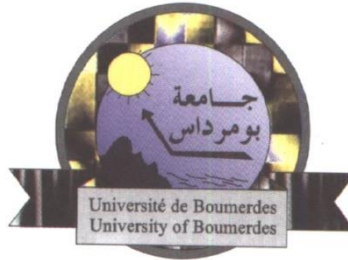


People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University M'Hamed BOUGARA – Boumerdès



Institute of Electrical and Electronic Engineering
Department of Electronics

Project Report Presented in Partial Fulfilment of
the Requirements of the Degree of

'Master'

In Computer Engineering

Title:

**Design and Implementation of an
Operating System For IA-32 Processors**

Presented By:

- **SEKHRI Aymen**
- **BOUDIAF Malek**

Supervisor:

Dr. NAMANE Rachid

Registration Number:...../2021

Abstract

An operating system is a set of software components that are used to manage the shared hardware resources between multiple programs, while maintaining an abstract interfacing layer to devices. This work discusses the approaches used to design the different components of such complex system, and how they are related to each other to construct layers for simple user programs to work in a secured system that is fair in sharing the CPU time and other hardware resources.

Our work presents first the theory and the background on memory management, interrupts, multitasking and modes of execution. Then it describes how these components are implemented in our operating system named *CyanOS*, and explains how to setup Intel 32bit processor's features and some other hardware buses and devices. At the end, it illustrates the way to how to modify and extend the functionality of the kernel, and how to write and compile a program running on this operating system.

Dedication

*“I would like to dedicate this report first and foremost to the sake of Allah our creator, my source of wisdom, knowledge and understanding and to my beloved parents, **Mahfoud** and **Ourdia Benmalek** for all their unconditional love , support and continuous encouragement throughout this academic journey.*

*A special feeling of love to my two little sisters, **Kaouthar** and **Tasnime** whose pure souls are my constant source of inspiration and to my uncle **Ahcene** who has been like a second father to me.*

*I dedicate this dissertation to my amazing friends whom I consider the best thing that happened to me throughout university. My roommate **Hicham Alla** one of the funniest people I know with whom I shared so many sleepless nights, my partner in this thesis **Aymen** who is the smartest person I know, my best friend **Houssam Ait Saadi** with whom I had so many memories, **Hamza Benrabah**, **Hicham Sahbi**, **Hamza Belmadani** and many others.”*

Malek

*“To my beloved parents, **Fouad** and **Naima**, and my little siblings for their constant support and love throughout my journey to survive this tough long life...*

*To my friends **Tayeb**, **Hicham**, **Malek** and **Zineddine** who had a huge impact on the development of my personality and cognition...*

To the people and random events that made me find my passion in computers, to who helped me find my Ikigai in an absurd world...

To who taught me how to be humble about what I know, and seek the endless journey to acquire what I don't know...

To who made me doubt my actions and be skeptical about my inner deep beliefs, for the sake of continuous self-reevaluations...”

Aymen

Acknowledgements

First and foremost, we would like to sincerely thank our supervisor **Dr. NAMANE Rachid** for his supervision and constant support. His valuable help of constructive comments and suggestions have greatly contributed to the success of this thesis work.

We also would like to thank **Andreas Kling** a blogger and a youtuber who created *SerenityOS* operating system, his work inspired the development of this project and his advices helped us move to the right track. He made us overcome the fear of initiating big projects, and taught us how divide and conquer complex tasks in programming effectively and with minimum efforts. He showed us how can you put your heart to build something big, without being rewarded except for the inspiration of many young programmers.

Finally, we acknowledge the work of **Robert Cecil Martin**, **Scott Meyers** and **Jason Turner** for their books and talks in CppCon conferences. They taught us about most modern C++ features and they can be used effectively, and how we can design a clean, solid and scalable software systems with a minimum technical debt.

Content

Abstract	I
Dedication	II
Acknowledgements	III
Content	IV
List of Figures	VII
List of Tables	IX
List of Abbreviations	X
General Introduction	1
Chapter 1: Theory and Background	2
1.1 Computers and Software	2
1.2 Computers before operating systems	3
1.3 Memory management.....	4
1.3.1 Primitive memory management.....	4
1.3.2 Segmentation	5
1.3.3 Paging	6
1.3.4 Virtual memory.....	7
1.3.5 Page tables and address translation	8
1.4 Interrupts	12
1.4.1 Interrupts and polling.....	12
1.4.2 Types of interrupts.....	12
1.5 Scheduling Algorithms.....	13
1.5.1 First-Come, First-Served scheduler	13
1.5.2 Priority based scheduler.....	13
1.5.3 Round Robin scheduler	14
1.5.4 Multi-level Queueing scheduler	14
1.6 Modes of execution	14
1.7 x86 instruction set and IA-32 processors	15
Chapter 2: Design and Implementation	17
2.1 Setting up IA-32 Protected Mode Features	17
2.1.1 Segmentation	18

2.1.2 Interrupts.....	20
2.1.3 Virtual Memory	23
2.2 Device’s Drivers.....	28
2.2.1 Intel Programmable Interrupt Controller (PIC 8259).....	28
2.2.2 PS/2 Keyboard.....	30
2.2.3 Peripheral Component Interconnect (PCI) bus.....	30
2.2.4 RTL8139 Ethernet Network Device.....	32
2.3 Multitasking	34
2.3.1 Kernel memory space	34
2.3.2 Processes and Threads	34
2.3.3 Task Synchronization	38
2.3.4 Interprocess Communication.....	40
2.4 User Mode.....	42
2.4.1 User mode and system calls.....	42
2.4.2 ELF executable loader	44
2.5 Heap Allocator	46
2.5.1 Fixed Partitioning	46
2.5.2 Dynamic Partitioning.....	46
2.5.3 Segregated Free List.....	48
2.6 Virtual File System.....	48
2.6.1 VFS Implementation.....	49
2.6.2 FileDescription	49
2.6.3 Handles	50
2.7 Kernel Architecture	52
2.7.1 Monolithic Kernel Architecture.....	52
2.7.2 Microkernel Architecture	52
2.7.3 Micro vs Monolithic kernel	53
2.7.4 Hybrid Kernel Architecture	54
Chapter 3: Results and Discussion.....	55
3.1 User program discussion: <i>shell</i>	55
3.2 User program discussion: <i>cat</i>	57
Conclusion and Future Work	60

Appendix A: Modern C++ Features	62
A.1 Templates	62
A.2 Lambda Expressions	62
Appendix B: Data Structures	64
B.1 Iterators.....	64
B.2 Vector	64
B.3 List.....	66
B.4 String	67
B.5 Stack.....	67
B.6 CircularBuffer	68
B.7 Bitmap.....	68
B.8 Result.....	68
Appendix C: System Calls.....	70
Appendix D: How to Compile <i>CyanOS</i>	72
D.1 Building cross compiler gcc.....	72
D.2 Building the operating system.....	73
Bibliography	74

List of Figures

Figure 1-1: Abstract view of the components of a computer system	3
Figure 1-2: Memory fragmentation	5
Figure 1-3: Segmentation with non-contiguous physical memory	6
Figure 1-4: Paging.....	7
Figure 1-5: Virtual memory using physical and secondary memory	8
Figure 1-6: Function of the MMU	8
Figure 1-7: Address translation.....	9
Figure 1-8: Address translation example	9
Figure 1-9: Implementing the translation lookaside buffer	11
Figure 1-10: Kernel mode and User mode.....	15
Figure 1-11: IA-32 Registers	16
Figure 2-1: IA-32 System-Level Registers and Data Structures	17
Figure 2-2: Flat Model	19
Figure 2-3: Segment Selector.....	20
Figure 2-4: Interrupt Procedure Call.....	21
Figure 2-5: Stack Usage on Transfers to Interrupt and Exception-Handling Routines	23
Figure 2-6: Segmentation and Paging.....	24
Figure 2-7: Virtual Address Translation to a 4-KByte Page using 32-Bit Paging.....	25
Figure 2-8: Paging-Structure Entries with 32-Bit Paging.....	25
Figure 2-9: Identity mapping	27
Figure 2-10: Using page fault in swapping.....	28
Figure 2-11: PIC 8259	29
Figure 2-12: Cascading two PICs	31
Figure 2-13: PCI structure	32
Figure 2-14: Enumerating all PCI devices.....	33
Figure 2-15: Higher Half Kernel model.....	34
Figure 2-16: Threads movement between scheduler lists.....	38
Figure 2-17: Spinlock pseudo code.....	39
Figure 2-18: Pipe Handles between two processes.....	41
Figure 2-19: Domain Sockets flow chart	42
Figure 2-20: System call execution.....	43
Figure 2-21: Sections and Segments.....	45
Figure 2-22: File in disk vs Program in memory	45
Figure 2-23: Dynamic allocation	47
Figure 2-24: Segregated free list.....	48
Figure 2-25: Virtual file system.....	49
Figure 2-26: Relationship between handlers and FileDescription	51
Figure 2-27: Monolithic kernel architecture	52
Figure 2-28:Microkernel architecture	53
Figure 3-1: Shell's pseudo code.....	56

Figure 3-2: Another shell's pseudo code	56
Figure 3-3: Navigate directories and execute programs in shell.....	57
Figure 3-4: Code snippet from cat	58
Figure 3-5: Using cat to read text files.....	59
Figure A-1: Template function	62
Figure A-2: Lambda expression example	63

List of Tables

Table 2-1: Protected-Mode Exceptions and Interrupts	22
Table 2-2: Format of a 32-Bit Page-Directory Entry that References a Page Table	26
Table 2-3: Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page.....	26
Table 2-4: Device connected to PIC	30
Table 2-5: Internal fields of a Process.	35
Table 2-6: Internal fields of a Thread.	35
Table 2-7: FSNode operations	50

List of Abbreviations

API	Application Programming Interface
CPL	Current Privilege Level
CPU	Central Processing Unit
DLL	Dynamic-link Library
FS	File System
GDT	Global Descriptor Table
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IA-32	Intel Architecture 32-bit
IDT	Interrupt Descriptor Table
IPC	Inter Process Communication
LDT	Local Descriptor Table
MMU	Memory Management Unit
NIC	Network Interface Controller
OS	Operating System
PCI	Peripheral Component Interconnect
PD	Page Directory
PDE	Page Directory Entry
PT	Page Table

PTE	Page Table Entry
RAM	Random Access Memory
RPL	Requested Privilege Level
TLB	Translation Lookaside Buffers
TLS	Transport Layer Security
VFS	Virtual File System

General Introduction

According to Moore's law, it is observed that the number of transistors in a dense integrated circuit doubles every two years; this means faster hardware with more features, and also means more complex devices and harder to configure. In the meanwhile, a new programmer might need to read the immensely large datasheets of all the devices installed on his computer just to write a program with a simple task. This inconvenience imposed the need of an abstract interfacing layer that programmers will be using to write simpler programs that do not have to be aware of the underlying hardware of the system, this interfacing layer is what is called an operating system.

The first chapter of this report presents a theoretical background about the different operating system designs and concepts. Chapter 2 describes the implementation of our *CyanOS* operating system, and gives arguments why our followed approaches are superior to the ones used in other existing implementations. Chapter 3 discusses the obtained results and explains the way some user mode applications are compiled, linked and executed. Finally, the report is ended with a conclusion and some suggestions for future works.

Chapter 1: Theory and Background

1.1 Computers and Software

Without its software, a computer is basically a useless lump of metal. With its software, a computer can store, process, and retrieve information; play music and videos; send e-mail, search the Internet; and engage in many other valuable activities to earn its keep. Computer software can be divided roughly into two kinds: system programs, which manage the operation of the computer itself, and application programs, which perform the actual work the user wants. The most fundamental system program is the operating system, whose job is to control all the computer's resources and provide a base upon which the application programs can be written. A modern computer system consists of one or more processors, some main memory, disks, printers, a keyboard, a display, network interfaces, and other input/output devices. All in all, a complex system. Writing programs that keep track of all these components and use them correctly, let alone optimally, is an extremely difficult job. If every programmer had to be concerned with how disk drives work, and with all the dozens of things that could go wrong when reading a disk block, it is unlikely that many programs could be written at all. [1]

Essentially, an operating system is a large and complex set of system programs that control the various operations of a computer system and provide a collection of services to user programs through an abstract interface of the underlying hardware resources. Since multiple programs can use these resources simultaneously, the operating system is also responsible for managing how resources are shared, i.e., sharing processor cores, RAM, hard disk, network interfaces, display device, keyboard, mouse... Therefore, any operating system should guarantee:

1. Availability of a convenient, easy-to-use, and powerful set of services that are provided to the users and the application programs in the computer system
2. Management of the computer resources in the most efficient manner

The services provided by an operating system are implemented as a large set of system functions e.g., scheduling of tasks, memory management, device management, file management, network management, and other more advanced services related to protection and security. Figure 1-1

shows a layered abstract view of the components of a complete computer system and the placement of the operating system in the latter.

1.2 Computers before operating systems

Before operating systems came to existence users and programmers used to directly interact with the bare computer's hardware where they needed to write very low-level programs that run directly on the CPU and that did everything including managing all the hardware resources. Users also needed to know all the small details about the hardware components and how they work which meant that if the latter changes the programs also needed to change accordingly. Concepts of multiprogramming and time-sharing were not possible at the time where the hardware supported only one program at a time - each user must wait until the previous program is done to "share" the hardware with other users. In conclusion, writing programs was incredibly complex and expensive and certainly not accessible for the average user.

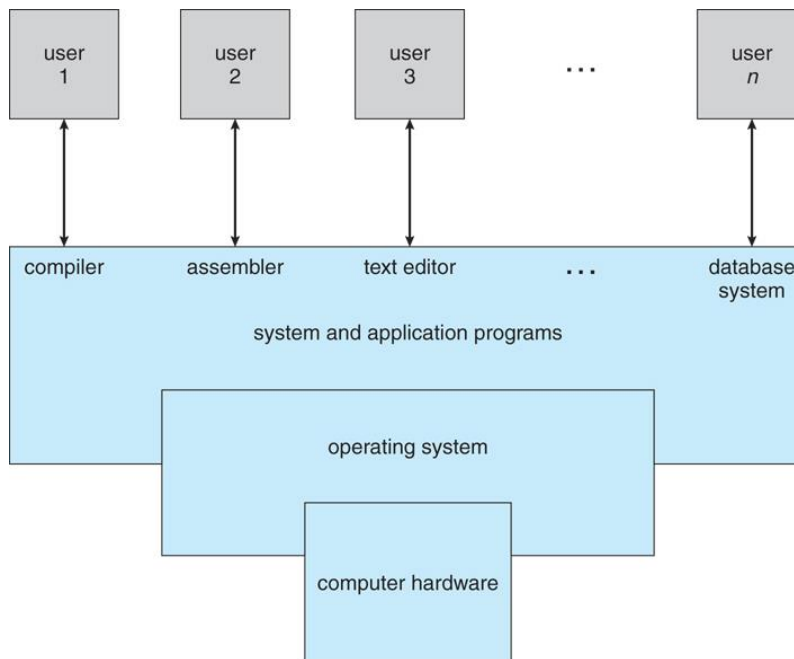


Figure 1-1: Abstract view of the components of a computer system

Each user had sole use of the machine for a scheduled period of time and would arrive at the computer with a program and data, often on punched paper cards and magnetic, paper tape or by setting a large set of on-off switches. The program would be loaded into the machine and the

machine would work until the program is completed or crashed. This really shows the importance of modern operating systems and the level of abstraction they provide for the user.

1.3 Memory management

Memory is the most important resource in a computer system thus; it must be carefully and wisely managed.

The part of an operating system that handles this resource is called the memory manager, it is responsible of allocating portions of memory for processes when needed and de-allocating it to be reused by other processes when there is no longer necessity for it, while it keeps track of all used or free memory regions. It is also responsible of transferring some portion of a memory that is owned by a process to a secondary memory storage like hard disk, whenever this process seems reasonable.

There are many memory management schemes from the primitive management like loading program directly to physical memory to the most sophisticated like *paging* and *virtual memory*.

1.3.1 Primitive memory management

The most primitive memory management is by loading all running programs directly into contiguous physical memory regions. When a program is no longer is used, the operating system will claim its memory and mark it as free to be used for the next problem. However, there are some flaws in this model; firstly, programs need to be aware about which address they will be loaded in at compile time. Secondly, there is a problem of fragmentation; when programs terminate and their memory is reclaimed, it may leave small portions of memory that another program may not fit in. Figure 1-2 shows some progression of creating and terminating processes that leaves the memory fragmented and cannot load a new program that its greater than 6 MB although there is enough total memory in the system.

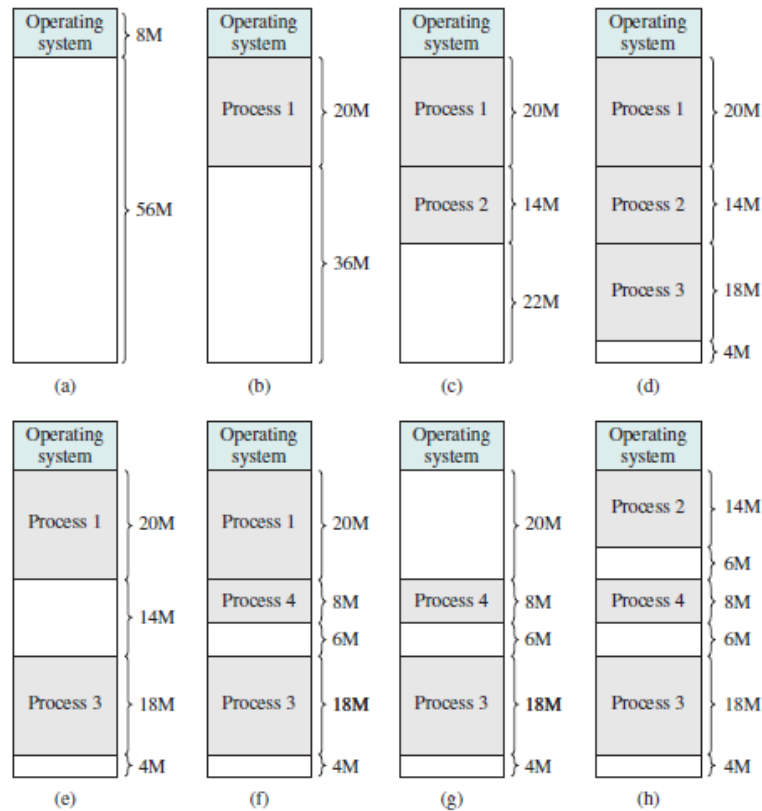


Figure 1-2: Memory fragmentation

1.3.2 Segmentation

Segmentation is dividing the physical memory into several different sizes regions, one for each process. The program will use offsets in these segments and does not have to be aware of the where the segment is in physical memory. The operating system maintains a map of segments to physical memory in a segment table. Every entry of the table contains the base address and the size of physical memory that corresponds segment. Segmentation can be used to map segments to either contiguous or non-contiguous physical memory regions depending on the system and processor. Segmentation solves the first flaw of primitive memory management discussed previously. However, it still suffers from the fragmentation. Figure 1-3 illustrates how semination works.

