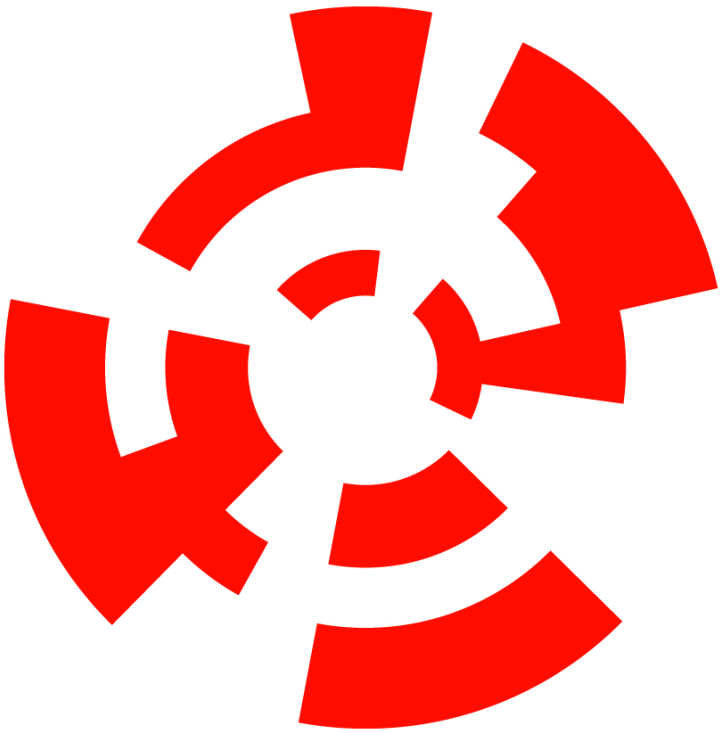
Operating Systems and Human Machine Interaction



MICHAEL COLLINS
HETAC BSC IN SOFTWARE DEVELOPMENT
DEPARTMENT OF INFORMATION TECHNOLOGY

LIMERICK INSTITUTE OF TECHNOLOGY
2018

Abstract

Computer systems have evolved since their appearance in the 1940's. At first these machines were usable only by a few expert engineers. Thanks to the efforts of generations of engineers, the computers of today are usable by the entire population. This change is due, in no small part, to the development of the operating system. This document highlights the importance of the human in a computer system. The computer exists merely to enhance human productivity. Successful computer systems are those which increase productivity or improve human comfort. Losing sight of these requirements results in systems which are of little value in the real world. This project investigates the design of an operating system with a focus on maximizing user productivity whilst minimizing human effort.

In researching this area, the author has focussed on previous papers which have revolutionized computer human interaction. Corbató's invention of the Time-Sharing system, in 1962, allowed multiple users to access a single computer through remote terminals. The UNIX system, of the 1970's, made users more productive by applying a pipeline design to its innovative command line interface. The Xerox Alto invented the Graphical User Interface, which is ubiquitous in today's world. The research of Liedtke, in the mid 1990's, redefined inter-process communication in a world connected by computers. These results and others, form the basis of this project; the design of a multi-tasking operating system referred to throughout this document as *Spartan*.

Acknowledgements

This project would not have been possible without the support of many people:

First of all, I would like to thank my supervisor, Brendan Watson. For his commitment to this role, and for always being available for guidance. For confronting me with the approach I was taking, and for continuously pushing me towards new challenges, and unexpected directions. His guidance, expertise, and experience, proved an invaluable resource, greatly enriching both the content and my own appreciation of this work.

Special thanks must go to the engineers at Analog Devices, to Martin Fogarty, and Brian Keane, for taking an interest in this project. Their expertise and familiarity with the target execution environment; the IBM PC Compatible, was an invaluable resource. This guidance had a great impact on this project, simplifying areas which previously seemed out of reach.

Finally, and most importantly, I would like to thank my family and my partner Alice, without whose support and understanding, this project would not have been possible.

Git Repository

The software accompanying this project is open sourced and is distributed under the MIT licence. The source code has been made publicly available and the Git Repository is accessible from the following URL:

https://github.com/collinsmichael/spartan.git

Table of Contents

List of Figures

Chapter 1 – Introduction

Computers are complicated systems, but operating systems make them easy to use. Human-machine interaction has evolved over time to produce the point and click graphical user interface used widely today. Operating systems make this possible by managing hardware resources and providing sensible abstractions. This project investigates the mechanisms used to effectively manage resources and improve the usability of a computer.

