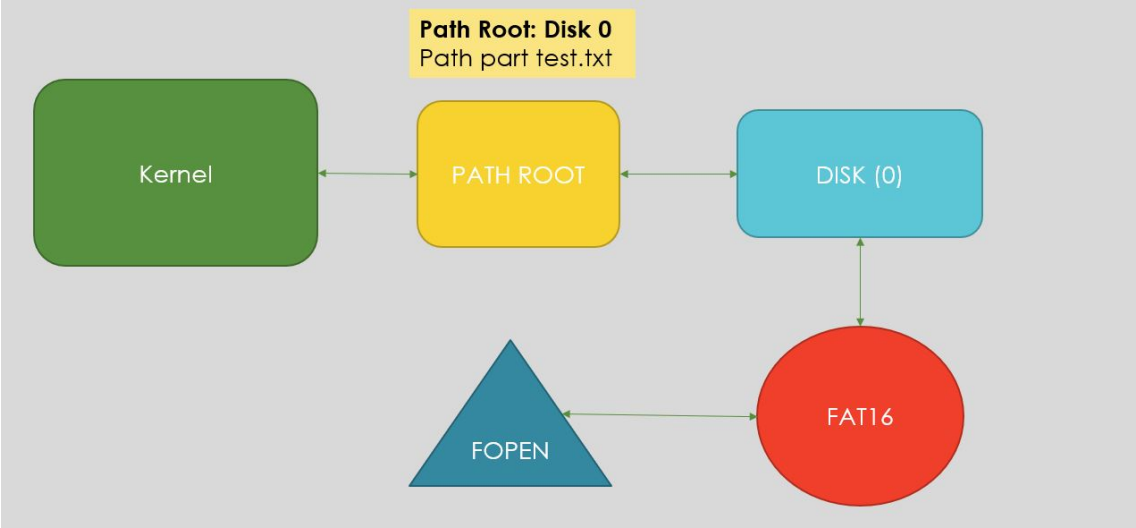


Figure 3.8: Illustration demonstrating the execution flow for opening a file in COS32

In COS32 we represent drives with numbers, in Windows you might see C:/file.txt to represent that the ”file.txt” file is in the C drive. In COS32 we use numbers. We would use the file path 0:/test.txt to load ”test.txt” from hard drive zero.

Figure 3.8 demonstrates what happens when the kernel wants to open a file. We have the FAT16 filesystem mounted on disk zero, so if we tried to open the file ”0:/test.txt” the kernel would call the ”open” function pointer in the filesystem structure that has been initialized on disk zero. See Figure 3.7. This would then communicate correctly with the underlying fat16_open function in our FAT16 filesystem driver, without the kernel ever being aware of what the FAT16 filesystem actually is. The kernel only knows that a filesystem is mounted on disk 0, it has no idea of the logic involved to open or read a file, that is the responsibility of the loaded filesystem module, that registered its self with the COS32 kernel.

```
2022  struct file_operations {
2023          struct module *owner;
2024          loff_t (*llseek) (struct file *, loff_t, int);
2025          ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
2026          ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
2027          ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
2028          ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
2029          int (*iopoll)(struct kiocb *kiocb, bool spin);
2030          int (*iterate) (struct file *, struct dir_context *);
2031          int (*iterate_shared) (struct file *, struct dir_context *);
2032          __poll_t (*poll) (struct file *, struct poll_table_struct *);
2033          long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
2034          long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
2035          int (*mmap) (struct file *, struct vm_area_struct *);
2036          unsigned long mmap_supported_flags;
2037          int (*open) (struct inode *, struct file *);
2038          int (*flush) (struct file *, fl_owner_t id);
2039          int (*release) (struct inode *, struct file *);
2040          int (*fsync) (struct file *, loff_t, loff_t, int datasync);
2041          int (*fasync) (int, struct file *, int);
2042          int (*lock) (struct file *, int, struct file_lock *);
2043          ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
2044          unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, uns
2045          int (*check_flags)(int);
2046          int (*flock) (struct file *, int, struct file_lock *);
2047          ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
2048          ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
2049          int (*setlease)(struct file *, long, struct file_lock **, void **);
2050          long (*fallocate)(struct file *file, int mode, loff_t offset,
2051                            loff_t len);
2052          void (*show_fdinfo)(struct seq_file *m, struct file *f);
2053  #ifndef CONFIG_MMU
2054          unsigned (*mmap_capabilities)(struct file *);
2055  #endif
2056          ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
2057                                     loff_t, size_t, unsigned int);
2058          loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
2059                                     struct file *file_out, loff_t pos_out,
2060                                     loff_t len, unsigned int remap_flags);
2061          int (*fadvise)(struct file *, loff_t, loff_t, int);
2062  } __randomize_layout;
```