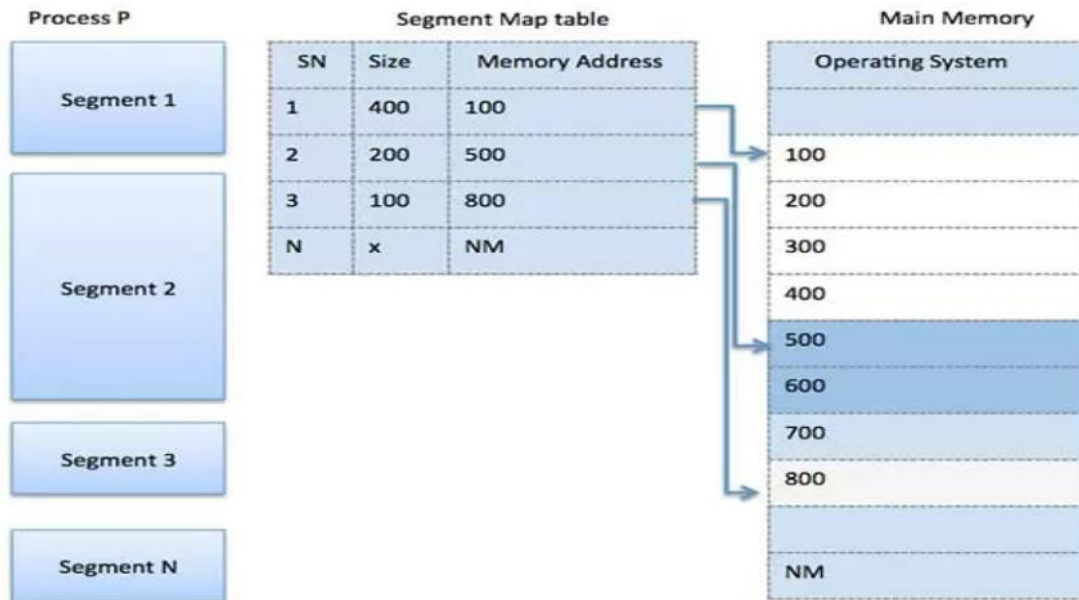


Figure 1-3: Segmentation with non-contiguous physical memory.

1.3.3 Paging

In order to attempt to solve the problem of fragmentation and for better memory utilization, a transition to noncontiguous memory managements techniques needed to take place. One of these techniques is *paging*. Paging is a memory management technique in which the address space of a process is divided into small fixed-sized blocks of logical memory called pages, each page is mapped to a physical memory block called a frame, both pages and frames have the same size which is chosen by the operating system and usually power of two. The frames allocated to the pages of a process do not need to be contiguous; in general, the system can allocate any unused frame to map a page for a particular process.

The operating system has a mapping table of page-frame for each process, making the address space of each process is independent while avoiding fragmentation. Figure 1-4 shows how paging works.

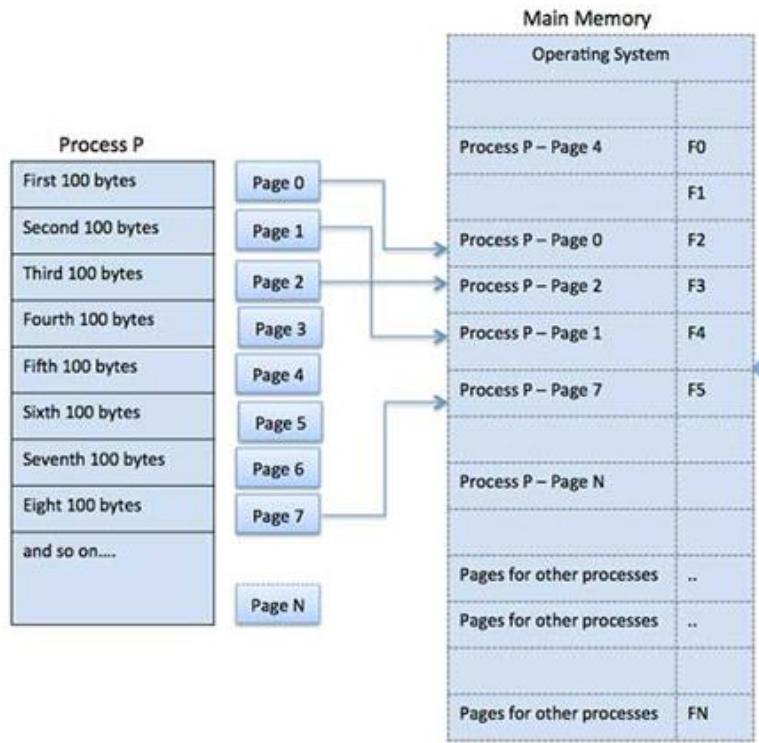


Figure 1-4: Paging

On computers without paging, physical memory is directly addressed by processes which raises the possibility of a process accidentally writing to another process's data which may corrupt it. Whereas, with paging, each process has an independent address space that is mapped to unique physical frames. This provides a memory protection by guaranteeing that physical address spaces do not overlap and that processes do not overwrite each other's data.

1.3.4 Virtual memory

The primary motive of virtual memory is to allow for processes to access more memory than the amount physically available through the use of secondary memory (disk) and a noncontiguous memory allocation scheme (usually paging). The memory manager sets up the disk to an extra physical memory, when the physical memory is about to run out. Figure 1-5 illustrates the use both physical memory and secondary storage.

In modern CPU's a *memory management unit* (MMU) is implemented into the hardware and it is responsible for the mapping of virtual addresses into the physical ones as shown in Figure 1-6.

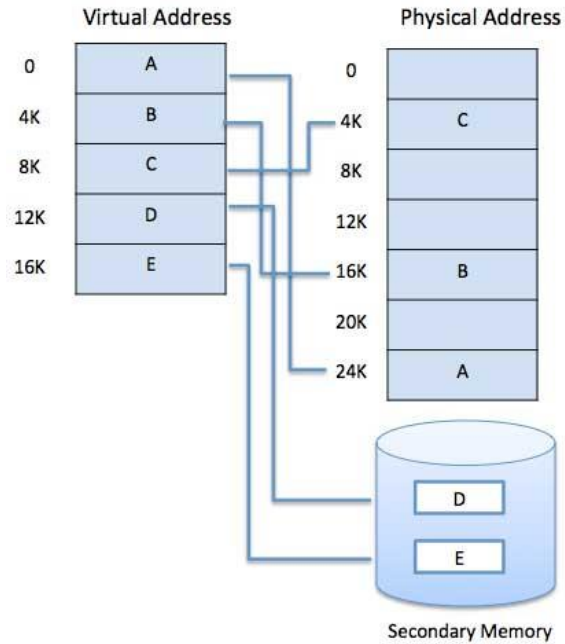


Figure 1-5: Virtual memory using physical and secondary memory

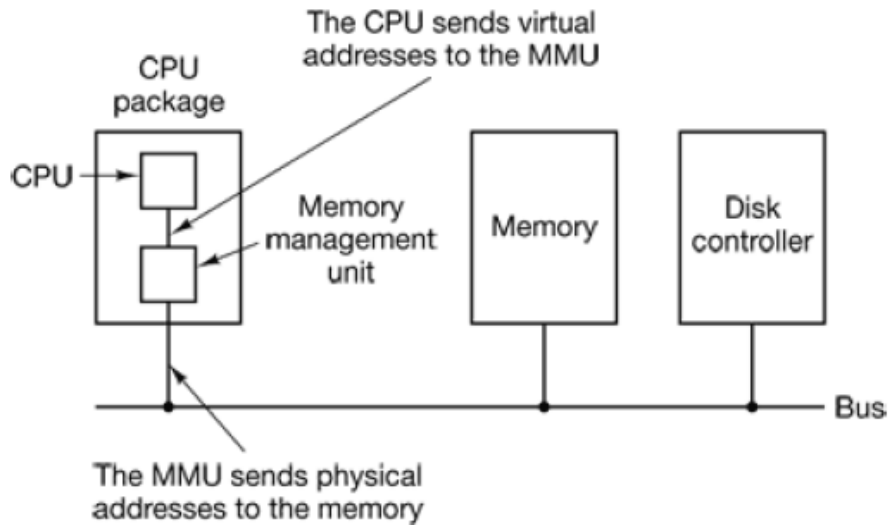


Figure 1-6: Function of the MMU

1.3.5 Page tables and address translation

A page table is a data structure that keeps track of all the mappings between virtual and physical memory. Each entry in the table contains two pieces of information: the virtual page number and the corresponding frame (physical page) number. It can be thought of as function that takes a

page number as argument and returns the frame number as output. Figure 1-7 illustrates the use of page table in the virtual address translation.

The virtual address is split into two fields, the high order bits represent the virtual page number for a virtual address and the low-order bits represent the offset of the address within the page itself. Figure 1-8 shows an example of address translation in the case of a 32-bit machine with 256MB of RAM and 4kb sized pages. There are 32bit virtual address and 28bit physical address, a 12 bit is used as an offset in the page or frame.

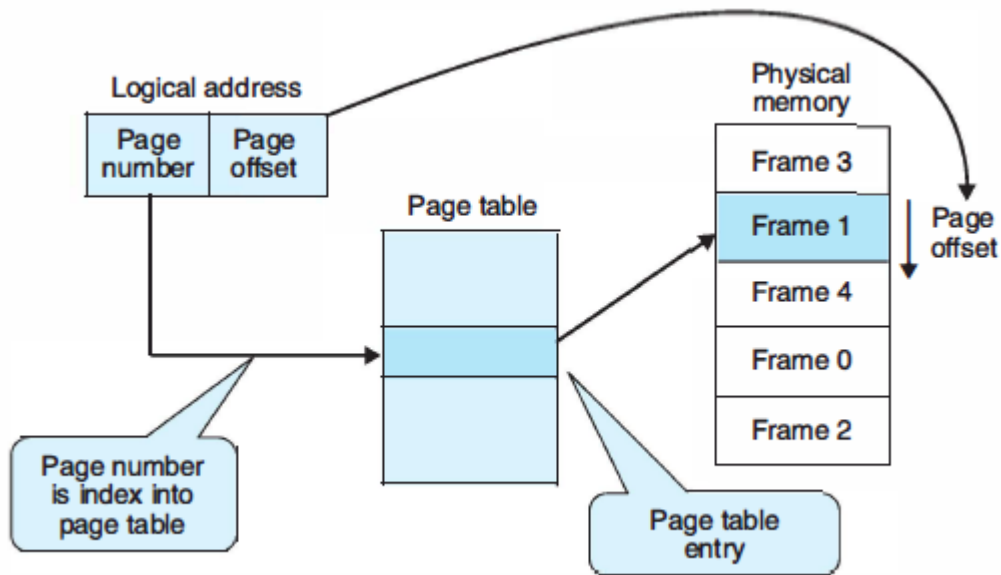


Figure 1-7: Address translation

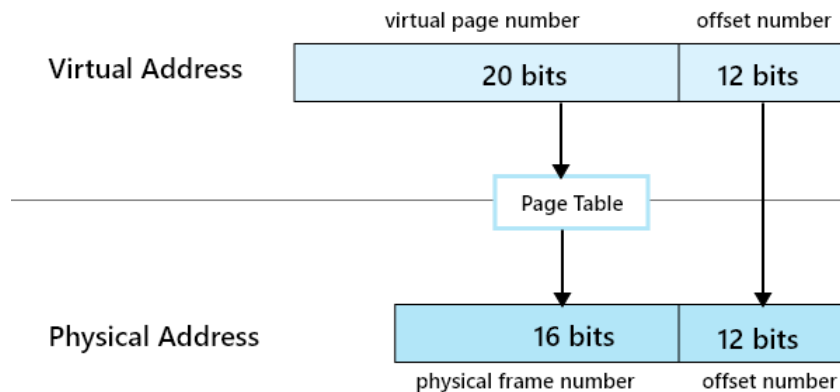


Figure 1-8: Address translation example

Each time a virtual address is referenced, the system performs a look up in the page table that corresponds to that particular process and checks if the page is available in physical memory

(RAM) if this is the case then the physical address is formed by combining the frame address and offset value. This address now can be used to address the main memory.

In the case where the page is unmapped and its corresponding frame is not in physical memory the CPU raises a *Page Fault* trap to the operating system which is an exception that requires the immediate attention of the OS which in turn will start a routine that swaps out a rarely used frame from the main memory to the secondary memory and fetches (swaps in) the referenced page in place of the freed frame, updates the entry in the page table and restarts the instruction from which the page fault was raised.

The concepts of extending the main memory using disk seems like a convenient solution to the problem of insufficient memory but reading from disk is significantly slower than accessing RAM. Consequently, handling a page fault can have a serious effect on performance especially in modern computers where CPUs are extremely faster than hard disks. An excessive rate of page faults puts the system into a state of *thrashing* where the system spends most of its time swapping pages rather than executing instructions.

In addition, page tables are stored in main memory and this introduces other issues:

1. Each reference to memory requires a virtual to physical memory mapping (several memory accesses on every reference). this process has an obvious effect on performance (bottleneck).
2. Page tables can get extremely large in size and each process must have one i.e., a 32-bit address space and a 4-kb page size will result one million page table entry for each process.

Even though these issues cannot be completely eliminated they can be reduced using some design considerations such as implementing *translation lookaside buffers* (TLB) to make mapping faster and designing smarter page replacement algorithms to reduce the problem of thrashing.

1.3.5.1 Translation Lookaside Buffers (TLB)

without the use of virtual memory, reading/writing data to physical memory needs only a single memory reference. With paging several memory references are needed which will have enormous impact on performance.

In real life systems, only a fraction of the page table entries is heavily used, whereas others are rarely referenced. This gave motivation to implement the **translation lookaside buffer** which is a hardware device that is similar to a cache memory and allows for mapping of frequently used pages without the need to access the actual page table in the main memory. TLBs are usually implemented in the MMU and they contain a small number of entries (no more than 64).

As illustrated in Figure 1-9 every time memory needs to be referenced the virtual page is first checked for availability in the TLB. If the page is present (**a TLB hit**) then its corresponding frame is read and used to form the physical address. Otherwise, a **TLB miss** will occur and normal page table mapping will take place, this in fact is slower than directly performing a page table mapping in the first place and this is the disadvantage of using the lookaside buffer.

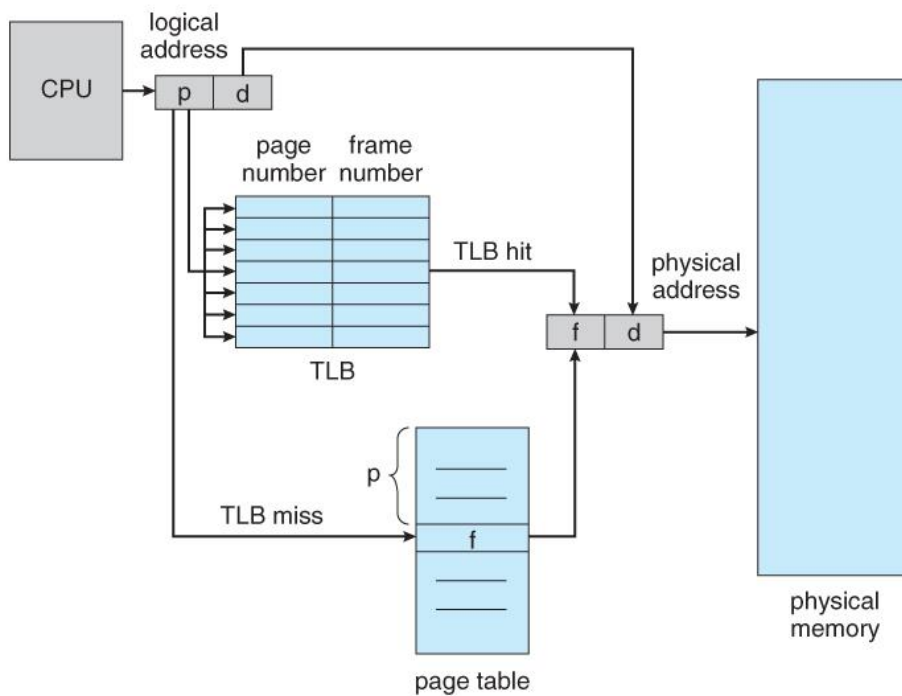


Figure 1-9: Implementing the translation lookaside buffer

1.3.5.2 Page replacement algorithms

Each time a page fault occurs, a page has to be swapped out from main memory to make space for the referenced page to be brought in. The decision of which page to remove from memory is the operating system's responsibility and it depends on the page replacement algorithm used.

The choice of which replacement algorithm to use is very important as it has direct relation to the performance of the system. ideally the goal is to evict a page that is the least used, this will decrease the chance of the system going into the *Thrashing* state and consequently a better performance. Thrashing occurs when a computer's virtual memory resources are overused, leading to a constant state of paging and page faults, inhibiting most application-level processing. This causes the performance of the computer to degrade or collapse. The situation can continue indefinitely until either the user closes some running applications or the active processes free up additional virtual memory resources. [2]

The ideal replacement algorithm to swap out the page that will not be used or referenced for the longest time, however, it is hard determine the behavior of the threads in the future. Therefore, there some more practical algorithms used in modern operating systems such as:

- First In First Out (FIFO)
- Least recent used (LRU) page replacement algorithm

1.4 Interrupts

The operating system is event driven and relies heavily on interrupts. An interrupt is a signal to the processor triggered by hardware or software indicating an event that needs immediate attention. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an Interrupt Service Routine (ISR) or Interrupt Handler. The ISR tells the processor what to do when the interrupt occurs. The interrupts can be either hardware interrupts or software interrupts. [3]

1.4.1 Interrupts and polling

The state of continuous monitoring is known as polling. The processor keeps checking the status of some devices; and while doing so, it does no other operation and consumes all its processing time for monitoring. This problem can be solved by using interrupts [3]; in interrupts, the processor responds only when the device triggers an interrupt. Therefore, the processor is not required to regularly monitor the status (flags, signals etc.) of interfaced and inbuilt devices.

1.4.2 Types of interrupts

Interrupts are generally classified into three types:

1. **Hardware Interrupts** are generated by hardware devices to signal that they need some attention from the OS. They may have just received some data (e.g., keystrokes on the keyboard or a data on the ethernet card); or they have just completed a task which the operating system previous requested, such as transferring data between the hard drive and memory.
2. **Traps or exceptions** are generated by the CPU itself to indicate that some error or condition occurred for which assistance from the operating system is needed.

1.5 Scheduling Algorithms

When a computer is multi-tasking, it frequently has multiple processes or threads competing for the CPU at the same time. This situation occurs whenever two or more of them are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the scheduler, and the algorithm it uses is called the scheduling algorithm [4]. The unit of scheduling is usually the threads, the scheduler chooses which thread to be executed next regardless of which process it belongs to. An optimum scheduling algorithm should minimize the average time that thread take to finish their job, while reducing the response time and keeping the CPU busy as much as possible.

1.5.1 First-Come, First-Served scheduler

It is scheduling algorithm that uses queuing system to schedule threads. With this algorithm, processes are assigned the CPU in the order they request it; therefore, there is one queue for ready processes and the scheduler chooses one to be executed until it finishes and moves to the second earliest thread. This algorithm is non-preemptive; meaning that the scheduler allows a thread to be executed until it finishes, without interruption even if it is blocked waiting for IO, or there is another higher priority thread is waiting in the queue. This algorithm has a lot of flaws; First, the IO bounded threads will spend most of their time blocking waiting for IO, while preventing other threads from using the CPU. This results, higher waiting time and poor efficient use of the CPU.