Operating systems are divided into two parts: The Kernel and the Shell. The Kernel interacts directly with the hardware and is designed to make the computer usable at the most basic level. Device Management, Memory Management, and Process Scheduling are essential components which make up a Kernel. Chapter 2 investigates the role of the Kernel and how it achieves efficiency by applying appropriate algorithms. The Shell is the larger part of an operating system, and its purpose is to make the computer easy to use. The Shell provides a user interface, such as a command line, or a windowing system. Chapter 3 defines the role of the Shell and outlines how the Shell builds upon the abstractions of the Kernel to provide generic data interchange, and intuitive user interactions.

This introduction outlines the development of operating system technology from 1960 onwards. The evolution of these systems has been driven by financial cost, and human comfort. The timeline of operating systems is illustrative of both practicality and ingenuity. The cost of early mainframes provided an incentive to increase throughput and eliminate downtime. Over time the cost of computers continued to fall, and with this came a shift in priorities from cost cutting to human comfort. This discussion will investigate these discoveries, and their impact on the people who interact with these systems on day to day basis.

1.1 Batch Processing

Early mainframe computers, such as the IBM 701, were sold with built in firmware called a Resident Monitor. These Resident Monitors are the forerunners to modern operating systems. From Larner (1987), we have a complete specification of the FORTRAN Monitor System. The services it provides are those expected of a modern BIOS (Basic Input Output System). The system lacks any concept of a file and is concerned only with IO such as: reading magnetic tape, or punch cards. Because of the low-level nature of the services provided by a Resident Monitor, the author takes a conservative approach and does not consider these early systems to be true operating systems. The role of the operating system was instead filled by human operators. These operators acted as a point of contact for the computer users, would schedule jobs for execution on behalf of those users, and performed book keeping duties when required, such as changing magnetic tapes.

According to Corbató (1963), at the time computers were rented at a cost between $300 to $600 per hour. Batch processing was used to maximise uptime. Under the Batch Processing system, the operator loads one batch of jobs into the computer at a time and gathers the output for later collection. Loading a batch into the system is a lengthy procedure. While one batch is executing the operator is busy compiling the next batch of jobs from paper card into the much faster magnetic tape. Occasionally these running jobs would require operator assistance when a program required data from a different magnetic tape. This would cause the entire system to halt while the operator changed tapes. The turnaround time between submitting a job and collecting results could take up to a day. Batch processing minimized, but did not eliminate, the down time spent switching between jobs. Corbató (1963), found the turnaround time of these systems to be a source of frustration for computer users, who compensated by having multiple jobs in development. This allowed them to work from their desk while waiting for results.

1.2 Time Sharing

The need for eliminating downtime, and reducing turnaround times, led Fernando José Corbató to develop the Time-Sharing System. Corbató (1962), used a periodic timer, analogous to an alarm clock, to signal to the system when it was time to switch between tasks. Corbató led development on the MULTICS Time Sharing System, which first came into use, in 1962, while it was still being designed. Time-Sharing allowed multiple users to interact directly with the computer at the same time. Time-Sharing changed the way we use computers. Users who would have previously submitted jobs to an operator, were now directly interacting with the system. Each user got a time slice of processing time which was rotated between all users in a round robin fashion. The task switching happened so quickly that by the time the user got another time slice they would not have noticed that they had been waiting.

Under the time-sharing system all resources, including time, are shared between users. This is a dramatic difference from the Batch Processing system, where each job has complete control over the entire system. However, supporting multiple users increased the complexity of the operating system. Keeping track of which resources have been assigned to which user requires a high degree of organisation. This led to the development of the abstractions which are still an integral part of modern operating systems. Corbató (1962), introduced the concept of a task, and that of a file. Ossanna, Mikus, and Dunten (1965), developed multiplexed device IO, which is essentially standard input output (IO).