Figure 3.9: Linux Kernel v5.13.10 file_operations structure

The Linux kernel also has a virtual filesystem layer part of it is shown in Figure 3.9. When we use pointers we can provide an abstraction between the kernel and kernel modules, this allows for filesystems and other drivers to be loaded and the kernel does not have to understand them. As they all use an interface that the kernel and all modules understand communication between modules can exist where the kernel has no knowledge of the module symbols and design.

Without abstractions such as described you would not be able to plug in a USB device, install a driver and have the system just understand it. The Windows and Linux kernel cores have no concept of the devices you plug into them, the core of the kernel will call the driver software functions through interfaces that are similar to the filesystem interfaces described. The driver code that exists in the loaded kernel module will then do the bulk

of the work and talk through other abstract interfaces if required [25].

3.5 PIC-Controller

In modern computers you have what is known as a PIC(Programmable Interrupt Controller), the PIC is a controller for handling keyboard presses, mouse clicks, hard-disk reads, and timer interrupts and more. The PIC should be mapped to an interrupt range by the kernel. Interrupts in the PIC are called IRQ(Interrupt Request's). In COS32 kernel we map the PIC to interrupt 0x20 which means the PIC IRQ's will offset from this interrupt. IRQ's are what are mapped to interrupts.

See the IRQ table here:

1. Programmable Interrupt Timer (IRQ 0)
2. Keyboard Interrupt (IRQ 1)
3. Cascade (used internally by the two PICs. never raised) (IRQ 2)
4. COM2 (if enabled) (IRQ 3)
5. COM1 (if enabled) (IRQ 4)
6. LPT2 (if enabled) (IRQ 5)
7. Floppy Disk (IRQ 6)
8. LPT1 / Unreliable "spurious" interrupt (usually) (IRQ 7)
9. CMOS real-time clock (if enabled) (IRQ 8)
10. Free for peripherals / legacy SCSI / NIC (IRQ 9)
11. Free for peripherals / SCSI / NIC (IRQ 10)
12. Free for peripherals / SCSI / NIC (IRQ 11)
13. PS2 Mouse (IRQ 12)
14. FPU / Coprocessor / Inter-processor (IRQ 13)

15. Primary ATA Hard Disk (IRQ 14)

16. Secondary ATA Hard Disk (IRQ 15)

In COS32 The ”Programmable Interrupt Timer” is mapped to real interrupt ”0x20” since we have mapped the PIC to interrupt 0x20 the table offsets from here. Likewise the ”Keyboard Interrupt” is mapped to interrupt 0x21. This means that in COS32 kernel when you press a key it will invoke the interrupt handler for interrupt 0x21, that is the keyboard interrupt in COS32 kernel for handling keyboard presses.

3.6 Different Program Types

3.6.1 Overview

In COS32 kernel you can load raw binary files or ELF file’s both are valid executable program formats. Depending which file you want to load determines how the program needs to be loaded into memory.

3.6.2 Raw Binary File

Raw binary files are files with no header or format. Raw binary programs would be classed as raw binary files. Raw binary programs are executable code and data that is loaded directly into memory and executed from the first byte loaded, their is no positioning or symbol information provided simply just raw data and executable code [23]. The COS32 kernel assumes that the code in a raw binary file has executable code from the first byte, the file is loaded into memory and executed.