1.5.2 Priority based scheduler

Another non-preemptive algorithm in which each thread has a priority, the scheduler chooses the higher priority thread to be executed next. This algorithm suffers from the same flaw as the

previous one, and it will bit put the CPU in a good use if an IO bounded thread has high priority. Another flaw is starvation, if more high priority threads are created, the lower priority one may not be executed ever.

1.5.3 Round Robin scheduler

A fair preemptive scheduling algorithm that allows each a thread to be executed for a fixed amount of time called *time splice*, and if a thread is blocked waiting for an IO, the scheduler will schedule another thread. This scheduling algorithm fixes the flaws of both previous algorithms, but it does not have a priority system, so a kernel thread that should be executed as soon as possible will wait its turn like any other thread.

1.5.4 Multi-level Queueing scheduler

This algorithm makes use of the previous two algorithms; it has multiple queues with different priorities, the threads in the same are scheduled using Round Robin. The scheduler does not process to schedule certain threads in a queue unless all the higher priority queues are empty. Threads can move to higher priority queues if they block and wait for IO operations, and does not use much CPU; this will minimize the response time and decrease the average waiting time in generally for all threads.

1.6 Modes of execution

Modern operating systems generally have two modes, *kernel mode* and *user mode*. The kernel mode has full unrestricted access to the hardware, privileged instructions, physical memory and virtual memory of all processes. While the user mode has a restricted access to some non-privileged level instructions and the virtual memory of the current process. Only few trusted programs must run in the kernel mode, including most of the operating system functions, because any faults may corrupt the memory, misconfigure the attached hardware, or even crash the whole system. However, in the user mode, any fault may cause damage only to the faulted process, and the system may terminate it at worst case.

When a process running the user mode wants to interact with hardware or any privileged operations, it will execute a system call through a set of APIs provided by the operating system. The system call will switch the execution flow from the user mode to a piece of program in

operating system running in the kernel mode. Figure 1-10 illustrates the interaction between the user mode and the kernel mode.

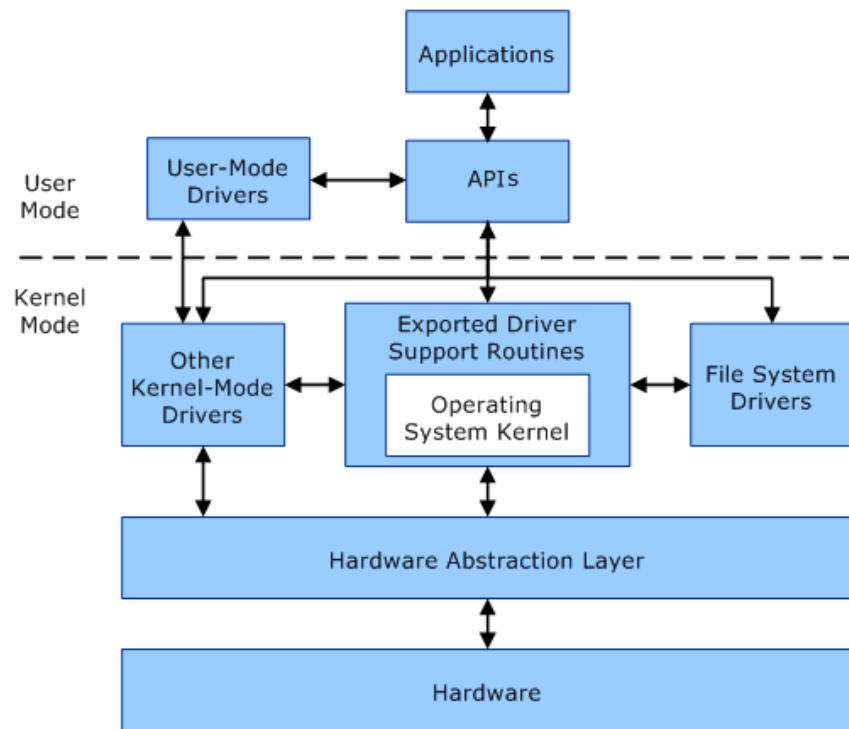


Figure 1-10: Kernel mode and User mode

1.7 x86 instruction set and IA-32 processors

x86 is a family of instruction set architectures initially developed by Intel based on the Intel 8086 microprocessor and its 8088 variants. The 8086 was introduced in 1978 as a fully 16-bit extension of Intel's 8-bit 8080 microprocessor, with memory segmentation as a solution for addressing more memory than can be covered by a plain 16-bit address. Many additions and extensions have been added to the x86 instruction set over the years, almost consistently with full backward compatibility. And instruction set later was extended to be a CISC design. It has Byte-addressing enabled and words are stored in memory with little-endian byte order. Memory access to unaligned addresses is allowed for all valid word sizes. The largest native size for integer arithmetic and memory addresses (or offsets) is 16, 32 or 64 bits depending on architecture generation. [5]

IA-32 (short for Intel Architecture, 32-bit) is the 32-bit version of the x86 instruction set family, designed by Intel and first implemented in the 80386 microprocessor in 1985 and currently used in all 32bit Intel processors. IA-32 is the first incarnation of x86 that supports 32-bit computing; as a result, the "IA-32" term may be used as a metonym to refer to all x86 versions that support 32-bit computing [6]. In some other contexts, certain iterations of the IA-32 ISA are sometimes labelled i486, i586 and i686, referring to the instruction supersets offered by the 80486, the P5 and the P6 microarchitectures respectively. The newer processors that work under IA-64 like Intel i3, i5 ,i7, i9 do emulate the older architecture like IA-32 and IA-16, which enables them to run the any 16bit or 32bit operating system. Figure 1-11 shows a list of the 32, 16, and 8 bit registers in IA-32.

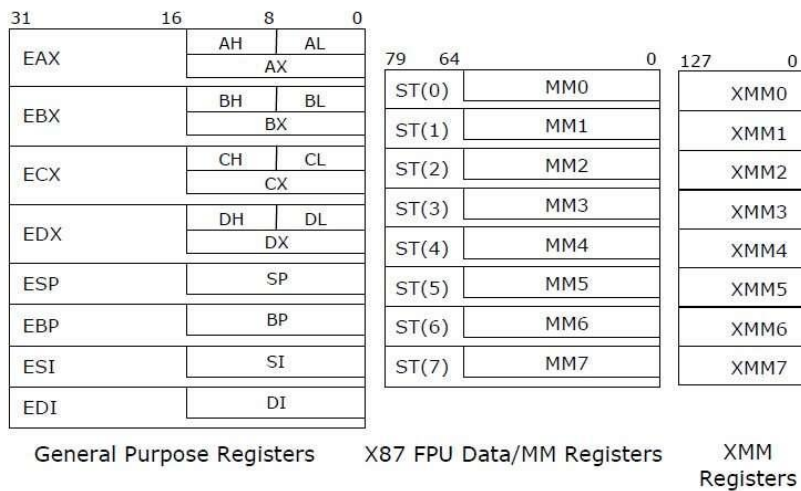


Figure 1-11: IA-32 Registers

Besides the 32bit registers, the IA-32 added few features to the x86 instruction set. Firstly, more addressing modes; all general-purpose registers can be used as base register, while all general-purpose registers except ESP can be used as an index register and can be multiplied by 1, 2, 4, or 8 before being added to the base register value and displacement. This allowed instructions like “ *MOV ECX, [EAX+EBX*4]* ”. Secondly, it extended the address space to 48-bit using segmentation, combining 16-bit segment number and a 32-bit offset within the segment. Thirdly, it supports virtual memory with different protection using paging with two level tables, which will be discussed in more details in the next chapter.

Chapter 2: Design and Implementation

2.1 Setting up IA-32 Protected Mode Features

In order to use the full capabilities of IA-32 processors, the programmer has to setup few features such as memory segments, paging, interrupt and exceptions. These features were probably initialized by the boot loader to execute the first few instructions of the operating system; however, the programmer needs to modify the settings according to operating system needs. Figure 2-1 provides a summary of system registers and data structures that applies to 32-bit modes.

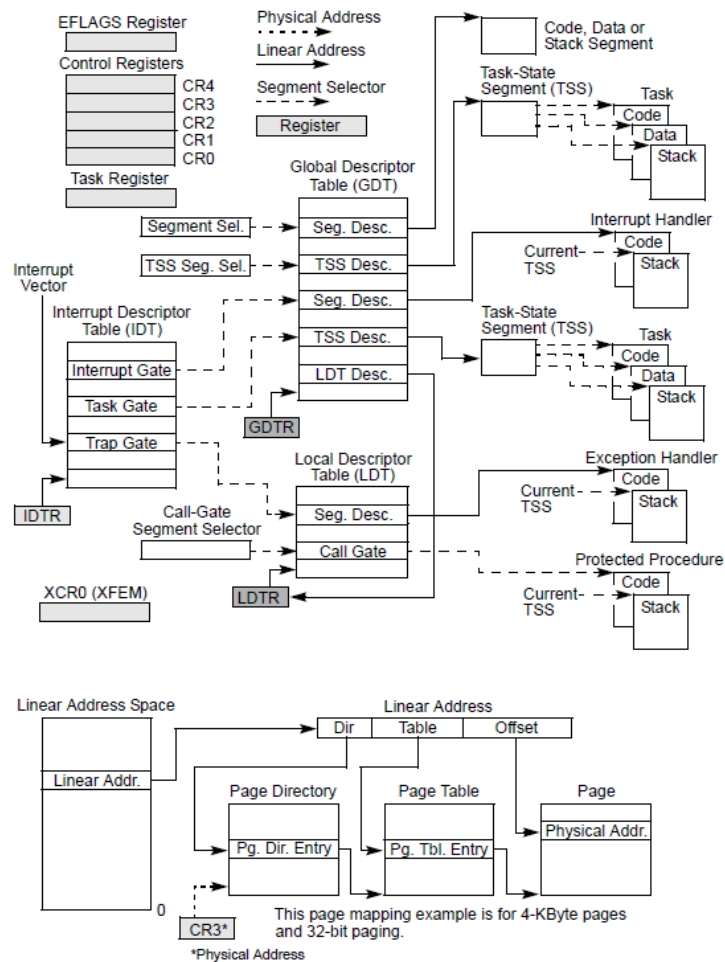


Figure 2-1: IA-32 System-Level Registers and Data Structures
Adapted from [7].

2.1.1 Segmentation

Segmentation is a memory management technique used to divide the virtual memory into multiple regions; an identifier and an offset in that segment are used to reference a specific address. Originally, the segmentation was used to access the different parts of the program like code, read-only data, and writable data, because of the limited memory address bus size in the old Intel processors. However, in modern processors, the 32bit or 64bit bus size is more than enough for most applications, and the need of segmentation has vanished, though IA-32 processor still supports this feature as backward compatibility and enforces it to be able to enter the protected mode. Thus, most modern operating systems that support IA-32 will setup all memory segments to be identical and cover the whole memory.

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT) as shown in Figure 2-1. These tables contain entries called segment descriptors. Segment descriptors provide the base address of segments as well as access rights, type, and usage information.

Each segment descriptor has an associated segment selector. A segment selector provides the software that uses it with an index into the GDT or LDT (the offset of its associated segment descriptor), a global/local flag (determines whether the selector points to the GDT or the LDT), and access rights information.

To access a byte in a segment, a segment selector and an offset must be supplied. The segment selector provides access to the segment descriptor for the segment (in the GDT or LDT). From the segment descriptor, the processor obtains the base address of the segment in the linear address space. The offset then provides the location of the byte relative to the base address. This mechanism can be used to access any valid code, data, or stack segment, provided that the segment is accessible from the current privilege level (CPL) at which the processor is operating. The CPL is defined as the protection level of the currently executing code segment.

The solid arrows in Figure 1-1 indicate a linear address, dashed lines indicate a segment selector, and the dotted arrows indicate a physical address. For simplicity, many of the segment selectors are shown as direct pointers to a segment. However, the actual path from a segment selector to its associated segment is always through a GDT or LDT [8].

The linear address of the base of the GDT is contained in the GDT register (GDTR); the linear address of the LDT is contained in the LDT register (LDTR), and the instruction *lldt r/m16* is used for that.

The mode where all segments cover the whole memory is called *flat mode*, in which the operating system and application programs have access to a continuous, unsegmented address space. To the greatest extent possible, this basic flat model hides the segmentation mechanism of the architecture from both the system designer and the application programmer.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created; one for referencing a code segment and one for referencing a data segment (see Figure 2-2). Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes [8].

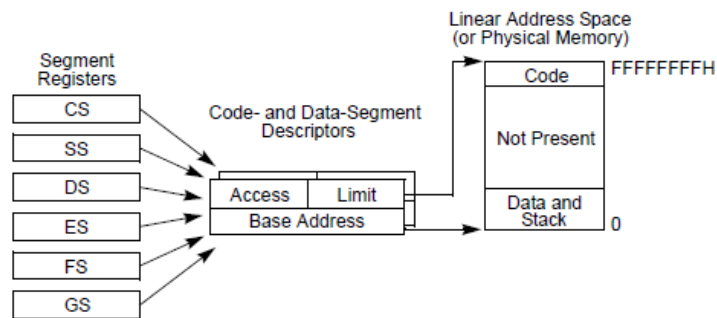


Figure 2-2: Flat Model
Adapted from [9].

2.1.1.1 Segment Selectors and Privilege Levels

A segment selector is a 16-bit identifier for a segment (see Figure 2-3). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following fields.

Index: Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

TI (table indicator) flag: Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

RPL (Requested Privilege Level): Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level. See section 2.4 for a description of how this field is used to switch between execution modes.

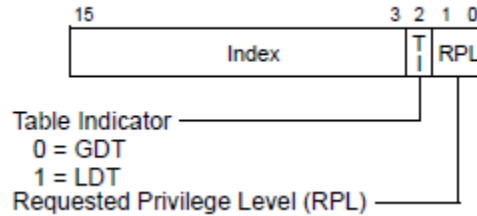


Figure 2-3: Segment Selector
Adapted from [10].

The segment selectors in IA-32 are: *cs*, *ds*, *es*, *ss*, *fs* and *gs*, the first four usually have the same index value in all processes and threads, but RPL field will differ depending on the privilege level that the current task is on. The use of selectors *ss* and *fs* very varies between operating systems, though the majority will use them for storing information related to the current process and thread.

2.1.2 Interrupts

In IA-32, *Interrupt Descriptor Table* (IDT) is the responsible of handling hardware interrupts, software interrupts and internal exceptions, and to aid the handling of exceptions and interrupts, each architecturally defined exception and each interrupt condition requiring special handling by the processor is assigned to unique identification number, called a vector number. The processor uses the vector number assigned to an exception or interrupt as an index into the entries of interrupt descriptor table (IDT). See Table 2-1 for the list of indices and their corresponding interrupts.

The allowable range for vector numbers is 0 to 255. Vector numbers in the range 0 through 31 are reserved by the Intel 64 and IA-32 architectures for architecture-defined exceptions and interrupts. While, vector numbers in the range 32 to 255 are designated as user-defined interrupts and are not reserved by the Intel 64 and IA-32 architecture. These interrupts are generally assigned to external I/O devices to enable those devices to send interrupts to the processor through one of the external hardware interrupt mechanisms.

Note that the difference between an interrupt and exception in the current context is that an interrupt is triggered by an external hardware, whereas an exception is a fault in the program or the processor configuration. All interrupts and exception can be manually triggered by the instruction *int n*, *n* is the interrupt number.

Similarly to GDT, IA-32 has a special instruction *lidt /m16* which is used to load the address of IDT into IDTR register.

2.1.2.1 Interrupt handlers

After an interrupt is triggered, the processor tries to translate the handler address from both IDT and GDT (see Figure 2-4). If it encounters any misconfigured entry in both tables, the processor will trigger double fault exception, if another fault is encountered while translating the double fault exception, a triple fault exception is triggered which will cause a system reset.

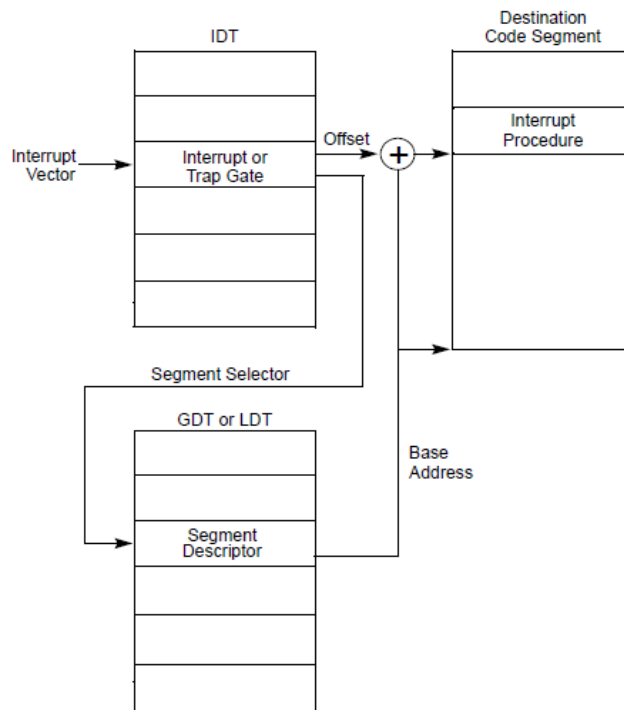


Figure 2-4: Interrupt Procedure Call

Adapted from [11].

During an interrupt handler call, if there is no privilege level change between the interrupt handler and the original code e.g., code is running in ring0 (kernel mode) and an interrupt is triggered to be executed in same privilege level, the processor pushes to the stack some

information about interrupt and how to return to the original interrupted code. However, when there is privilege level change e.g., code is running in ring3 (user mode) and an interrupt is triggered to be executed in ring0 (kernel mode), the processor uses a new stack memory for the handler to store the interrupt information.

Interrupt index	Description
0x00	Division by zero
0x01	Single-step interrupt
0x02	Non Maskable Interrupt
0x03	Breakpoint
0x04	Overflow
0x05	Bound Range Exceeded
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double Fault
0x09	Coprocessor Segment Overrun (<i>386 or earlier only</i>)
0x0A	Invalid Task State Segment
0x0B	Segment not present
0x0C	Stack Segment Fault
0x0D	General Protection Fault
0x0E	Page Fault
0x0F	<i>Reserved</i>
0x10	x87 Floating Point Exception
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception
0x14	Virtualization Exception
0x15	Control Protection Exception

Table 2-1: Protected-Mode Exceptions and Interrupts

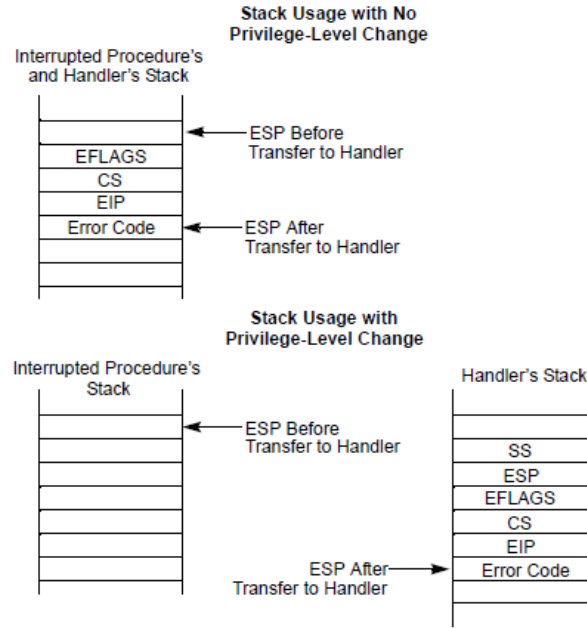


Figure 2-5: Stack Usage on Transfers to Interrupt and Exception-Handling Routines
Adapted from [12].