1.3 UNIX

MULTICS was a sophisticated system and ahead of its time in many respects, however, it required large amounts of memory to run. At the time memory was expensive and the cost of the hardware required to run MULTICS was out of reach for all but the largest organisations. At this time, even large companies rented computers on an hourly basis. In an interview with

Peter Seibel, Ken Thompson (2009), criticized the MULTICS system of being over-engineered. Bell Labs designed UNIX to be a simple alternative to MULTICS. UNIX required less memory to run, and so could be deployed on systems which did not have enough memory to load MULTICS. This reduced the cost of the hardware, thereby making the freely distributed UNIX an attractive alternative.

The UNIX system was released in 1971 and introduced an innovative command line interface called the Shell. Ritchie (1978), states that the UNIX Shell was central to the success of UNIX. The Shell promoted generic text-based data interchange between utility programs. These utilities formed the larger part of the UNIX system. Each utility performed a single simple task, but could be combined to solve larger problems, this approach is called the Pipe and Filter design pattern. Utility programs embody the UNIX slogan “Do one thing well”. This principle kept the core of the system simple by providing most of its functionality through additional utilities. Lions (1976), lists the complete Kernel which contained only 9000 lines of code, given the sophistication of the system, this is an historic achievement in minimisation.

According to Ritchie (1974), UNIX makes no distinction between devices, programs, or secondary storage. “Everything is a file” is a UNIX philosophy which promotes generic data interchange, together with the pipe and filter design pattern. These innovations greatly simplified the complex task of programming for the end user. This made the UNIX system the most popular operating system of the 1970’s and early 1980’s.

1.4 Xerox Alto

The Xerox Alto was developed at the Palo Alto Research Centre (PARC). The PARC think tank invented many successful technologies still widely used today such as laser printing and Ethernet. However, PARC is best known for the development of the Graphical User Interface (GUI) used by the Alto Personal Computer, released in 1973. Other systems at this time

required technical expertise from their users. The Alto was a true Personal Computer, it was highly intuitive to operate and could fit next to a desk. This interface has become common in operating system design. Many features introduced by the Alto system are instantly recognisable today. The Alto redefined the way humans interact with computers. The topic of Graphical User Interface is revisited in Chapter 3, where it will be discussed in detail.

Despite its revolutionary user interface, the Alto was never widely adopted. According to Kay (1987), this was due to its retail price of up to $100,000. Given that the Alto was marketed as a Personal Computer, this cost kept the system out of reach of its intended market. History shows, that when choosing between human comfort and affordability, the market is willing to sacrifice comfort for cost. Just as the prohibitive cost of MULTICS created demand for UNIX in the business sector, so too did the unrealistically priced Alto leave room in the consumer market for the first true personal computer, the Apple.

1.5 Apple Computers

In the mid 1970's computer hardware became available to early adopters, and a community of electronic enthusiasts called the *Homebrew Computer Club* was born. In 1976, one of these enthusiasts, Steve Wozniak, developed a system called the Apple Computer 1, which used off the shelf components to interface to a television. The system was an instant success and led to the establishment of Apple Computers. The early Apple computers created much hype but were of little practical use to non-technical users. The marketing genius of Steve Jobs pushed the Apple in just the right direction to respond to the market demands. The personal computer market truly began in 1978, when Apple bought the rights to the Xerox Alto graphical user interface and used it in the Apple Lisa. The Macintosh was released shortly thereafter and offered a UNIX-like system called MacOS. This brought the elegant Shell of UNIX with the innovate interface of the Alto, into the minds and homes of consumers for the first time.

1.6 Windows

Throughout the 1980's Apple dominated the home computer market. However, as the market matured, consumers began to expect inter-operability between devices. By only supporting its own products, Apple failed to respond to this demand, and ultimately, this need was met instead by the IBM Personal Computer (PC). Apple did not support the PC platform, instead the PC was packaged with various forms of DOS and later, in 1985, with the first version of Windows.

To this day, Windows provides unparalleled support for peripheral devices, and has consistently been an early adopter of new technology. Windows users expect and receive plug and play compatibility between their system and off the shelf devices. This strategy has allowed Microsoft to maintain market dominance. Windows has become the most popular Operating System and the most successful software product line of all time, making Bill Gates one of the wealthiest people on earth.

1.7 Linux

Linux is a Kernel, and a UNIX clone, created by Linus Torvalds. The Linux system emphasizes freedom, both in terms of cost, and in liberties. Linux has always placed its users first and is the flagship product of the open source movement, distributing its source code free of charge. A community of technical users actively contribute to the development of Linux. The system is highly customizable and comes in many varieties called "Distros".