3.6.3 ELF File

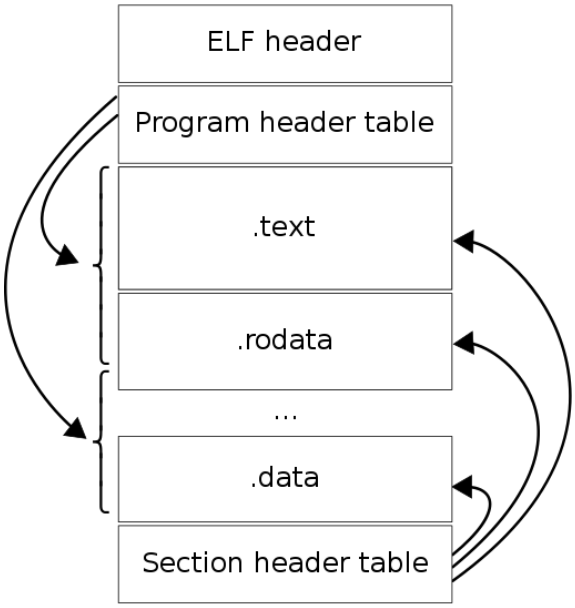


Figure 3.10: ELF File Format

Figure 3.10 provided under GPL License

https://commons.wikimedia.org/wiki/File:Elf-layout–en.svg by Surueña

ELF(Executable and Linkable Format) files are executable files that COS32 can load successfully. ELF files contain a ELF header file which describes the type of machine that this executable file can run on, also described in this header file is important information such as how many sections are in this file along with information on where in the file the program header and section header information can be located. Using the information provided to us in the ELF file we are able to successfully load the data and machine code from the executable file into memory. The ELF file also tells us if sections of code should be writable or not. This allows ELF32 to protect the ELF executable file data such as constant strings from being overwritten accidentally from the ELF executable file code due to a bug in the program we are loading. Executable file formats assist kernels in loading executable files correctly and ensuring programs cannot accidentally damage each-other. The kernel is assisted from file formats such as ELF as the ELF file contains attributes to tell the kernel how memory should be loaded and protected for a given executable program.

3.7 Multi-tasking

3.7.1 What is multi-tasking?

Multi-tasking is where the kernel will schedule CPU time between multiple processes, it is because of multi-tasking why you are able to open multiple programs on your computer and see them preform at the same time [14]. When a computer has only one processor core multi-tasking can still be preformed, the kernel switches the address the processor is executing so fast that it appears that everything is happening at once even though on a one core processor only one instruction is running at a time.

On a multi-core processor a kernel can use other cores two ensure that multiple instructions are executed at the same time for best performance.

3.7.2 How is Multi-tasking implemented in COS32 Kernel?

COS32 kernel implements multi-tasking by taking advantage of the PIC IRQ(Interrrupt Request) code zero. This IRQ is the timer interrupt, every couple milliseconds it interrupts the processor and calls the interrupt routine setup by the kernel responsible for handling the timer interrupt. The COS32 kernel then switches to the next process.

3.7.3 Switching to the next process explained

When any interrupt to the COS32 kernel takes place, the kernel saves the last running tasks general purpose registers and also the address that should be executed when it returns from the interrupt. This allows us to restore the state of the task in the future if it is needed.

In the event of a timer interrupt we would now switch to the next task by taking the page directory of the saved task and setting our current page directory back to that one. Switching page directories ensures we have now remapped the virtual memory of the user process so the system is seeing the system memory as that user program should. Next the saved registers of the next task are extracted from the saved task queue.The processor general purpose registers are now set to those saved registers. Now that the state of the next task

in the queue has been partially restored. The next step is to preform a task return to drop back into user land. This is achieved by faking an interrupt return just as was done to get into user land in the first place as explained in section 3.2.2.

3.7.4 Multi-tasking Breakdown

Assume a situation where two user programs/tasks are running. The process to switch between these tasks goes as follows.

1. The timer interrupt (IRQ 0) gets invoked (as it does every few ms)

2. The kernel is switched out of ring 3 back into ring 0 and the kernel routine for the timer interrupt begins to execute

3. The kernel saves the registers and state of the programs task that was last running before this interrupt

4. The kernel takes the next task from the task queue

5. The kernel extracts the registers and state of that task and restores the processors registers and the page tables of this particular task

6. The kernel now fakes an interrupt return to drop back into executing where that next task was previously paused. (Paused meaning registers saved and execution diverted to the kernel or another program)

7. The user task was now switched and will continue executing until the next timer interrupt where it will then switch back to the other task using the same process.