2.1.3 Virtual Memory

The memory management facilities of the IA-32 are divided into two parts: *segmentation* and *paging*. Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another, however in flat mode as discussed before, the processor makes segmentation transparent. Paging on the other hand, provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks. When operating in protected mode, some form of segmentation must be used. There is no mode bit to disable segmentation. The use of paging, however, is optional. The paging mechanism can be configured to support simple single-program systems, multitasking systems, or multiple-processor systems that uses shared memory. When paging is enabled, all the addresses in the system (including the addresses in IDT and GDT) will be considered as virtual addresses, and need to be translated into actual physical addresses by the *memory management unit* (MMU), as shown in Figure 2-6.

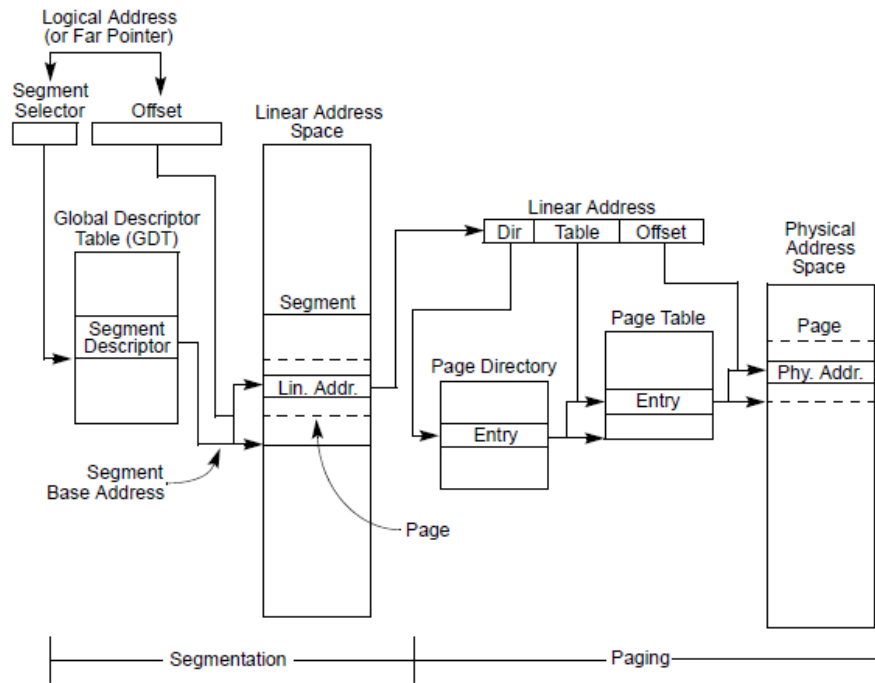


Figure 2-6: Segmentation and Paging
Adapted from [13].

2.1.3.1 Paging Setup

The MMU translates addresses through a series of two tables, *page directory* (PD), and *page table* (PT), both contain 1024 entries of 4 bytes. Each *page directory entry* (PDE) points to a page table, while each *page table entry* (PTE) points to a physical memory page; a memory frame, which is typically 4kb in size. This gives up to 4GB virtual address space accessible to the processor. Figure 2-7 illustrates the translation process of a 32-bit address into a memory frame, the most significant ten bits of any virtual address represent an index in the page directory, whereas the next ten bits represents an index in the corresponding page table, the last twelve bits are an offset in the 4kb memory frame.

Besides the pointers to PT or frame, PDE and PTE contain some important information about the pages they point to. Figure 2-8, Table 2-2, and Table 2-3 give details of each field.

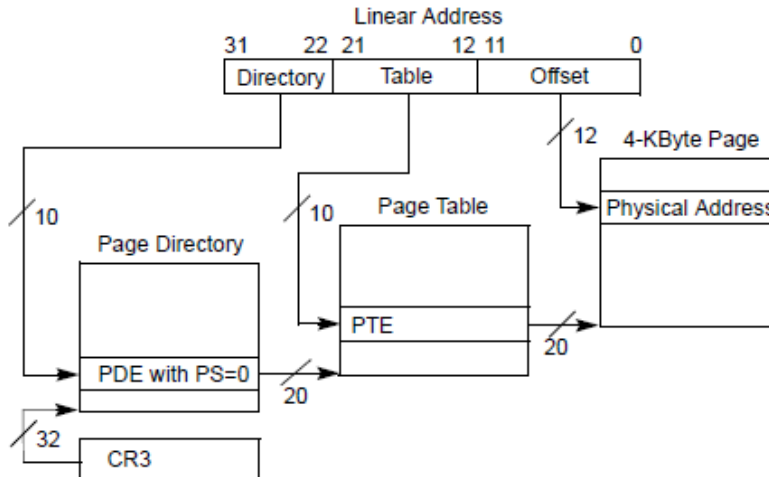


Figure 2-7: Virtual Address Translation to a 4-KByte Page using 32-Bit Paging
Adapted from [14].

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory ¹										Ignored							P	P	Ignored			CR3										
Address of page table										Ignored			<u>0</u>	I	A	P	P	U	R	<u>1</u>	PDE: page table											
Ignored																	<u>0</u>				PDE: not present											
Address of 4KB page frame										Ignored			G	P	A	D	A	P	P	U	R	<u>1</u>	PTE: 4KB page									
Ignored																	<u>0</u>				PTE: not present											

Figure 2-8: Paging-Structure Entries with 32-Bit Paging
Adapted from [15].

2.1.3.2 Identity Mapping

Enabling the paging in IA-32 architecture can be tricky. At first all addresses are physical address including the address of the current instruction in *EIP* and the address of the stack in *ESP*, and at the moment of enabling paging, all these addresses will be invalid in virtual memory. A simple solution to this problem is *identity mapping*, which is mapping physical addresses of some pages to the same virtual addresses so the switching code does not encounter invalid addresses after enabling paging.

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page); otherwise, ignored
11:8	Ignored
31:12	Physical address of 4-KByte aligned page table referenced by this entry

Table 2-2: Format of a 32-Bit Page-Directory Entry that References a Page Table
Adapted from [16].

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry; otherwise, reserved (must be 0) ¹
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global; ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

Table 2-3: Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page
Adapted from [16].

The identity mapping has to cover only few pages of the operating system that are responsible of enabling paging, after that it is possible to jump into a page where the it is not identity mapped. Figure 2-9 illustrates the identity mapping.

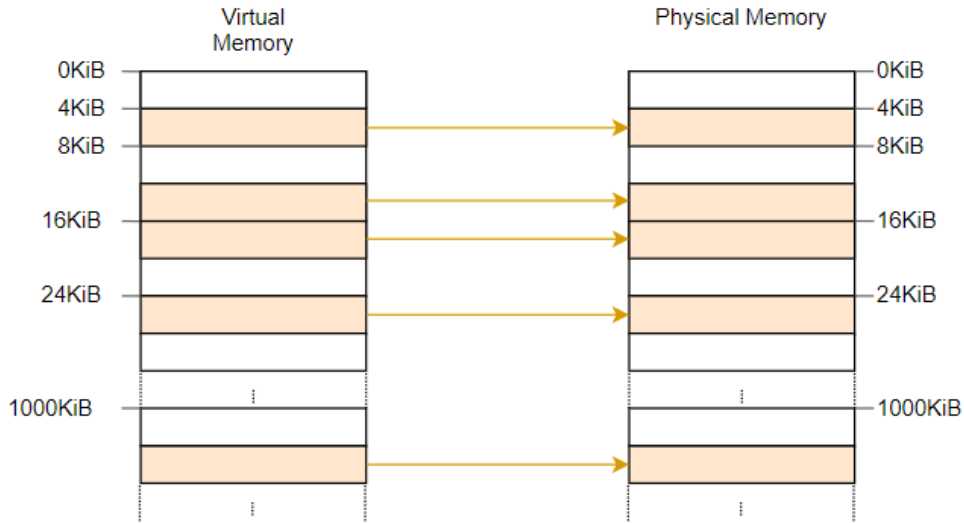


Figure 2-9: Identity mapping

2.1.3.3 Memory Page Allocator

Each page table is 4kb in size, with 1024 one of it, they will take roughly 4Mb to map all 4Gb space of virtual memory, and since each process has its own virtual space, hence page tables, huge chunks of memory will be used for the sole reason of mapping. A more reliable approach is to only map a page when it is needed, so at first only one page directory and page table exist, and they grow by the need of more memory.

A memory allocator is a convenient technique to allocate 4Kb pages whenever the kernel (or user applications) needs. It works by keeping track of all used and free physical and virtual pages, *CyanOS* uses a simple bitmap which is initialized to 0 at first, and each allocated page its bit will be set to one. A small optimization is implemented by a variable that keeps the index of the last allocated page, the next time a page is required, the allocator start searching starting from that variable.

2.1.3.4 Page Faults and Memory Swapping

A page fault is an exception generated by the processor when the program tries to access a page that does not exist, has higher privilege level, or to write to read-only page. One use of the page faults is to detect an access to certain page specially in memory swapping; when a page is swapped out of memory, the kernel sets its present bit in PTE to zero (see Table 2-3), this way the page is not present and will trigger a page fault whenever a program tries to access it, the

exception handler will decide then whether to put the process to sleep or swaps in the required page from the disk. Figure 2-10 illustrates the process of swapping.

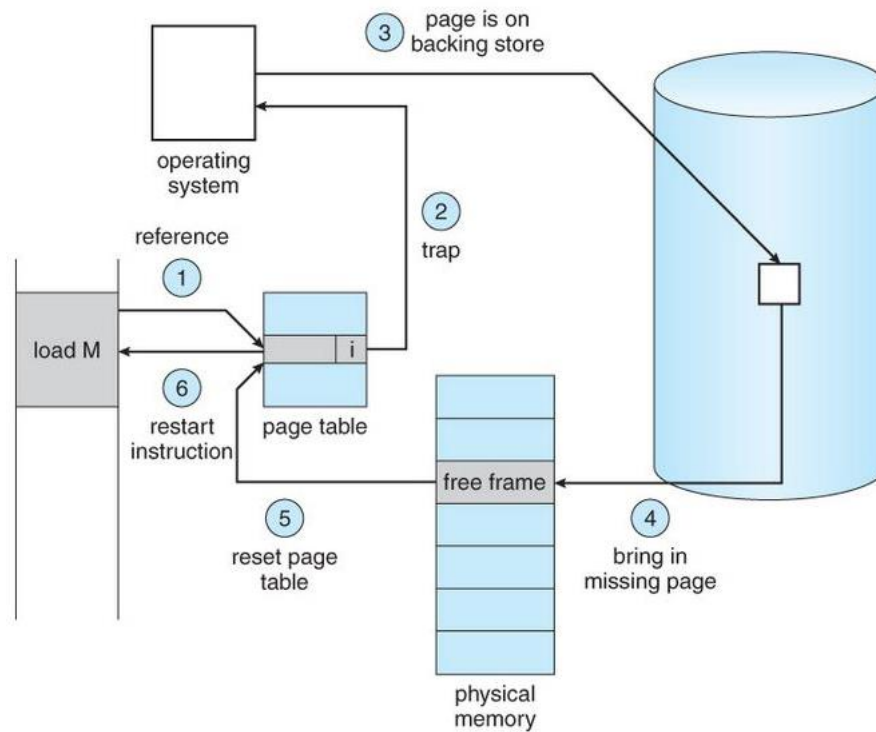


Figure 2-10: Using page fault in swapping

2.2 Device's Drivers

2.2.1 Intel Programmable Interrupt Controller (PIC 8259)

The Intel 8259 is a Programmable Interrupt Controller (PIC) designed for the Intel 8085 and Intel 8086 microprocessors to manage hardware interrupts and send them to the appropriate system interrupt. The 8259 combines multiple interrupt input sources into a single interrupt output to the host microprocessor, extending the interrupt levels available in a system beyond the one or two levels found on the processor chip. The 8259A was the interrupt controller for the ISA bus in the original IBM PC and IBM PC AT. The 8259 has coexisted with the Intel APIC Architecture since its introduction in Symmetric Multi-Processor PCs. Modern PCs have begun to phase out the 8259A in favor of the Intel APIC Architecture. However, while not anymore a

separate chip, the 8259A interface is still provided by the Platform Controller Hub or Southbridge chipset on modern x86 motherboards [17]. Figure 2-11 shows how the devices are connected to CPU through PIC.

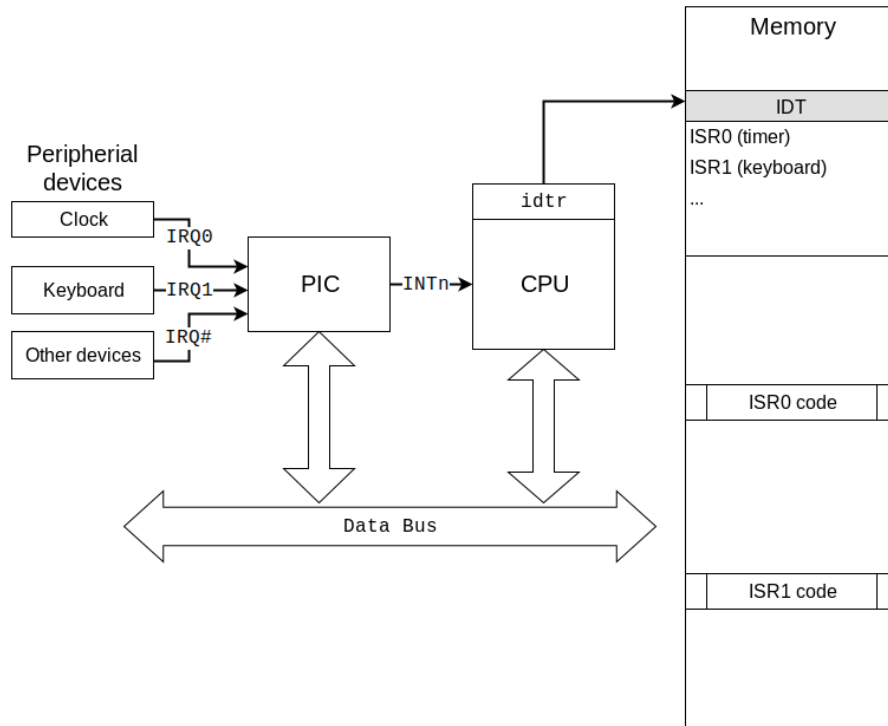


Figure 2-11: PIC 8259

To start receiving hardware interrupts from the PIC, it must be configured first using the following steps:

- 1- Remap the interrupts. The PIC uses interrupts 0 - 15 for hardware interrupts by default, which conflicts with the CPU interrupts. Therefore, the PIC interrupts must be remapped to another interval.
- 2- Select which interrupts to receive. You probably do not want to receive interrupts from all devices since you do not have code that handles these interrupts anyway.
- 3- Set up the correct mode for the PIC.

Since a PIC is capable of handling just 8 devices, the need of more devices made motherboard designers to cascade two PICs so it is possible to handle 15 devices. Figure 2-12 shows how two PICs are cascaded, while Table 2-4 lists the different devices that are connected to PIC.

Refer to PIC 8259 manual to find the different control words to initialize and configure the different devices on PIC.

Interrupt Number	Device
0	CMOS real-time clock
1	Free for peripherals / legacy SCSI / NIC
2	Free for peripherals / SCSI / NIC
3	Free for peripherals / SCSI / NIC
4	PS2 Mouse
5	FPU / Coprocessor / Inter-processor
6	Primary ATA Hard Disk
7	Secondary ATA Hard Disk
8	CMOS real-time clock
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

Table 2-4: Device connected to PIC

2.2.2 PS/2 Keyboard

The PS/2 Keyboard is a device that talks to a PS/2 controller using serial communication. The PS/2 controller has an input port to read the scan codes of the pressed key (scan code is different than ASCII code). It also generates an interrupt at IRQ1 in PIC.

2.2.3 Peripheral Component Interconnect (PCI) bus

Peripheral Component Interconnect (PCI) is a local computer bus for attaching hardware devices in a computer and is part of the PCI Local Bus standard. The PCI bus supports the functions found on a processor bus but in a standardized format that is independent of any given processor's native bus. Devices connected to the PCI bus appear to a bus master to be connected directly to its own bus and are assigned addresses in the processor's address space. It is a parallel

bus, synchronous to a single bus clock. Attached devices can take either the form of an integrated circuit fitted onto the motherboard (called a planar device in the PCI specification) or an expansion card that fits into a slot. [18]

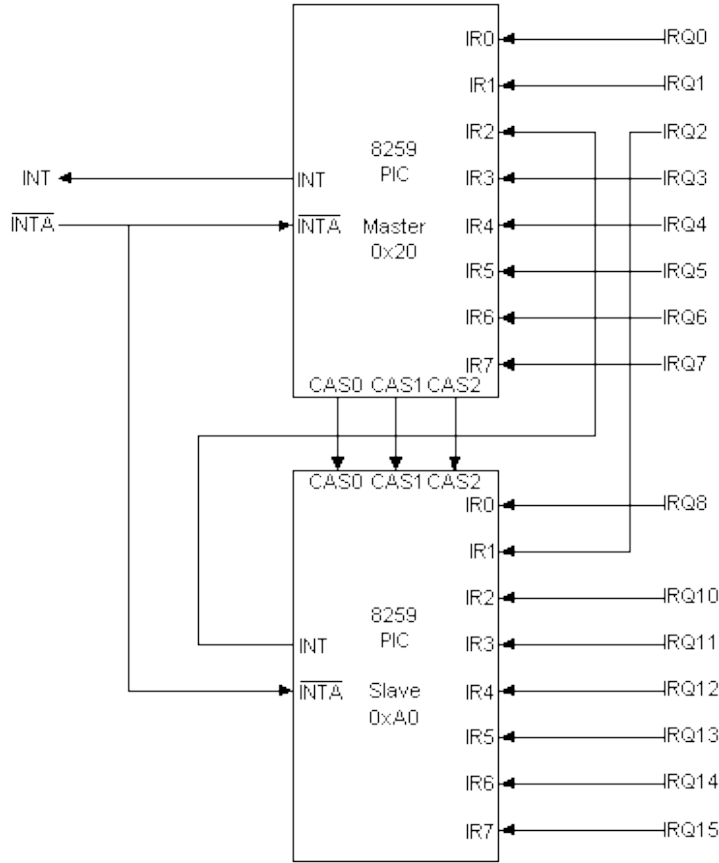


Figure 2-12: Cascading two PICs

The PCI specification provides for totally software driven initialization and configuration of each device. Therefore, each one provides a 256-byte configuration registers. These registers can provide information about the device such as device id, vendor id and device class, and some information about its functionality such as the interrupt line or the base address of device's IO (either memory mapped or ports). Refer to [19] for a detailed explanation on each field of the configuration space.