Due to its low cost, and lack of restrictions, Linux has been adopted in the industrial sector, and has seen success both in the server market, and in embedded systems. Linux is the basis of many forks including the Android OS system. According to Android Authority (2018), Android OS held an 80% share of the mobile market in 2017. Linux systems have continued to rise in popularity and have seen increased support from manufactures now offering official driver support.

1.8 Conclusions

Two factors have emerged as critical to the success of an operating system. The cost of early systems can have a negative effect on the adoption of a system, as was the case with MULTICS, the Alto, and to a lesser extent the Apple. Often in this sector innovative systems arrive decades before their time. The cost of owning tomorrow's technology today can place the technology out of reach of the public, resulting in the commercial failure of that product. Today's systems are affordable enough such that cost can largely be ignored.

The remaining factor deciding the success of an operating system is how well the system improves the usability of a computer. The general-purpose command line of UNIX, the intuitive Graphical User Interface of the Alto, and the universal hardware support of Windows, have shown that successful systems emphasize usability, and have consideration for their users. Future systems must continue to place the user first if they are to succeed in the market.

PART I

Research

Chapter 2 – The Kernel

Corbató (1962), and Stallings (2005), state that the purpose of an operating system is to run other programs. A key component of an operating system is the Kernel which interacts directly with the hardware. The Kernel is primarily concerned with making best use of the hardware, and promotes productivity, and inter-operability. The main components of a Kernel are; the Device Manager, Memory Manager, and Process Scheduler. Section 2.1 covers the topic of Device Management, introduces the components which make up a computer system, and outlines the mechanisms used to communicate with key devices. Section 2.2 outlines the role of a Memory Manager and investigates the algorithms which effectively manage this limited resource. Section 2.3 addresses the issue of Process Scheduling in a multitasking system.

2.1 Device Management

Device Management is central to the issue of inter-operability. A system with little support for devices will find it difficult to survive in today's plug and play market. Stallings (2005), describes an operating system as "an interface to the hardware". An operating system is expected to support a large variety of devices. The Kernel is responsible for communicating directly with the hardware. To achieve this goal, both the Kernel and the device must speak the same language. And so, the topic of Device Management is governed by protocols. Before discussing any of these protocols, it is necessary to introduce related terminology, and provide sufficient background to bring the reader up to speed. To this end, this section introduces the execution environment of the Intel 80386 processor running on a PC Compatible motherboard.

2.1.1 The PC Compatible Motherboard

The backbone of a Personal Computer (PC) is the motherboard, which connects the various components of a computer. Ron White (2015), states that the PC Compatible motherboard, with an 80386 CPU, is the dominant configuration in the world of Personal Computing. This hardware configuration has become the industry standard in the Personal Computer market. The term PC refers to a standard format introduced with the IBM Personal Computer in 1981. The PC was launched into a market which was plagued by hardware incompatibility, even within a single product line. The IBM PC came with the guarantee that hardware which worked on the original PC would also work with its' successors. BYTE Magazine (1984), criticized the Apple III for not following the example of the IBM PC.