At this point the task was switched, this happens so fast that it provides the illusion to an observer that everything is happening at once. In reality on a one core processor only one instruction is being executed at any single time.

3.8 Program Termination

To terminate an application in COS32 the processor must first be executing in kernel land. Once in kernel land their is no concern about being interrupted through a keyboard interrupt or a timer interrupt and the kernel has full privileges when accessing CPU instructions. Once in kernel land the unloading function must be executed, this function is called ”process_terminate” in COS32 and all that it does is call ”process_free” which will unload the given process. See figure 3.11.

```
507  int process_terminate(struct process *process, int error_code)
508  {
509      // Free the current process
510      process_free(process);
511      return 0;
512  }
```

Figure 3.11: The function responsible for terminating programs in COS32 kernel

The process_free function will determine if the program loaded is a raw binary program or an ELF program, depending on the type of program it is determines how the kernel will unload it. If its an ELF program then the kernel must unload the ELF file by deleting the data that was loaded from the ELF file. If its a raw binary then the kernel can just delete the loaded data directly. All process memory allocations not deleted by the process must also be deleted to prevent a memory leak in the COS32 operating system. The task of the process can then also be deleted. Once the process has been completely unloaded from memory the kernel can then remove it from the process list. If the process that is being unloaded is the current process then the process has to be switched to avoid the kernel returning to a user space application that is no longer loaded in memory and is now non-existent. See the code below for a deeper understanding of how processes are unloaded in COS32.

```
void process_free(struct process *process)
{
    struct process *next_process = process_get_first_ignore(process);
    if (!next_process)
    {
        panic("No more processes available to switch to\n");
    }
    if (process_current() == process)
    {
        process_switch(next_process);
    }
    process_terminate_subprocesses(process);
    process_free_allocations(process);
    task_free(process->task);
    process_free_data(process);
    if (!(process->flags & PROCESS_USE_PARENT_VIDEO_MEMORY))
    {
        video_free(process->video);
    }
    if (process->flags & PROCESS_UNPAUSE_PARENT_ON_DEATH)
    {
        process_wake(process->parent);
    }
    kfree(process);
    processes[process->id] = 0;
}
```

Chapter 4

Results

The experimentation with kernel development by creating the COS32 kernel has shown that processors have evolved to become more secure, with higher performance. The security built into the processors of today did not exist at the time the first MS-DOS operating system was released. Even in 1982 when protected mode was first introduced it was not widely implemented until years later.

The COS32 kernel has shown that processors have evolved with kernels, each time a new security or performance feature is created in a processor future kernels implement it. This report and study shows how important security in processors and kernels is. Without kernel's using processors protection mechanisms to protect user-programs from each other we would still be in the days when a user program could hijack a kernel.

This report has addressed the following questions

1. How do you create a kernel?
2. What are the components of a kernel?
3. Why are kernels important?
4. How are user programs isolated from other user programs, what role does the kernel play in this?
5. What is memory protection and how can paging and virtual memory help with this?

6. How did kernels from the 1980s and early 1990s work and how they are they different from the kernels of today?

7. The difference between 16-bit, 32 bit and 64-bit kernels.

Chapter 5

Conclusion

This report has demonstrated the history of kernels and why kernels and processors evolved to have stronger security capabilities, points made in this report have been proven by the creation of the COS32 kernel which is a 32-bit protected mode kernel. Readers should now have a greater understanding of what file systems are and how virtual file systems can be used as an abstraction interface to allow kernels to support an infinite amount of loadable file systems. Readers should now understand that in the Linux kernel virtual file system layer is used for the purpose of supporting new and prior filesystems without needing to rewrite the Linux kernel core code everytime a new filesystem should be introduced.

You the reader should now have an understanding of how protection rings in a processor can help protect malicious code from executing memory that it should not have access too. You should now understand how paging can provide the illusion of more memory than exists in a system and how by using paging we can choose what memory user programs see.