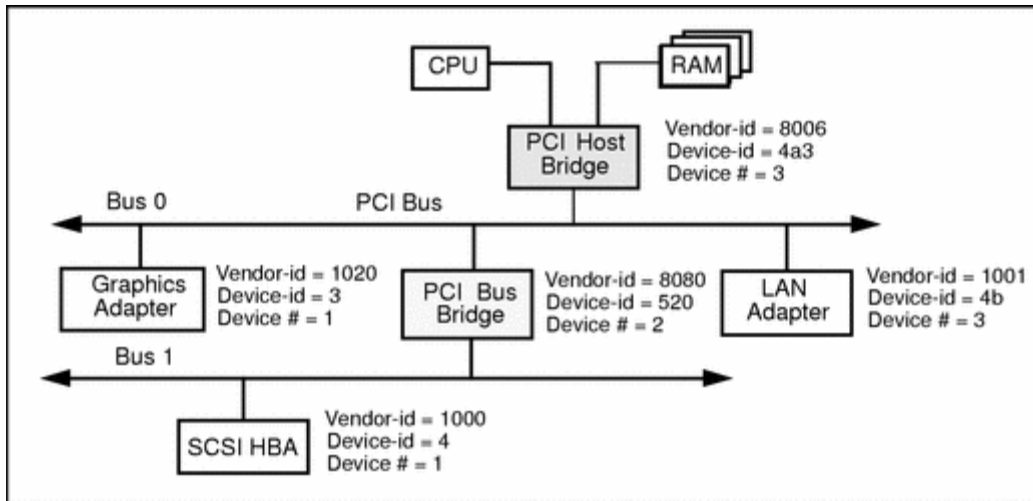


Figure 2-13: PCI structure

PCI has 256 buses, each bus has 32 slots, and each slot has 8 functions. Obviously not all these buses, slots, and functions will be filled with a physical hardware. Therefore, one way to enumerate the PCI devices in the system is by scanning only the first 8 buses, and scan the slots in each bus, then scanning the function in each slot. If a function has the type of *PCIBridge*, its *secondary bus number* will point to a bus that has more slots and functions that will be scanned recursively too. Figure 2-13 shows the structure of PCI devices, whereas Figure 2-14 shows a code snippet on how to enumerate these devices.

2.2.4 RTL8139 Ethernet Network Device

Realtek RTL8139 has the simplest interface among NICs (Network interface controller) devices and it is emulated by a software like *Qemu* or *Bochs*, for these reasons it was the chosen driver to implement networking in *CyanOS*.

Since RTL8139 is connected to PCI, it needs to be detected by the previously discussed method (The PCI vendor ID is 0x10EC and the device ID is 0x8139), and the configuration space will provide the command addresses, MAC address, interrupt line of this device. RTL8139 is configured to write the received data to a specific address of buffer in the memory, and then it will trigger an interrupt. For transmitting packets, The RTL8139 NIC uses a round robin style for transmitting packets. It has four transmit buffer registers to hold the address of the buffer containing the data of the packet to be sent. It should be noted, that all addresses provided to

RTL8139 must be physical addresses instead of virtual, so *virtual_to_physical_address* is used to convert. Refer to RTL8139 for detailed information about the process.

```
void PCI::scan_pci(Function<void(PCIDevice&)> callback){
    for (size_t bus = 0; bus < 8; bus++) {
        scan_bus(callback, bus);
    }
}
void PCI::scan_bus(Function<void(PCIDevice&)& callback, u8 bus){
    for (size_t slot = 0; slot < 32; slot++) {
        scan_slot(callback, bus, slot);
    }
}
void PCI::scan_slot(Function<void(PCIDevice&)& callback, u8 bus, u8 slot){
    PCIDevice slot_device{bus, slot, 0};
    if (slot_device.does_exist()) {
        scan_function(callback, bus, slot, 0);
        if (slot_device.has_multiple_functions()) {
            for (size_t function = 1; function < 8; function++) {
                scan_function(callback, bus, slot, function);
            }
        }
    }
}
void PCI::scan_function(Function<void(PCIDevice&)& callback, u8 bus, u8 slot, u8 function){
    PCIDevice function_device{bus, slot, function};
    if (function_device.does_exist()) {
        callback(function_device);
        if (PCIDevice{bus, slot, function}.header_type() == PCIDevice::HeaderType::PCIBridge) {
            scan_bus(callback, PCIBridge{bus, slot, function}.secondary_bus_number());
        }
    }
}
}
```

Figure 2-14: Enumerating all PCI devices

2.3 Multitasking

After discussing how the kernel initializes the required features in IA-32, this part describes a higher level overview of the different components of *CyanOS* to achieve multitasking, that includes processes, threads, context switch, synchronization, interprocess communication and the user space.

2.3.1 Kernel memory space

As many modern operating systems, *CyanOS* uses the *higher half kernel* model; meaning all processes share the same kernel in their virtual space which is from 0xC0000000 to 0xFFFFFFFF, and independent in the rest of the virtual space. This implies that the values of PDEs in page directory from 768 to 1023 are identical in all page directories of all processes in the system, which all marked as *supervisor* (refer to Table 2-3) to disallow any access to these pages from the user space. Figure 2-15 shows the mapping of virtual spaces of two processes to the physical memory.

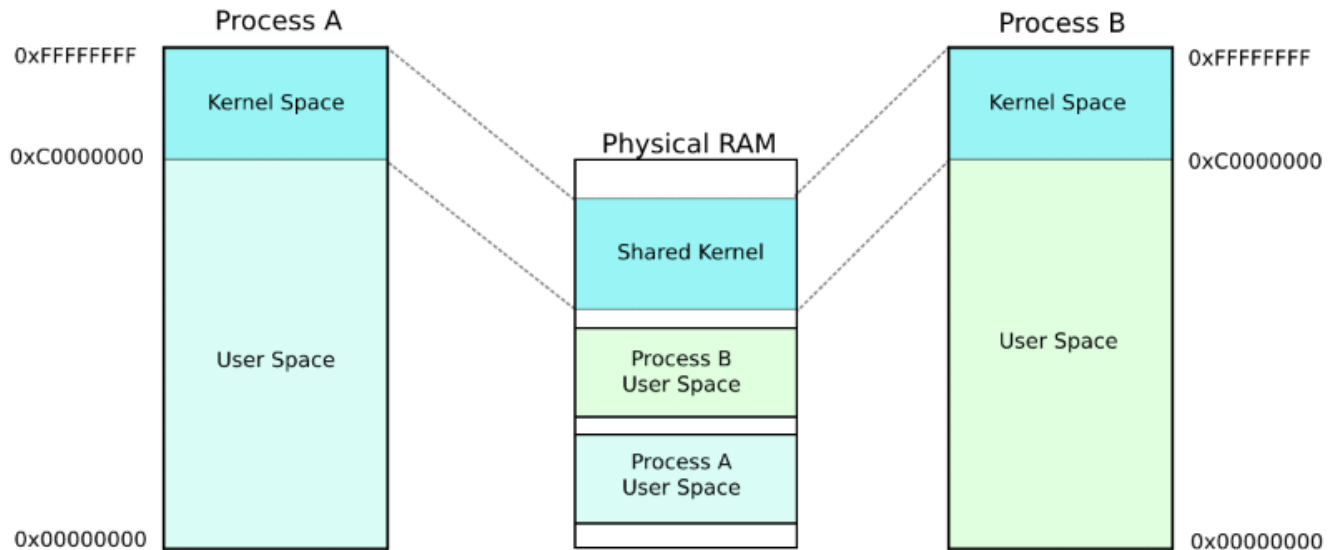


Figure 2-15: Higher Half Kernel model

2.3.2 Processes and Threads

As any other modern kernel, *CyanOS* has processes and threads; process is a running program loaded from the file system that has its own virtual address space (page directory and tables) and

set of resource like handles (see section 2.6.3 for more details), it also contains set of threads. A thread on the other hand has its own stack, execution flow and may share some data on the heap.

Table 2-5 and Table 2-6 Illustrate the different internal information fields stored in the kernel about processes and threads in *CyanOS*.

Field	Description
id	A unique identifier to the process.
name	Process's name
path	The location of the loaded program on the file system.
privilege_level	The privilege level of the process which can be either <i>Kernel</i> or <i>User</i> .
parent	A reference to the parent process.
state	The state of the process which can be either <i>Ready</i> , <i>Blocked</i> , <i>Suspended</i> or <i>Zombie</i> .
handles_list	The list of the handles in the process (check 2.6.3 for more details).
threads_list	The list of the threads in the process.
page_directory	The physical address of the page directory of the process's virtual space.

Table 2-5: Internal fields of a Process.

Field	Description
id	A unique identifier to the thread.
privilege_level	The privilege level of the thread what can be either <i>Kernel</i> or <i>User</i> .
parent_process	A reference to the processes holding this thread.
entry_point	The address of the first instruction to be executed by the thread.
stack_pointer	The address of the stack for the thread.
state	The state of the process which can be either <i>Ready</i> , <i>BlockedSleep</i> , <i>BlockedQueue</i> , <i>BlockedQueueTimed</i> or <i>Suspended</i> .
blocker_waitqueue	If the process is blocked, this field contains a reference to the blocking waitqueue. (See section 2.3.3.2 for details)

Table 2-6: Internal fields of a Thread.

2.3.2.1 Life Time of a Process

When the kernel takes control from the bootloader, it initializes some internal data structures and devices then it creates the first process in the system which called “Adam” and has the id of zero. This process is a special since it has no parent and no actual associated program in the file system; it is also responsible of spawning some programs like the shell or a desktop GUI manger.

Creating a new process is done by *CreateProcess* system call, the kernel initiates the internal information about this process and creates a page directory for it. Next, an ELF file is loaded from the file system to the executable loader (See section 2.4.2 for details), if the file is invalid it will clean the allocated data and *CreateProcess* will return an error. Finally, the kernel creates a new thread that starts from the entry point of the executable file, the path of the current process and its arguments will be passed to the main function.

When *CreateProcess* system call succeeds it returns a *Handle*, which is a wrapper for a *ProcessDescription*. *ProcessDescription* is kernel’s object that references a process, which allows the kernel to do operations on this process including (wait, suspend, terminate). Similarly, A foreign process can obtain a *Handle* for another process using *OpenProcess* system call. Nevertheless, All *Handle*-s of the process must be closed by *CloseHandle* after use, otherwise the kernel assumes that the process resources are still needed.

A program can use *SuspendProcess* to suspend all the threads in a process, which can be resumed later by *ResumeProcess*. Moreover, *TerrminateProcess* is used to close a certain process and release all its resources as well as its threads’ if there is no *ProcessDescription* referencing it. However, if one or more *ProcessDescription* is still not closed, the kernel marks this process’s state as *Zombie* and waits until all the *ProcessDescription* to be closed so it releases the resources. Furthermore, *WaitSignal* system call is used to wait until a process is terminated and returns the error code that process returned.

2.3.2.2 Life Time of a Thread

Similarly to a process, a thread has *CreateThread*, *OpenThread*, *SuspendThread*, and *TerminateThread* which work exactly like the operations on a process. Additionally, the kernel provides *Sleep* and *Yield* system calls, *Sleep* will suspend the execution flow of a thread for a

certain duration, while Yield allows the thread to voluntarily give up its time slot of the scheduler.

2.3.2.3 Context Switch

In *CyanOS*, the unit of scheduling is threads; the kernel chooses which thread to be scheduled regardless of which process it belongs to. The scheduler divides all threads in the system into five lists: *ready*, *sleeping*, *blocking*, *timed_blocking* and *suspended*. *Ready* list contains any thread that is ready to run and its time slot has expired, *sleeping* list contains threads that executed *Sleep* system call and waiting for a specific time duration to pass, *blocking* list on the other hand are threads which are blocked waiting in a waitqueue (see section 2.3.3.2 for more details about waitqueue), and finally *timed_blocking* is a list of threads blocked in a waitqueue but have a time out. Figure 2-16 shows the FSM of these transactions.

The kernel starts the scheduler by initializing the Intel 8253 Programmable Interval Timer that is responsible of generating interrupts at 1ms intervals; these interrupts will invoke the scheduler to processes the five lists of threads accordingly. Starting with *sleeping* and *timed_blocking* lists, the scheduler enumerates every element of these lists and checks if their waiting time has been elapsed, if so, the thread will be moved to *ready* list. Furthermore, the scheduler chooses a thread from the *ready* list to be executed next, there are plenty of scheduling algorithms as discussed earlier in chapter 1, but for the sake of simplicity *CyanOS* currently uses a simple preemptive Round-Robin algorithm to make debugging the kernel much easier since the next thread to be executed can be easily predicted (this helps much with debugging bugs related to deadlocks and race conditions).

When the scheduler chooses the next thread to be executed, it saves the context (registers including the stack and instruction pointers) of the current executing thread, and preparing to load the context of the next thread after exiting the scheduler's interrupt routine. The scheduler also checks whether the current thread's parent process is different from the next thread's, if so, it loads the page directory of the next process which will switch to the new process's virtual space.

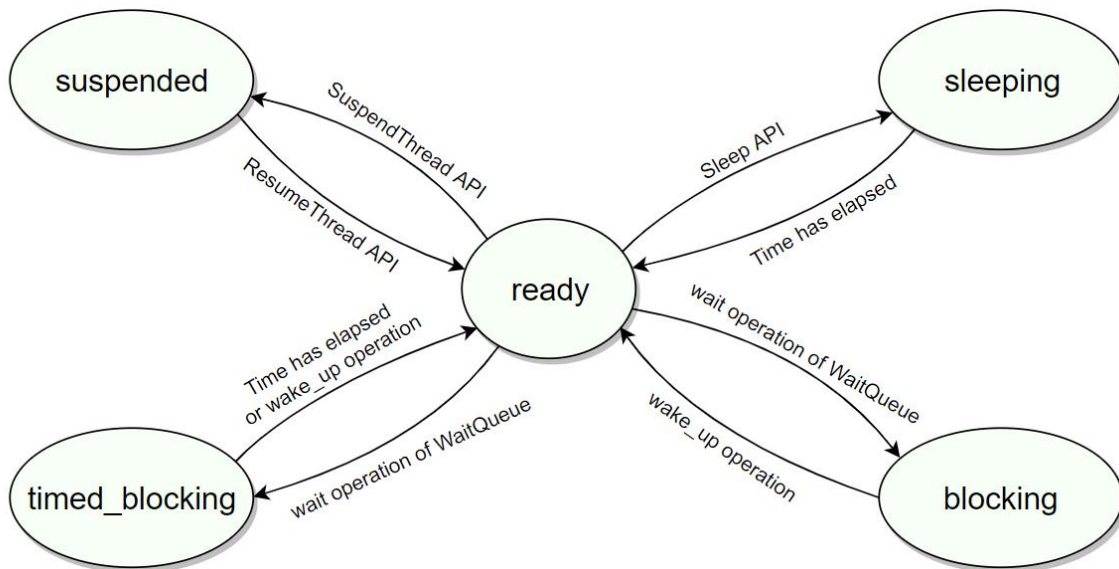


Figure 2-16: Threads movement between scheduler lists

2.3.3 Task Synchronization

Synchronization is an important part of the operating system; it insures mutual exclusion, fairness and non-starvation between threads. It can be done through six primitives: *Spinlock*, *WaitQueue*, *Mutex*, *Semaphore*, *MessagingWaitQueue* and *MultiWaitQueue*.

2.3.3.1 Spinlock

Spinlocks are low-level primitive since they do not rely on the scheduler infrastructure, they work by disabling all interrupts in the current processor core (it is more relevant in multi-core processor), and keep checking a flag until it is set (spinning in a loop). Checking and setting the flag should be an atomic operation, that is why *xchg* instruction is used in IA-32 processors.

Spinlocks can be used anywhere in the kernel including interrupt handlers; however, they should be used wisely, a poor used spinlock can lead to severe performance issues, therefore it is recommended to use spinlocks only where the critical section is very short. It can be noted that kernel components rely heavily on the use of the spinlocks to protect any shared data. Figure 2-17 shows a pseudo code for a spinlock.

2.3.3.2 WaitQueue

WaitQueue is a higher level primitive that allows a thread to be blocked until some condition is met or a certain timeout has elapsed. It has four operations: *wait*, *wait_on_event*, *wake_up* and *wake_up_all*. *wait* operation is used to block the current thread until it is woken by another

thread using *wake* or the timeout has elapsed, while *wait_on_event* is similar to *wake* but a condition is passed to it as a C++ template function (see Appendix A.1), this function operation is blocked until the condition is satisfied after waking up or the timeout is passed. Moreover, *wake_up* and *wake_up_all* are used to wake up a single thread and all blocked thread respectively.

```
void StaticSpinlock::initialize()
{
    m_value = 0;
}
void StaticSpinlock::acquire()
{
    DISABLE_INTERRUPTS();
    while (test_and_set(&m_value) != 0) { // atomic operation
    }
}
void StaticSpinlock::release()
{
    ASSERT(m_value != 0);
    m_value = 0;
    ENABLE_INTERRUPTS();
}
```

Figure 2-17: Spinlock pseudo code

And to describe how these operation work internally... *wait* moves the current thread from *ready* list of the scheduler to *timed_blocking* list if a timeout is provided, if not, the thread is moved to *blocking* list. *wait_on_event* uses *wait* internal in a loop until the condition is met or the timeout is passed, *WaitQueue* also saves a reference to the blocked thread in internal list. *wake_up* and *wake_up_all* enumerate the threads in the internal list and move them from *timed_blocking* or *blocking* lists to *ready* list.

WaitQueue is heavily used in almost all blocking functions in the kernel, especially the virtual file system which will be discussed in VFS Implementation.

2.3.3.3 Mutex and Semaphore

Semaphores are another synchronization primitive which are wrappers around *WaitQueue* with few restrictions. The semaphores are initialized with value to be in the internal counter, acquire operation is used to decrease the counter and block the current thread if the value is less than

zero, using *WaitQueue*'s *wait*, release operation increases the value of the internal counter and wake up the blocking threads if the value is less or equal to zero, using *WaitQueue*'s *wake_up*.

A *Mutex* is a semaphore with an internal counter of one (i.e., binary semaphore).

2.3.3.4 MessagingWaitQueue

MessagingWaitQueue is similar to *WaitQueue*, but its *wait* operation returns any data type passed to it by the other thread in the *wake_up* operation. This primitive utilizes C++'s templates to achieve its purpose which might lead to larger binaries size if it was overused, however it is still an efficient technique to pass data between threads after completion a task, without much code to be written.

2.3.3.5 MultiWaitQueue

Another primitive that is similar to *WaitQueue*, but allows a thread to be waiting in multiple queues and it will be unblocked only after being waken up by all the queues. It is useful when a task needs to start only after few other threads completed their tasks.

2.3.4 Interprocess Communication

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other by sharing data in a synchronous way. *CyanOS* provides IPC using *Pipes* and *Domain Sockets*.

2.3.4.1 Pipes

Pipes allow two processes to communicate in standard producer–consumer fashion: the producer writes to one end of the pipe (the write end) and the consumer reads from the other end (the read end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction [20]. A pipe can be either named; has a name and path in the file system, or can be anonymous. In both cases, creating a pipe or opening an existing one returns a *Handle* that is used to perform a synchronous *read* and *write* operation, both operations will block until a data is available to read, and data has been written, respectively. Figure 2-18 shows shared pipe handles between two processes.