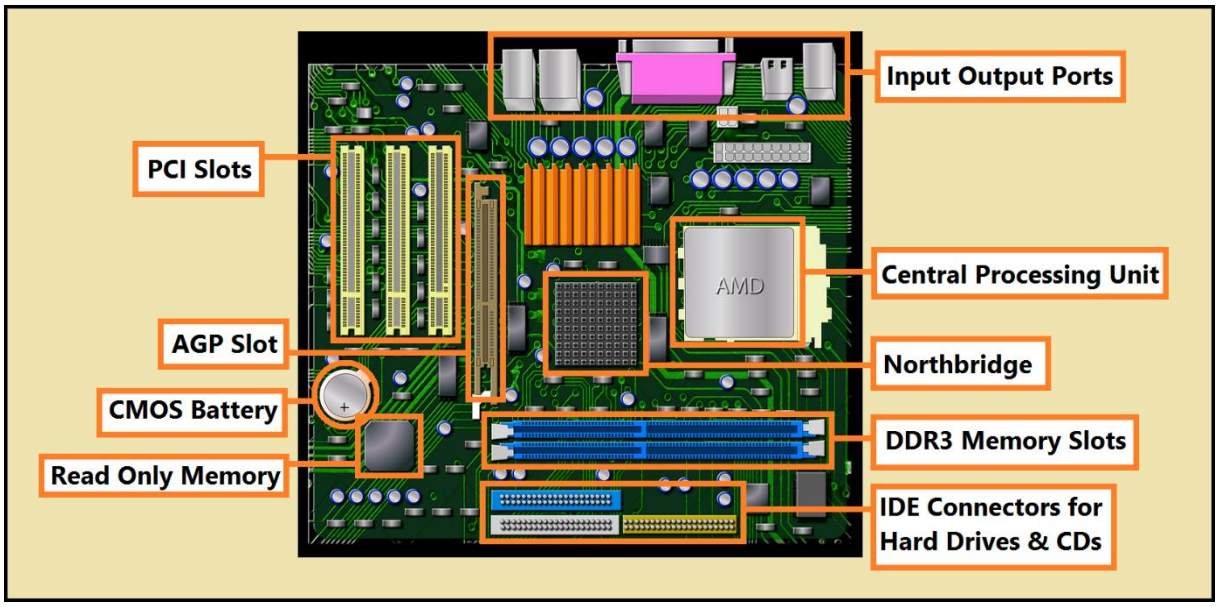


Figure 2.1: Modern PC Compatible motherboard, source: https://www.pctechguide.com/

IBM's approach to hardware compatibility made the PC hugely successful. The IBM PC became so popular that other manufacturers copied the design. These are called PC Compatibles. All PC Compatibles provide a Basic Input Output System (BIOS) compatible with the original IBM PC. Figure 2.1 illustrates a modern PC Compatible motherboard and highlights the most important components which make up a PC.

2.1.2 The Central Processor

Motherboards are designed to support a generation of processors. A generation is a collection of processor models which share the same form factor called a Socket. The Central Processing Unit (CPU or Processor) is the heart of the system. The role of the CPU is to execute programs. All other components exist only to support the CPU. The x86 architecture is the dominant architecture found on PC Compatible motherboards. The x86 is a family of processors derived from the 8086, which is a 16-bit processor introduced in 1978. The modern 64-bit processors are completely backward compatible with the original 8086. Intel and AMD are two brands competing in this market space. Both companies produce compatible CPUs that understand the same binary instruction set. Intel's (2018), Manuals Volume 3, details the mechanisms used to control the 80386 CPU, and to set up an effective environment in which to execute applications.

2.1.3 Device IO on the 80386 Processor

Intel (2018), states that the 386 supports both interrupts and polled input/output (PIO). Polling is a simple approach to device IO, in which the CPU constantly queries the device to check if data is present. This approach places the CPU into an idle state, while the device prepares the data. Using interrupts is preferred, as it avoids idle downtime, and allows the CPU to proceed with other work. When the device is ready, it will issue an interrupt signal to the CPU. This signal causes the CPU to automatically stop what it is doing, so it can service the device.

The Intel (2018), uses both Port IO (PIO), and Memory Mapped IO (MMIO) to communicate with devices. The IBM (1981, 1989, and 1990), PC Compatible BIOS reserves port mappings, and regions of memory, to accommodate communication between the CPU and devices. Each device has its own range of port addresses, and possibly memory addresses. Each address serves its own purpose, and the meaning of each address is determined by the protocol governing the device.

2.1.4 Random Access Memory

The second most important component in a computer system is the memory. This comes in the form of RAM, which stands for Random Access Memory. RAM stores both the data and the instructions that make up programs. This storage medium is volatile, once the power supply is removed the contents of RAM are lost.

2.1.5 Secondary Storage Devices

Secondary storage such as Hard Disk Drives (HDD), Solid State Drives (SDD) and Compact Discs (CDs) provide persistent data storage. These devices are slower than RAM but have the benefit of being non-volatile. These devices are connected to the motherboard through the Integrated Drive Electronics (IDE) connectors. IDE connectors first appeared in 1987 with the introduction of the IBM AT (Advanced Technology) PC. The IDE interface carries legacy support for obsolete technologies such as Floppy Disk Drives (FDD) and Magnetic Tapes. Secondary Storage Devices are governed by a series of standards, the names of which include the letters ATA (AT Attachment). Examples are SATA (Serial ATA), PATA (Parallel ATA), and ATAPI (ATA Packet Interface).