References

[1] Singh Avtar. “8086 And 80286 Microprocessors: Hardware, Software, and Interfacing”. In: (1990).

[2] Tom Bauer. *Maximum addressable memory under the current operating systems*. Accessed on 20/09/2021. 2020. URL: https://www.compuram.de/blog/en/how-much-ram-can-be-addressed-under-the-current-32-bit-and-64-bit-operating-systems/.

[3] Caugh. “An Overview Of Microsoft Disk Operating System”. In: ().

[4] Duarte. “CPU Rings, Privilege, and Protection”. In: (2008).

[5] Dale Fisk. “THE EVOLUTION OF OPERATING SYSTEMS”. In: (2005).

[6] Kroah-Hartman Greg, SuSE Labs, and Novell Inc. “Linux Kernel Development - How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It”. Suse, 2007, pp. 239–244.

[7] P. B HANSEN. “THE EVOLUTION OF OPERATING SYSTEMS”. In: (2000).

[8] SEN-CUO RO SEAU-CHUEN HER. “i386/i486 Advanced Programming: Real Mode Protected Mode Virtual 8086 Mode”. In: (2012).

[9] Intel. “Intel® 64 and IA-32 ArchitecturesSoftware Developer’s Manualn”. In: (2016).

[10] Shuaibing Lu, Zhechao Lin, and Ming Zhang. “Kernel vulnerability analysis: A survey”. In: Institute of Electrical and Electronics Engineers Inc., June 2019, pp. 549–554. ISBN: 9781728145280. DOI: 10.1109/DSC.2019.00089.

[11] Benjamin David Lunt. “FYSOS: The Virtual File System”. In: (2014).

[12] Oliver. *Walking through MS-DOS the latest featured repository on GitHub*. Accessed on 24/02/2021. 2018. URL: https://blog.sourcerer.io/featured-github-repository-ms-dos-5ecf27cacb38.

[13] OSDEV.ORG. *Global Descriptor Table*. Accessed on 19/09/2021. 2021. URL: https://wiki.osdev.org/Global_Descriptor_Table.

[14] OSDEV.ORG. *Multitasking Systems*. Accessed on 17/09/2021. 2021. URL: https://wiki.osdev.org/Multitasking_Systems.

[15] OSDEV.ORG. *Paging*. Accessed on 17/09/2021. 2021. URL: https://wiki.osdev.org/Paging.

[16] OSDEV.ORG. *Security*. Accessed on 19/09/2021. 2021. URL: https://wiki.osdev.org/Security.

[17] Tom Shanley. "Protected Mode Software Architecture". In: (1996).

[18] B. Sivaiah, T. S.N. Murthy, and T. Vandana Babu. "Boot multiple operating systems from ISO images using USB disk". In: Institute of Electrical and Electronics Engineers Inc., 2014. ISBN: 9781479923205. DOI: 10.1109/ECS.2014.6892602.

[19] Arash TC. "The Holy Book Of X86 Volume 2". In: (2017).

[20] Alibaba Tech. "Entering God Mode - The Kernel Space Mirroring Attacks". In: (2018). URL: https://hackernoon.com/entering-god-mode-the-kernel-space-mirroring-attack-8a86b749545f.

[21] Indiana University. *What is a swap file?* Accessed on 19/09/2021. 2018. URL: https://kb.iu.edu/d/aagb.

[22] Timo OAlanko AInkeri Verkamo. "Segmentation, paging and optimal page sizes in virtual memory". In: 1983.

[23] Van Wolverton. "Running MS-DOS: Version 6.22". In: (2003).

[24] J. Xiao, H Huang, and H Wang. "Kernel Data Attack is a Realistic Security Threat". In: *University of Delaware* ().

[25] Erez Zadok and Ion Badulescu. "A Stackable File System Interface For Linux". In: Columbia University.

[26] Nezer J. Zaidenberg and Eviatar Khen. “Detecting Kernel Vulnerabilities during the Development Phase”. In: Institute of Electrical and Electronics Engineers Inc., Jan. 2016, pp. 224–230. ISBN: 9781467392990. DOI: 10.1109/CSCloud.2015.91.