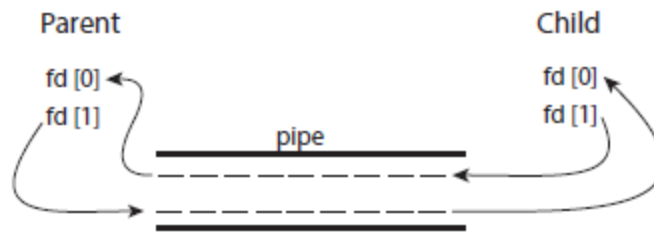


Figure 2-18: Pipe Handles between two processes

Internally, a pipe consists of a *CircularBuffer* (see Appendix B.6) and a *WaitQueue* and they work as a reader-writer problem. *read* operation will try to read any data available in the buffer, if no data is found, it will block until some other thread writes more data. While *write* will try to write data to the buffer, and it will block if it is full, until some other thread reads the data.

2.3.4.2 Domain Socket

Domain Socket on the other hand is bidirectional and can allow multiple *readers* and *writers*. it consists of client that initiates the socket, and a server that accepts it, while socket can be with multiple clients, only one server is allowed. To start a socket connection, the server initiates a socket with a name to be installed in the file system and calls *listen* to mark the socket as passive and can accept incoming connection requests, then it calls *accept* which will block until an incoming connection has come from the client using *connect*, *accept* will return a handle to the newly created connection. After establishing the connection, the server and the clients can use the handles to write and read data from both ends. The flowchart in Figure 2-19 illustrates this process.

The internal design of domain socket is similar to pipes; however, it has two *CircularBuffers* and two *WaitQueues* for both incoming and outgoing data.

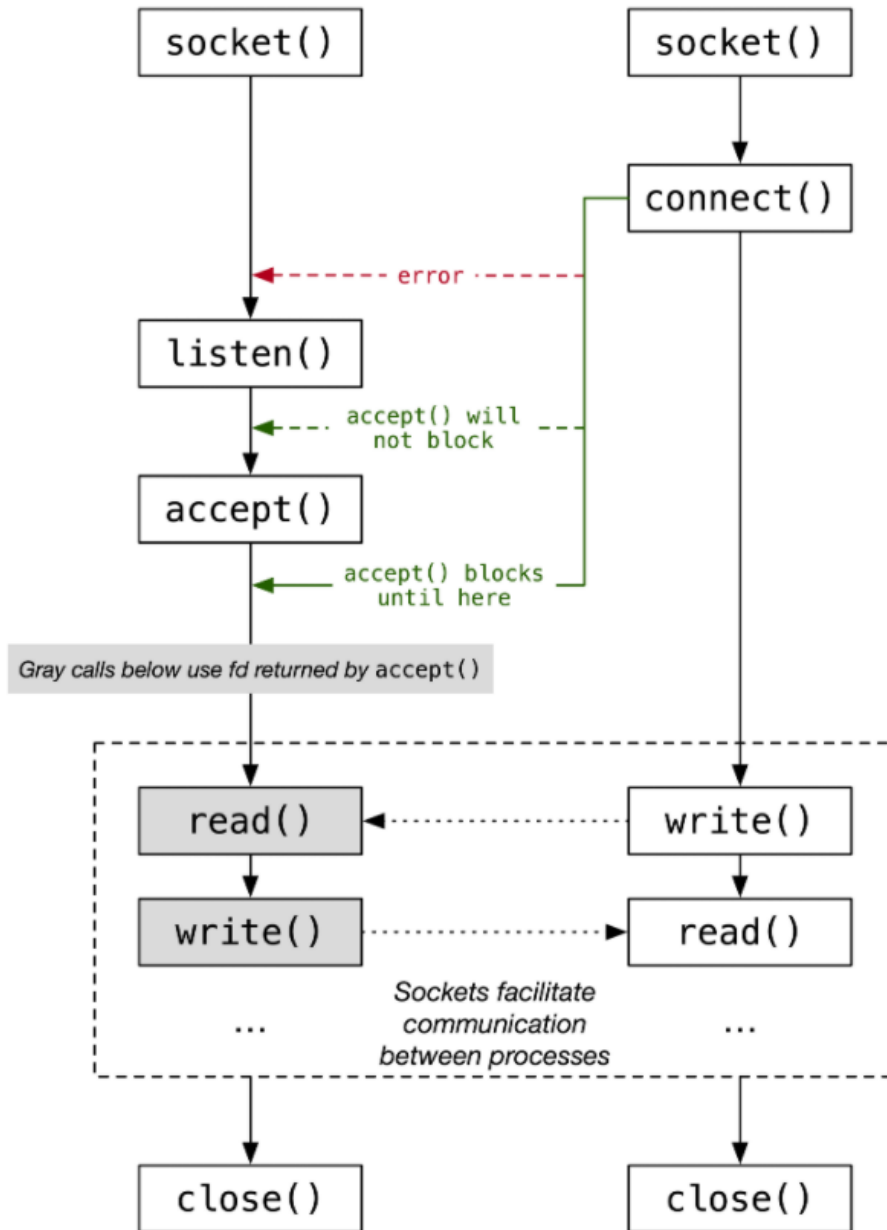


Figure 2-19: Domain Sockets flow chart

2.4 User Mode

2.4.1 User mode and system calls

As discussed in 2.3.1, the virtual space of a process is divided into user space and shared kernel space. After a user thread or process is created and the execution flow is about to switch to user mode, the kernel loads 3 in the RPL field of the segment selectors *CS*, *DS*, *ES* and *SS* (see section

2.1.1.1), *GS* will point to a block of data contains information about the current process (e.g., pid, path, arguments), while *FS* points to a block of data contains information about the current thread (e.g., tid).

When a thread enters the user mode, it cannot access any address that is in the kernel space; any access will lead to page fault which will terminate the current process. However, the user mode is very limited in its privileges and cannot deal with interrupts or IO operation for example, that is why system calls are provided to requests a service from the kernel to be executed. A system call can be performed in *CyanOS* using ``int 0x80`` which will trigger an interrupt that is designed by the kernel to handle system calls, the system call number is loaded into *EAX* register, whereas the arguments of the system call are loaded in *ECX*, *EDX*, *EBX*, *ESI*, *EDI*. And since each system call has a unique number, the kernel calls the required system call with the appropriate arguments. List of all possible system calls are mentioned in Appendix C. Figure 2-20 illustrates the execution of a system call.

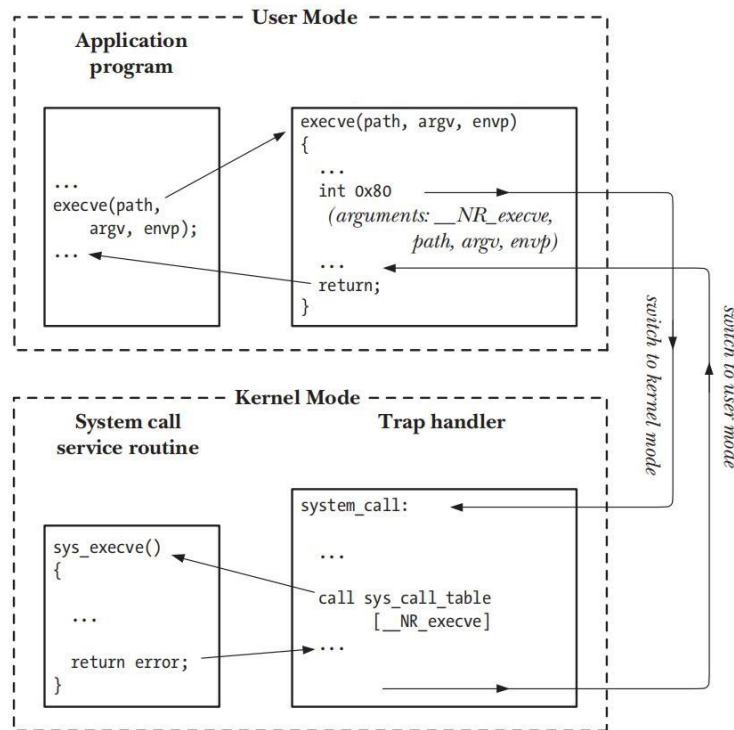


Figure 2-20: System call execution

2.4.2 ELF executable loader

Executable and Linkable Format (ELF) is a common standard file format for executable files, object code, shared libraries, and core dumps. First published in the specification for the application binary interface (ABI) of the Unix operating system version named System V Release 4 (SVR4) [21], and later in the Tool Interface Standard [22], it was quickly accepted among different vendors of Unix systems and even non-Unix systems like PlayStation 4, PlayStation 5 and Wii.

By design, the ELF format is flexible, extensible, and cross-platform. For instance, it supports different endiannesses and address sizes so it does not exclude any particular central processing unit (CPU) or instruction set architecture [23]. This is what allowed it to be adopted by many different operating systems and compilers including gcc and clang.

An ELF file consists of *ELF header*, *section headers* and *program headers*. The *ELF header* is 32 bytes long, and identifies the format of the file. It starts with a sequence of four unique bytes that are 0x7F followed by 0x45, 0x4c, and 0x46 which translates into the three letters E, L, and F. Among other values, the header also indicates whether it is an ELF file for 32 or 64-bit format, uses little or big endianness, shows the ELF version as well as for which operating system the file was compiled for in order to interoperate with the right application binary interface (ABI) and CPU instruction set. Furthermore, *section headers* describe the different regions of the binary file (i.e., section's name, offset, size, type, flags...). Whereas *program headers* describe the segments that are used at run-time, and tells the system how to create a memory image of the program in the process. It is important to note that some sections may not be a segment and will not be mapped to memory (i.e., section that contains symbols and debugging information), while some segments may not have a section in disk (i.e., segments that contain uninitialized data) as shown in Figure 2-21 and Figure 2-22.

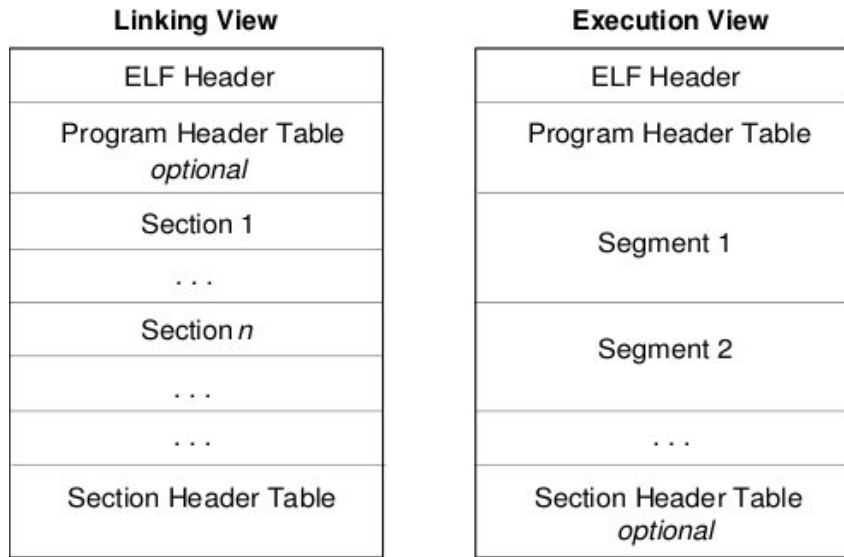


Figure 2-21: Sections and Segments

When a process being executed, the ELF loader in the kernel starts by verifying the different fields *ELF header* to ensure that this executable file is supported by the processor and the operating system. Afterwards, it uses *program headers* to allocate a memory space for each segment, then writing the corresponding data to it form the binary file.

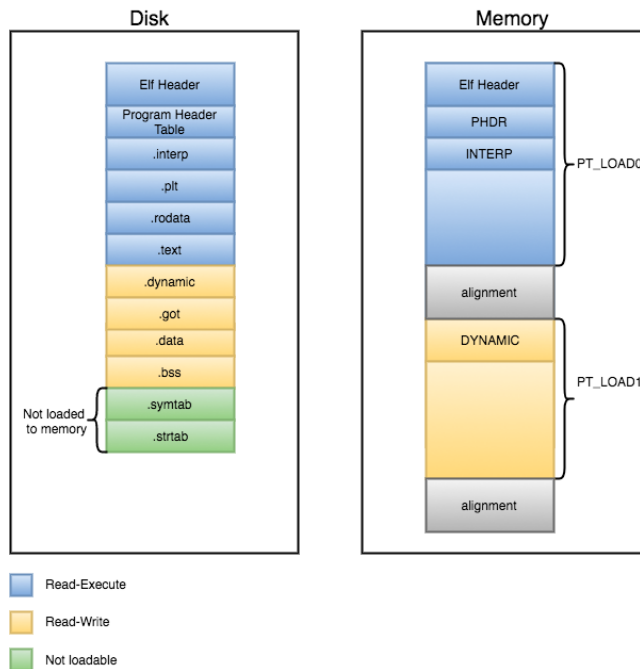


Figure 2-22: File in disk vs Program in memory

2.5 Heap Allocator

As discussed in Memory Page Allocator, the kernel provides a memory allocator that reserves memory blocks for the program, however this memory blocks are always page aligned i.e., they are always multiple of 4kb pages. This can be inconvenient and wasteful of precious resources since most programs needs to allocate memory of relatively small sizes, a more suitable tool for this is the heap allocator. There are several techniques for this purpose and the following parts will discuss the advantage and disadvantages of each one.

2.5.1 Fixed Partitioning

The simplest scheme for managing the heap memory is to partition it into equal-sized regions with fixed boundaries e.g., 1024 bytes each. Any requested memory whose size is less than or equal to the partition size can be loaded into any available partition. However, there are two difficulties with the use of equal-size fixed partitions:

- 1- A requested memory may be too big to fit into a partition. In this case, the programmer must design the program with the use of overlays so it must allocate multiple blocks with the same size.
- 2- Main memory utilization is extremely inefficient. Any requested memory, no matter how small it is, occupies an entire partition. In our example, there may be a memory request whose length is less than 100 bytes; yet it occupies a 1-kbyte partition. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as *internal fragmentation*. [24]

2.5.2 Dynamic Partitioning

To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. With dynamic partitioning, the partitions are of variable length and number. When a heap memory is requested, the allocator reserves exactly as much memory as it requires and no more. An example, using 64 Mbytes of main memory, is shown in Figure 2-23. Initially, main memory is empty, except for the OS's memory. The first three memory requests are allocated, starting where the operating system ends and occupying just enough space for each

block (see Figure 2-23b, c, d). This leaves a “hole” at the end of memory that is too small for a fourth memory allocation. Suppose Block 2 is freed from memory and a fourth allocation was placed in the place of Block 2 previously (see Figure 2-23e, f), then Block 1 is freed and a fifth allocation taken its place (see Figure 2-23g, h). Now there is two 6mb holes and one 1mb, the allocator cannot allocate any memory greater than 6mb anymore, although there is clearly 18mb left in total.

this example shows that this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as *external fragmentation*, indicating the memory that is external to all partitions becomes increasingly fragmented. This is in contrast to internal fragmentation, referred to earlier. [25]

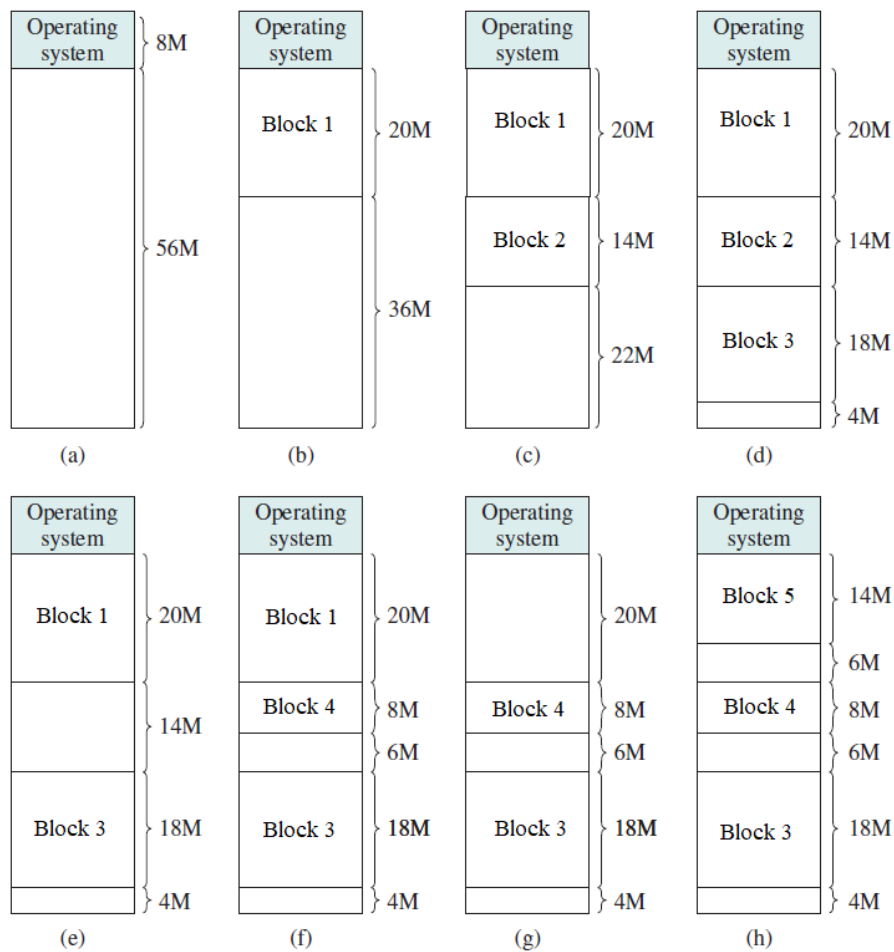


Figure 2-23: Dynamic allocation

2.5.3 Segregated Free List

To compromise between the previous techniques, the Segregated free list allocator uses a pool of multiple fixed sized regions (e.g., 1kb, 2kb 4kb 8kb) and when a certain size is requested, the allocator reserves a block from the smallest fit of fixed sized regions (e.g., 3kb will be reserved in 4kb region). This technique has no external fragmentation, while minimizing the internal fragmentation. Another advantage of this allocator is reserving similar memory objects in physically close locations, which will help the cache to fasten the memory access. That is why it is implemented in *CyanOS*. Figure 2-24 illustrates the structure of segregated free list.