2.1.6 CMOS Battery and Real Time Clock

The Complementary Metal-Oxide Semiconductor (CMOS) battery acts as a power source when the main power supply is shut off. The battery provides power to the Real Time Clock (RTC), so it can continue to keep track of time when main power is removed. The CMOS battery also powers the CMOS memory. This memory is used by the Firmware to persist essential configuration settings needed during a process called the Power-On-Self-Test (POST). The POST initializes the system when main power is first turned on. CMOS memory is accessible to the operating system and is used to configure the Real Time Clock.

2.1.7 The Chipset

All these components are connected to the CPU through the System Bus, which is controlled by the Chipset. The Chipset is the Memory Controller Hub and is divided into two parts: The Northbridge and Southbridge. Together they are responsible for the transfer of signals and data throughout the system. The role of the Southbridge is to manage slower bus lines such as the IDE bus. The Northbridge is responsible for high speed transfers such as RAM. Only the Northbridge is directly connected to the CPU through a bus called the Front Side Bus (FSB). The FSB is a critical bottleneck in the PC motherboard. This bottleneck is illustrated in figure 2.2, showing that all bus paths to the CPU must pass through the FSB.

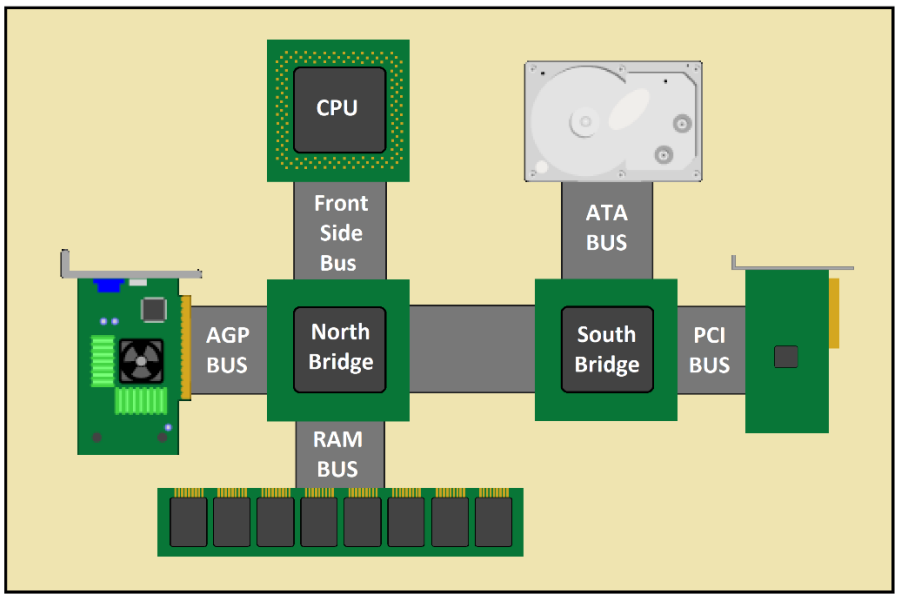


Figure 2.2: The System Bus in a PC Compatible

2.1.8 Firmware / BIOS

Read Only Memory (ROM) contains the system Firmware. Firmware is software developed by the motherboard manufacturer. Firmware is intimately coupled to the hardware, and will only work on a specific hardware configuration, unlike most software which is portable across many hardware configurations. The BIOS is an example of Firmware, which is considered legacy, The Unified Extensible Firmware Interface (UEFI) is the modern successor to the PC BIOS.

2.1.9 Video Output

VESA (2018), is a set of video modes which range from monochrome (1-bit), to full 32-bit RGB. The pixel resolutions start from 320x200 and go up to 4K displays. VESA (2018), uses memory mapped IO, where regions of main memory are rewired to video memory. This memory is called the frame buffer and its contents represents pixels. The frame buffer is accessed through normal memory read and write operations. The chipset detects addresses in this range and dispatches the read or write operations to the video controller instead of main memory. The VESA display adapter can be treated just like normal memory.

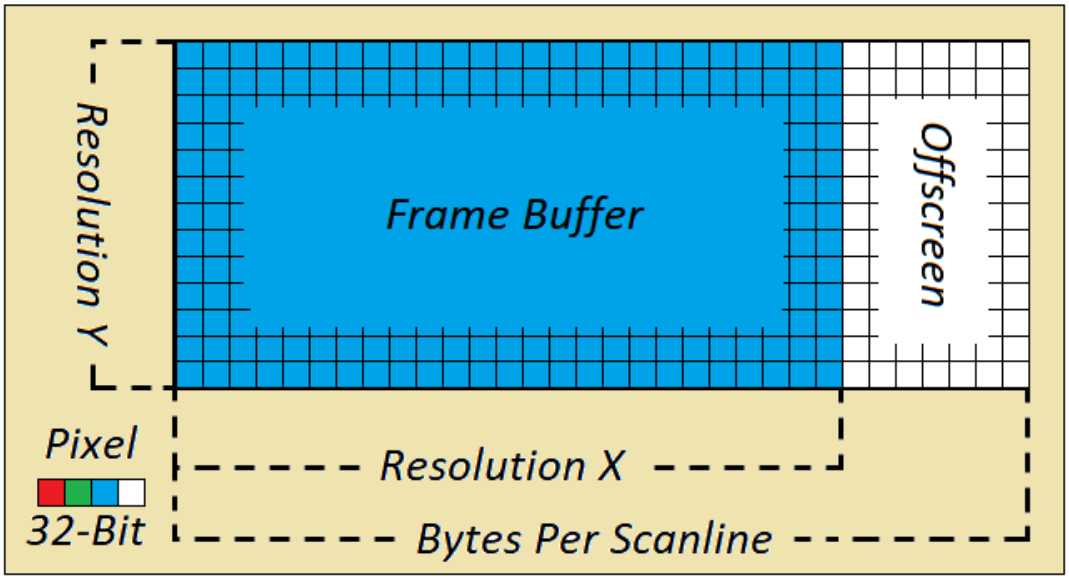


Figure 2.3: VESA Video Mode Information, showing the main features of interest.

The address of the memory mapped frame buffer is not fixed, to acquire this information the video adapter must be queried. This is achieved through BIOS service calls which provide the capabilities of the adapter, and the parameters of the video mode. This information includes, the pixel resolution, both horizontal and vertical, the bit depth, the physical address of the frame buffer, and the bytes per scanline. Figure 2.3 illustrates the information required to plot pixels: the colour depth, the bytes per scanline, and the resolution. The full VESA specification is available at: https://www.vesa.org/

2.1.10 Keyboard

The keyboard interface of the PS/2 was based on that of the earlier XT, and AT models. According to IBM (1989, and 1990), this interface assigns numbers to each key, called scan-codes. These are not ASCII codes, they are keyboard scan-codes. The operating system is responsible for mapping these to the appropriate character encoding. Figure 2.4 displays the scan-codes found on the standard 101-key keyboard layout.

IBM (1989, and 1990), states that the PS/2 interface is a packet-based protocol. A normal packet is a single byte and encodes a 7-bit scan code which corresponds to a specific key, and a single bit which indicates if the key has been pressed (clear) or released (set). Figure 2.4 shows the encoded packets when the "A" key has been pressed, and then released.

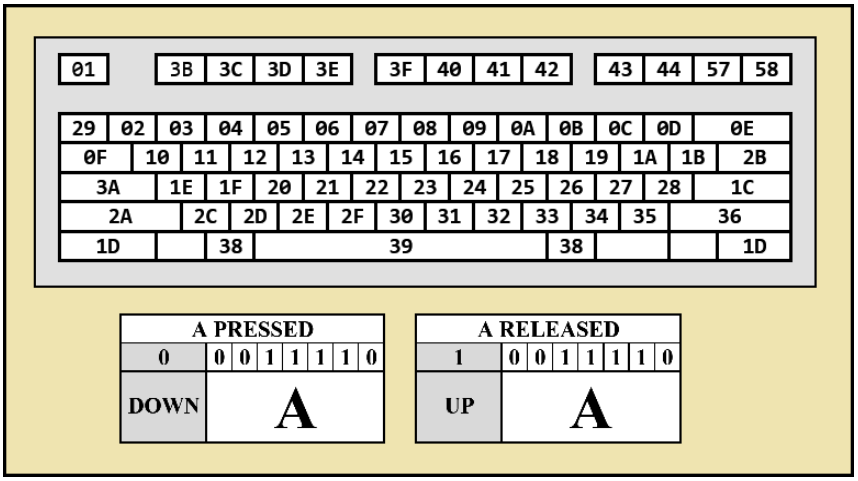


Figure 2.4: Top: the standard 101-key layout, illustrating the scan-code mappings, Bottom: Both the key pressed, and key released PS/2 keyboard packet encodings.