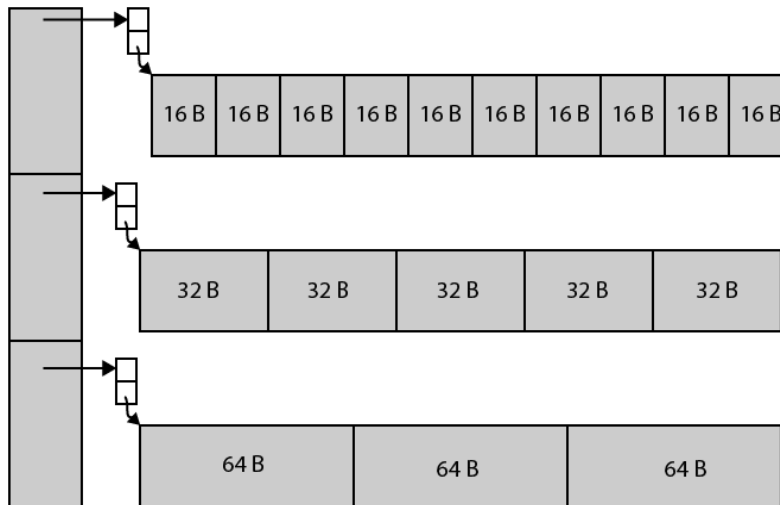


Figure 2-24: Segregated free list

2.6 Virtual File System

A virtual file system (VFS) or virtual filesystem switch is an abstract layer on top of a more concrete file system, device driver, network sockets or virtual kernel modules. The purpose of a VFS is to allow client applications to access different types of objects in the kernel in a uniform way. A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference. It can be used to bridge the differences in Windows, classic Mac OS/macOS and Unix filesystems, so that applications can access files on local file systems of those types without having to know what type of file system they are accessing. A VFS specifies an interface (or a "contract") between the kernel and a concrete file system. Therefore, it is easy to add support for new file system types to the kernel simply by fulfilling the contract [26]. Figure 2-25 shows how the VFS is used by multiple units.

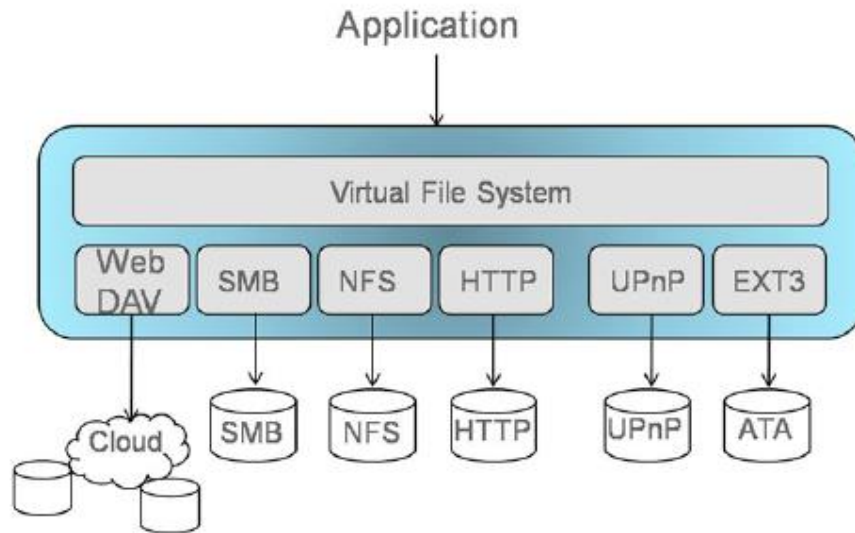


Figure 2-25: Virtual file system

It is important to note that the design of the VFS is an important factor to determine the flexibility of the system, a well constructed VFS can help the programmer to port any new file system or device driver to the new operating system with minimum work.

2.6.1 VFS Implementation

CyanOS has a virtual class called *FSNode* that contains 13 virtual functions which are used to interact with a certain node in the VFS, see Table 2-7 for the complete list of these functions. If a programmer needs to add his own driver or FS into the VFS, he needs to inherit from *FSNode* and implement the functions he needs. If a function from the base class (i.e., *FSNode*) is not implemented in the derived class, it will be considered as unused, and if the user calls it for the certain node, it will return an error stating that the operation is invalid. The first node must be mounted to the VFS, then it can have multiple child nodes, which they can have child nodes too.

Examples of VFS nodes in the system are *USTAR* filesystem, *Pipes*, *Domain Sockets*, *IP Sockets*, keyboard driver, VGA driver.

2.6.2 FileDescription

FileDescription is a kernel object that is a wrapper around *FSNode* to describe the current state of an opened *FSNode*. It contains mode, flags, and permission of the opened node as well as the current reading/writing offset. The operations on *FileDescription* are the same as the ones on

FSNode in Table 2-7 with slightly different parameters, and with addition to *seek* operation which is used repositions the file offset of the opened *FSNode* reading/writing position.

E.g., If a file with the size 1000 byte is opened by a program, the *FileDescription* will have zero as the current offset, when reading/writing some data; say 50 bytes, the offset will be increased by 50.

Besides *FileDescription*, the kernel has *ProcessDescriptions* and *ThreadDescriptions* which are used to describe the state of an opened process or thread.

Function	Description
open	Inform that node that it has been opened.
close	Inform that node that it has been closed.
create	Create a child node inside this node.
remove	Remove a child node from this node.
link	Create a symbolic link to this node.
unlink	Delete a symbolic link to this node.
connect	Connect to a server.
listen	Mark the server as passive to the incoming connections.
accept	Accept a given connection.
read	Read a number of bytes from the node.
write	Write a number of bytes to the node.
dir_lookup	Get a child node from its name.
dir_query	Enumerate all child nodes

Table 2-7: *FSNode* operations

2.6.3 Handles

Handles are integers that are used in the user applications to reference the *FileDescriptions*, *ProcessDescriptions* and *ThreadDescriptions*, and uniquely identify them in a process. Handle is always a first parameter to system calls that are dealing with kernel objects such as files, processes, threads. Each process has table of *Descriptions*, and a handle is basically an index in that table, and closed Handle does not delete an its entry in the table; it will be marked as closed instead. Figure 2-26 shows the relationship between handles and *FileDescriptions*.

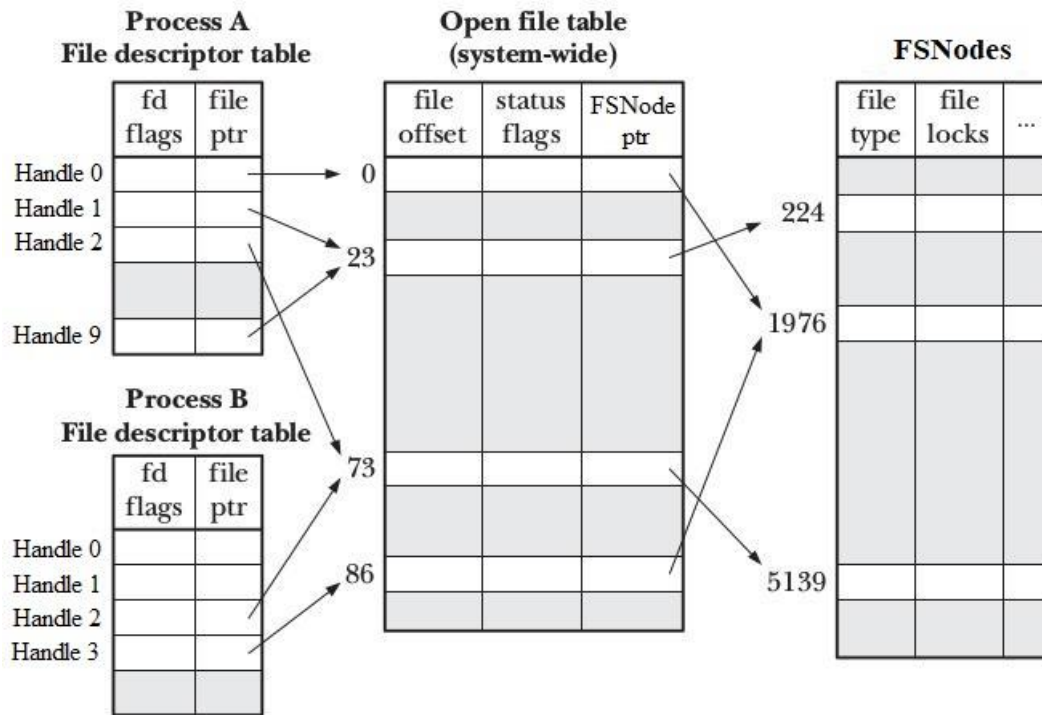


Figure 2-26: Relationship between handlers and FileDescription

2.7 Kernel Architecture

2.7.1 Monolithic Kernel Architecture

Monolithic operating system services are compiled as single, monolithic process that runs in a single memory address space in kernel mode, whereas applications run in user mode and can request system services from the kernel. Thus, the kernel has two tasks; resource management and a driver for devices, examples of monolithic operating systems are SerenityOS, Unix and Linux. Figure 2-27 shows high-level perspective, a monolithic kernel structure.

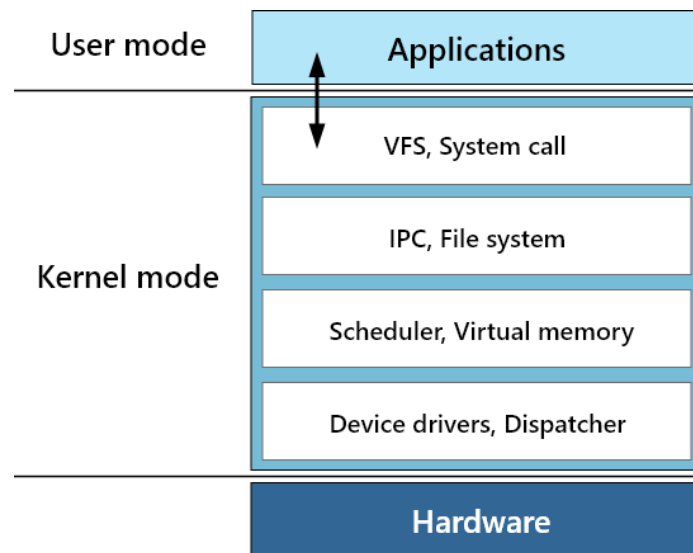


Figure 2-27: Monolithic kernel architecture

2.7.2 Microkernel Architecture

The microkernel architecture provides the minimum of functionality and services needed to run in the kernel while the rest of the OS services run as separate processes with different address spaces outside the kernel. They communicate by different IPC mechanisms such as message parsing. The kernel's job is to handle IPC, interrupt, multitasking and virtual memory, while the device drivers and other services are in the user mode as separate processes. This architecture is illustrated in Figure 2-28. Examples of microkernel operating systems are Minix 3, AmigaOS and beOS.

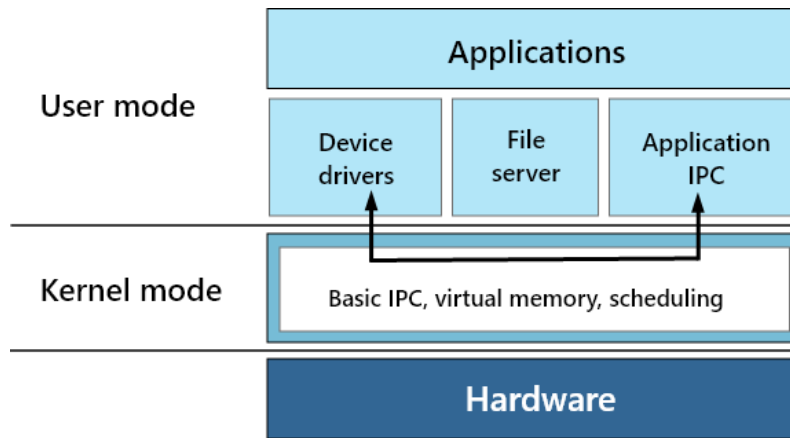


Figure 2-28: Microkernel architecture

2.7.3 Micro vs Monolithic kernel

The two approaches differ primarily in the implementation where in the monolithic architecture all the kernel and OS services run in a single address space whereas, in the microkernel only minimum kernel services are kept within a single address space and the rest are run as different processes with separate address spaces. Each design has its advantages and disadvantages which will be discussed below.

Size: the size of monolithic kernel is comparatively larger than microkernel because all OS services are all compiled into single file that will be loaded to the memory. However, in the microkernel, the bare minimum services are contained in the compiled kernel which makes its size smaller.

Speed: the execution of the monolithic kernel is notably faster as communication between OS services in the same address space does not require any context switch nor switching the virtual address space, unlike with the micro architecture where these services communicate through heavy use of IPC mechanisms which introduces substantial amount of overhead in the system.

Extensibility: adding new features to microkernel is as simple as adding a new process for that feature. Whereas for a monolithic kernel, new features require modification and recompilation of the whole kernel.

Security: the fact that OS services reside in different address spaces for a microkernel means that if a failure occurs in any of these services, the operating system and other services remain unaffected. On the other hands, if a service fails in monolithic kernel, the entire system will fail.

2.7.4 Hybrid Kernel Architecture

Hybrid kernel is based on a combination of both architectures; it combines the speed and simpler design of monolithic kernel with the modularity and execution safety of microkernel. This is why it is the architecture implemented in *CyanOS*.

A hybrid kernel runs some of its important services in the kernel space to reduce the performance overhead of a traditional microkernel, while still running some other services in the user space. For instance, a hybrid kernel design may keep the bus controllers like PCI or USB inside the kernel, whereas the individual drivers of the devices attached to these busses as user mode programs outside the kernel. This allows bus controllers to be fast and reliable while keeping the device drivers in a safer environment, and can be easily modified and added.

Chapter 3: Results and Discussion

This chapter will discuss some user mode programs and how they can be compiled, executed and interact with the operating system.

To compile a user program in *CyanOS*, a cross compiler must be present in your host system which is used to build executable code for a platform other than the host it is running in (e.g., Windows or Linux). In contrast to normal compilers, cross compilers will not assume any configuration about the current environment, and it does not use any libraries or headers that are not built-in in the C++ language itself. Appendix D.2 discusses how to build a cross compiler `gcc`.

In addition to a cross compiler, user programs need to be linked with a library called *systemlib*; this library contains the important functions needed for a program to work in *CyanOS*. It initializes some information fields about the current process and thread before calling *main* function of the program, manages the IO operations like *printf*, *get_char* and *scanf*, and has all system call functions. The actual entry point of a program is in *systemlib*, then it calls the *main* function.

3.1 User program discussion: *shell*

The shell is the first user mode program to be executed by the operating system; like Linux, it is a simple interface to explore the file system and execute other programs and view their output.

As shown in Figure 3-1 and Figure 3-2, the shell is merely a super loop that keeps waiting for input characters from the keyboard and parses the corresponding commands. It starts by calling *get_char* from *systemlib*, this function opens a handle to the keyboard driver in */devices/keyboard* using *OpenFile* system call, and then it tries to read a character using *ReadFile* system call which will block the current thread until a key is pressed in the keyboard. The shell saves the entered characters in buffer, and when the enter key is pressed; it tries to parse the given command.

If the given command is recognized by the shell, like *ls* (list all files in the current directory), *cd* (change the current directory), *cwd* (displays the current directory full path), it will be performed,

otherwise, the shell assumes that the user tries to execute a program.

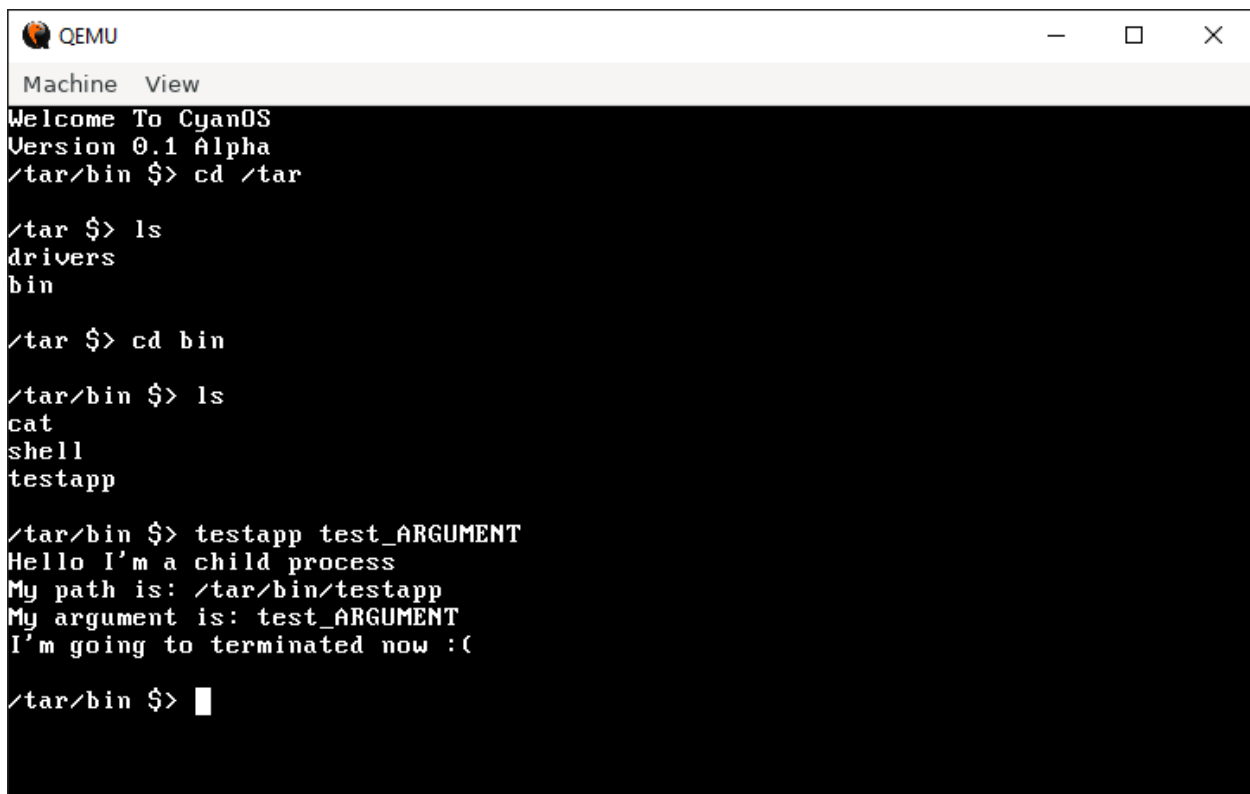
```
char input_char;
const size_t max = 1000;
char buff[max];
int index = 0;
while ((input_char = get_char())) {
    if (index < max) {
        if (input_char == '\n') {
            printf("\n");
            buff[index] = 0;
            execute_command(buff);
            index = 0;
        } else if (input_char == '\b') {
            if (index > 0) {
                printf("\b");
                index--;
            }
        } else {
            buff[index++] = input_char;
            putchar(input_char);
        }
    } else {
        printf("\ncommand is too long!");
        index = 0; // command is too long
    }
}
```

Figure 3-1: Shell's pseudo code

```
Handle child = CreateProcess(working_directory + input_command, args, 0);
if (!child) {
    printf("Undefined command.\n");
    return;
}
WaitSignal(child, 0);
CloseHandle(child);
```

Figure 3-2: Another shell's pseudo code

To start a new process, *CreateProcess* is called with the path of the program, if it fails it returns zero (like all system calls) and the error code can be read using *GetLastError*. After that, the shell calls *WaitSignal* system call which will block until the process of the given handle is terminated. Finally, *CloseHandle* is used to release the kernel resources of the handle. Figure 3-3 shows how some of shell's commands can be used.



```
QEMU
Machine View
Welcome To CyanOS
Version 0.1 Alpha
/tar/bin $> cd /tar

/tar $> ls
drivers
bin

/tar $> cd bin

/tar/bin $> ls
cat
shell
testapp

/tar/bin $> testapp test_ARGUMENT
Hello I'm a child process
My path is: /tar/bin/testapp
My argument is: test_ARGUMENT
I'm going to terminated now :(

/tar/bin $> █
```

Figure 3-3: Navigate directories and execute programs in shell

3.2 User program discussion: *cat*

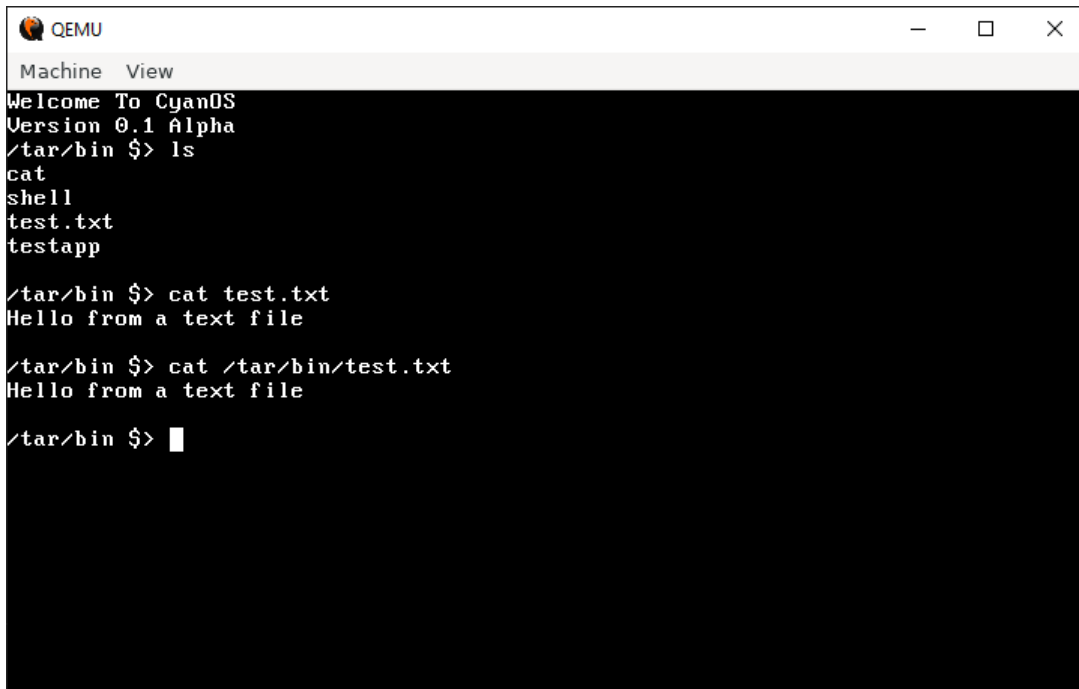
Cat is another simple program similar to Linux's, it is used to read text files and print them to the screen.

As shown in the code snippet in Figure 3-4, *Cat* starts by opening a file handle to the file passed to it as an argument, then it uses *QueryFileInformation* system call to get information about the opened file and saves it in a structure *FileInfo*. An important information needed for this structure is the size of the file, which is used to create a heap buffer that fits and avoids buffer overflow. Then the system call *ReadFile* is called to read the file and fill the given buffer. Next, the text file is printed using *printf* that writes to a device driver */devices/console* which is used to

control the text mode screen. And finally, the program frees the heap memory and the kernel resources of the handle using `delete[]` and `CloseHandle` respectively. Figure 3-5 shows how `cat` can be used to read text files

```
Handle fd = OpenFile(argv[1], OM_WRITE | OM_READ, OF_OPEN_EXISTING);
if ((result = GetLastError())) {
    return result;
}
FileInfo info;
QueryFileInformation(fd, &info);
if ((result = GetLastError())) {
    CloseHandle(fd);
    return result;
}
char* buff = new char[info.size + 1];
memset(buff, 0, info.size + 1);
ReadFile(fd, buff, info.size);
if ((result = GetLastError())) {
    CloseHandle(fd);
    return result;
}
printf(buff);
printf("\n");
delete[] buff;
CloseHandle(fd);
```

Figure 3-4: Code snippet from `cat`



```
QEMU
Machine View
Welcome To CyanOS
Version 0.1 Alpha
/tar/bin $> ls
cat
shell
test.txt
testapp

/tar/bin $> cat test.txt
Hello from a text file

/tar/bin $> cat /tar/bin/test.txt
Hello from a text file

/tar/bin $> █
```

Figure 3-5: Using cat to read text files

Conclusion and Future Work

In this report we have presented the theory, design and the implementation of the different components of an operating system kernel. We discussed the design of various approaches suggested by books and papers, as well as the ones implemented mature operating systems. And it also argued why the approach chosen was the most suitable for this operating system.

It should be stressed on the importance of the design scheduler, the context switch and the interprocess communication to have good performance, while the hybrid kernel architecture and the design of the virtual file system helps to maintain a scalable system.

And although this operating system is initially designed for IA-32 processors, the project's code is organized in a such way that makes the architecture-related functions are collected in files within the same directory while they are called by other higher-level functions. This makes it easier to port the operating system to other architecture since minimum code will be rewritten which mainly related to paging and interrupts.

Due the limited time, not all of the planned features were implemented in this project, thus, we will discuss some of them, and explains how can you contribute to this open source project. The first important feature is the networking stack. Although CyanOS has fully functional networking stack (with protocols IPv4, ICMP, TCP, UDP, DHCP, ARP and DNS), it wasn't really mentioned in this report due the number of pages constraint while this topic needed huge discussion. Our network stack implementation was good enough for the most part specially in the primitive protocols like IPv4, UDP, DHCP, ARP and DNS, however, more complex protocols like TCP need a better error handling and optimization specially with the internal buffer. Additionally, another layer can be added for handling HTTP requests and maybe even requests through Transport Layer Security (TLS) encryptions.

Currently, the main display is text mode, so the next important feature is the graphical user interface (GUI). It works by having multiple layers on top of each other; at the lowest layer there will be a GPU driver that manages the GPU configurations and writes pixels on the screen. After that, the OS should provide a higher layer to draw particular shapes on the screen and manages input devices like mouse clicks and keyboard strokes. The final layer is a library provided to user

mode applications, its purpose is to manage high level GUI components like textboxes, labels, buttons and windows, and handle any events like moving windows, clicking on the button or writing on a textbox.

And the last feature is using dynamic shared libraries instead of static libraries; currently, libraries like *systemlib* are statically linked with every user application, which means that all executable files have an identical part which is the code of that library. A better mechanism is to have libraries dynamically linked like Dynamic-link library (DLL) in windows; the executable will have just the name of the library, the operating system then loads the library in a shared memory between all processes. This way, the same library code will be not be in multiple executable files nor will be loaded into the memory of multiple processes.

Appendix A: Modern C++ Features

A.1 Templates

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept. There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>. [27]

```
template <typename T> T add(T num1, T num2) // template function that has a template T.
{
    return num1 + num2;
}
void main()
{
    char result1 = add<char>(1, 5);           // Returns the addition of two char variables.
    char result2 = add<int>(1, -2);          // Returns the addition of two int variables.
    char result3 = add<double>(1.2, 4.3);    // Returns the addition of two double variables.
}
```

Figure A-1: Template function

As shown in the example in Figure A-1, the function *add* is a blue print for the addition operation that works on multiple data types. the function later is called by specifying the type of the variable T.

A.2 Lambda Expressions

Lambda is an object that is a wrapper around an anonymous function, that can be invoked, stored or passed as an argument. They are usually used to encapsulate few lines of code are passed to algorithms or asynchronous methods.

As shown in the example in Figure A-2, a lambda object `check_even_lambda` is created which holds the few lines of code that checks that the passed number is even. Later in the loop the lambda object is invoked by passing a number to it and returns a boolean result.

```
auto check_even_lambda = [](int number) {
    if (number % 2 == 0)
        return true;
    else
        return false;
};

for (size_t i = 0; i < 100; i++) {
    if (check_even_lambda(i)) {
        printf("number %d is even!", i);
    }
}
```

Figure A-2: Lambda expression example

Appendix B: Data Structures

This part contains some data structures that used in *CyanOS* and acts like the *standard library* equivalent in C++20. It discussed some the data structure containers and their main functions.

B.1 Iterators

Iterators are not data structure containers per say, but more like pointers to elements of data structure containers. Each container has its own iterator but they all share the same interface functions.

Function	Description
<code>operator++ ()</code>	Moves the iterator to the next element in the container.
<code>operator-- ()</code>	Moves the iterator to the previous element in the container.
<code>operator+ (int count)</code>	Advances the iterator by <u>count</u> elements from the current one.
<code>operator- (int count)</code>	Advances the iterator by <u>count</u> elements from the current one.
<code>T operator* ()</code>	Returns the value of the element pointed by this iterator.
<code>bool operator== (const Iterator& other)</code>	Checks whether two iterators point to the same element.
<code>bool operator!= (const Iterator & other)</code>	Checks whether two iterators point to the different elements.

B.2 Vector

A data container that stores elements in contiguous memory locations, thus, can be accessed by their index. The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted.

Function	Description
Iterator begin()	Returns an iterator that points to the first element of the container.
Iterator end()	Returns an iterator that indicates that the last element has been passed.
Iterator insert(Iterator element, U&& new_node)	Adds a new <u>element</u> in the position of the iterator <u>node</u> .
Iterator push_front(U&& new_data)	Adds an element <u>new_data</u> to the start of the container.
Iterator push_back(U&& new_data)	Adds an element <u>new_data</u> to the end of the container.
void reserve(size_t size)	Reserves a new size for the internal storage, it must be greater than the current capacity.
void pop_front()	Removes the first element of the container.
void pop_back()	Removes the last element of the container.
void clear()	Removes all elements of the container.
void remove(Iterator element);	Removes an element pointed by the iterator. This will invalidate the iterator, so it must be obtained again.
bool remove_if(Predicate predicate)	Removes all elements that satisfy the condition that are checked in the lambda function <u>predicate</u> .
Iterator find(const T& element)	Returns an iterator of an element if it is found in the container.
T& operator[](size_t index)	Returns an element pointed by the provided index.
size_t size()	Returns the number of elements of that are actually in the container.
size_t capacity()	Returns the maximum capacity of element that the internal storage can hold. It can be increased

	by reserve function.
--	----------------------

B.3 List

A data container that is very similar to Vector, but it stores the elements in doubly linked list, however since the element are not in contiguous locations, elements can not be accessed by their index.

Function	Description
Iterator begin()	Returns an iterator that points to the first element of the container.
Iterator end()	Returns an iterator that indicates that the last element has been passed.
Iterator insert(Iterator element, U&& new_node)	Adds a new element in the position of the iterator <u>node</u> .
Iterator push_front(U&& new_data)	Adds a new element to the start of the container.
Iterator push_back(U&& new_data)	Adds a new element to the end of the container.
void reserve(size_t size)	Reserves a new size for the internal storage, it must be greater than the current capacity.
void pop_front()	Removes the first element of the container.
void pop_back()	Removes the last element of the container.
void clear()	Removes all elements of the container.
void remove(Iterator element);	Removes an element pointed by the iterator. This will invalidate the iterator, so it must be obtained again.
bool remove_if(Predicate predicate)	Removes all elements that satisfy the condition that are checked in the lambda function <u>predicate</u> .
Iterator find(const T& element)	Returns an iterator of an element if it is found in the container.

size_t size()	Returns the number of elements of that are actually in the container.
---------------	---

B.4 String

A container that manages the ascii strings i.e., sequences of char-like objects. It stores is as a pointer of an array and a size.

Function	Description
String& operator+= (const String& other)	Concatenates a string with another string.
String operator+ (const String& other)	Creates a new string container with concatenation of the current string and another string.
String substr(size_t pos, size_t len)	Creates a new string that is part of the current string by a position and a size.
size_t find(const String& str, size_t pos = 0)	Finds the position of a substring in this string.
String& push_front(char c)	Adds a character to the start of the string.
String& push_back(char c)	Adds a character to the end of the string.
String& insert(size_t pos, const String& str)	Adds a string in a position..
void erase(size_t pos, size_t len)	Removes part of the string specified by a position and a size.
char operator[] (size_t index)	Returns a character from an index.
size_t length()	Returns the length of the string.

B.5 Stack

A container stores elements in contiguous memory region but gives the functionality of a stack i.e., LIFO (last-in, first-out)

Function	Description
void push(U&&)	Pushes an element to the stack.
T pop()	Returns and removes the last element from the stack.
size_t size();	Returns the number of elements in the stack.

B.6 CircularBuffer

A container that uses a fixed-size buffer as if it were connected end-to-end in a circle. gives the functionality of a queue i.e., FIFO (first-in, first-out)

Function	Description
void queue(U&&)	Pushes an element to the queue.
T dequeue();	Returns and removes the first element of the queue.
size_t size();	Returns the number of elements in the buffer.
size_t capacity()	Returns the total number of elements that the container can hold
bool is_full()	Checks whether the buffer is full.

B.7 Bitmap

A container that holds a list of bits in contiguous memory region.

Function	Description
void set(size_t position)	Sets a bit a certain position to one.
void set_range(size_t position, size_t count)	Sets a range of bit in a certain position to one.
void clear(size_t position)	Sets a bit a certain position to zero.
void clear_range(size_t position, size_t count)	Sets a range of bit in a certain position to zero.
bool check_set(size_t position)	Checks a certain bit if it is set to one.
bool check_clear(size_t position)	Checks a certain bit if it is set to zero.

B.8 Result

A container to handle errors in the operating system. It either contains a type returned by a function or an error. A function may return Result<T> with T is a type of the data to be returned if no error happened.

Function	Description
bool is_error()	Checks whether the returned function has an error.

unsigned error()	Returns the error code if there is an error, otherwise it returns zero.
T& value()	Returns the original data from the function if there is no error.

Appendix C: System Calls

OpenFile	Creates a new file/device or opens an existing one, and returns the file handle pointing the <i>FileDescription</i> of that file.
Socket	creates a new socket either domain socket or network IP socket (TCP or UDP), it returns a handle to the newly created socket.
Pipe	Creates a new pipe, and return handle to it.
ReadFile	Reads a file/device and fills a user buffer provided to it. This system call may block the current thread when the file/device is not ready to be read.
WriteFile	Write a file/device from a user buffer provided to it. This system call may block the current thread when the file/device is not ready to be written into.
QueryDirectory	Lists of all the files in given directory, the information is filled in <i>FileInfo</i> structure.
QueryFileInformation	Gives information about the a given file (e.g., its size), the information is filled in <i>FileInfo</i> structure.
CloseHandle	Closes a handle and releases the kernel resources reserved for it and the linked <i>FileDescription</i> .
Sleep	Blocks the current thread a given amount of time.
Yield	Gives up the current thread's time slice and schedule another thread.
CreateThread	Creates a new thread that executes a certain address, and returns its handle.
SuspendThread	Suspends a thread from execution.
ResumeThread	Resumes a suspended thread to execution.
TerminateThread	Terminates a thread.
CreateProcess	Creates a new process from a give file in file system, and returns its handle.
SuspendProcess	Suspends all threads in a process.
ResumeProcess	Resumes all suspended thread in a process.

TerminateProcess	Terminates a process.
WaitSignal	Blocks the current thread until the thread or process of given handle terminates.
VirtualAlloc	Allocates a block of memory, the memory will be aligned to the page size i.e., 4kb
VirtualFree	Frees a block of memory and allows it to be reused in the future.

Appendix D: How to Compile *CyanOS*

The normal compilers cannot compile this operating system correctly due their assumptions of the environment by compiler since no headers or libraries of the host operating system can be used. That is why the only way to compile is using a cross-compiler, we will be using gcc 10 for its popularity and its support to the latest features of C++.

D.1 Building cross compiler gcc

Before starting the process of building the compiler a regular gcc compiler (not cross-compiler) is required. In addition to that, few dependencies must exist in the host machine; assuming it is Linux (you can use a Linux environment in Windows 10 using WSL2). The packages can be installed using the commands:

```
make install
sudo apt-get update
sudo apt install build-essential bison flex libgmp3-dev libmpc-dev libmpfr-dev texinfo
```

Then download and extract the latest version of *binutils* from ftp.gnu.org/gnu/binutils/ . Then enter to the extracted directory and use the following commands to build it.

```
export PREFIX="$HOME/opt/cross"
export TARGET=i686-elf
export PATH="$PREFIX/bin:$PATH"

mkdir build-binutils
cd build-binutils
../configure --target=$TARGET --prefix="$PREFIX" --with-sysroot --disable-nls
--disable-werror
make
make install
```

Now after building *binutils*, you need to build *gcc*. Download the latest version from ftp.gnu.org/gnu/gcc . Then enter the extracted directory and use the following commands to build it.

```
mkdir build-gcc
cd build-gcc
../configure --target=$TARGET --prefix="$PREFIX" --disable-nls --enable-languages=c,c++
    --without-headers
make all-gcc
make all-target-libgcc
make install-gcc
make install-target-libgcc
```

D.2 Building the operating system

Now to build the operating system, follow the commands to download the dependencies:

```
sudo apt-get install gcc-multilib g++-multilib build-essential nasm python3 cmake
    grub2 xorriso mtools qemu
```

Then build the system

```
git clone --recursive https://github.com/AymenSekhri/CyanOS.git
cd ./CyanOS
mkdir build && cd build
cmake .. -G "Unix Makefiles"
make
```

Bibliography

- [1] A. Tanenbaum and A. Woodhull, Operating Systems Design Implementation, 1992, p. 1.
- [2] P. Denning, Thrashing: Its causes and prevention, 1968.
- [3] "Embedded Systems - Interrupts," [Online]. Available: https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm.
- [4] A. Tanenbaum, Modern Operating Systems, 4th, Ed., 1992, p. 149.
- [5] "x86 - Wikipedia," [Online]. Available: <https://en.wikipedia.org/wiki/X86>.
- [6] R. W. Green, "What do IA-32, Intel 64 and IA-64 Architecture mean?," May 5, 2009. [Online]. Available: <https://software.intel.com/en-us/articles/ia-32-intelr-64-ia-64-architecture-mean/>.
- [7] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 64.
- [8] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," 2021. [Online]. Available: Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, 3C, and 3D: System Programming Guide.
- [9] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 91.
- [10] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 95.
- [11] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 203.
- [12] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 204.
- [13] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 90.
- [14] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 114.
- [15] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 115.
- [16] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, p. 117.
- [17] "Intel 8259," [Online]. Available: https://en.wikipedia.org/wiki/Intel_8259.
- [18] V. C. Hamacher, Z. G. Vranesic and S. G. Zaky, Computer Organization (5th ed.), 2002.
- [19] "Peripheral Component Interconnect," [Online]. Available: <https://wiki.osdev.org/PCI>.

- [20] G. G. a. P. B. G. Avi Silberschatz, Operating System Concepts, 2002, p. 140.
- [21] " System V Application Binary Interface," 1997.
- [22] "Executable and Linking Format (ELF) Specification," 1995.
- [23] "Executable and Linkable Format," [Online]. Available:
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.
- [24] W. Stallings, Operating Systems: Internals and Design Principles, 1992, p. 344.
- [25] W. Stallings, Operating Systems: Internals and Design Principles, 1992, p. 349.
- [26] "Virtual file system," [Online]. Available: en.wikipedia.org/wiki/Virtual_file_system.
- [27] "C++ Templates," [Online]. Available:
https://www.tutorialspoint.com/cplusplus/cpp_templates.htm.