The PS/2 keyboard is not memory mapped and is accessed through port IO. IBM (1989, and 1990) reserves port 0x64 (PS/2 status), and port 0x60 (PS/2 data) for this purpose. The operating system can poll port 0x64 to test if bit-1 is set. This indicates that a keystroke packet is available, which can be read from port 0x60. The keyboard can be mapped to an interrupt, which avoids the need to poll the status port.

2.1.11 Mouse

The mouse interface of the PS/2 shares the same ports as that of the keyboard. IBM (1989, and 1990) reserved bit-0 of port 0x64 (PS/2 Status), to indicate that mouse input is available on port 0x60 (PS/2 data). A PS/2 mouse packet is made up of three bytes. Figure 2.5 shows the information contained in a mouse packet. These packets contain button states, together with x, y movement deltas. These x, y deltas represent the movement applied to the mouse relative to the last reported position. This sequence of movements must be recorded to keep track of the absolute position. The operating system is responsible for tracking these movements and converting them to absolute co-ordinates.

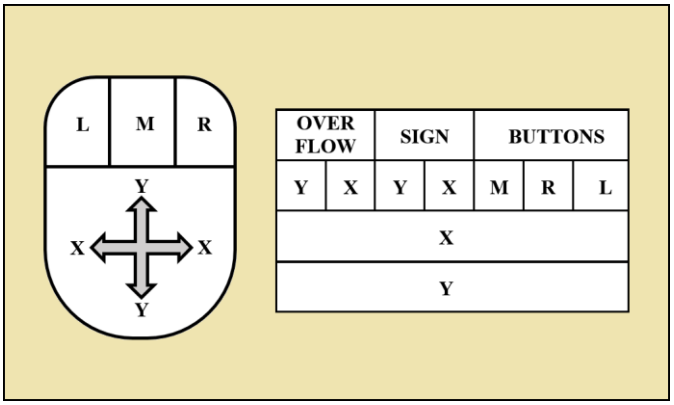


Figure 2.5: PS/2 Mouse Packet Protocol.

IBM (1990), states that the PS/2 controller is much slower than the CPU, and so it will trigger an interrupt, to signal the arrival of each byte seperately. This necessitates a buffering system where the driver will collect groups of three bytes, then process them as a unit. The first byte contains button states, sign bits, and overflow bits. The next byte is the low order 8-bit x co-ordinate, followed by the same for the y co-ordinate. The units of movement reported by the mouse have no corelation to screen space. When high resolution video modes are in effect the mouse movement covers less screen space. This is overcome by applying a scalar which exagerates movement. Further refinements can control the scalar to simulate the effects of inertia and acceleration. This enhances the interface giving a more natural feel.

2.1.12 The Clock

The clock is used in a multi-tasking system to synchronize task switching. The 80386 processor provides multiple timers that can fill this role. The Programable Interval Timer (PIT) is one option. The PIT serves a secondary purpose of driving the speaker. Because of this purpose the PIT operates at 1,193,180 Hz which is suitable for generating musical notes. Modern systems provide a High Precision Event Timer (HPET) which has features that make it ideal for process scheduling, such as a configurable frequency, and count down trigger events. but is not guaranteed to be present. Finally, the Real Time Clock (RTC) has the single purpose of recording the system time. This section focuses on the RTC due to its simplicity. The RTC generates interrupts at a monotonic rate. The interval is controlled by setting the value of a divider. The divider splits up the base frequency to produce lower frequencies. Available frequencies range from 2,048 Hz to 32,768 Hz.

The RTC is controlled by CMOS memory. This memory contains 256 bytes of data. The structure of this data is fixed with each byte serving its own purpose. To modify CMOS memory the system must request read and write operations. This is achieved through port IO. IBM (1989) reserves ports 0x70 and 0x71 for this purpose. Port 0x70 is used to specify the offset within the CMOS address space, and port 0x71 is used to transmit data. Setting up the RTC requires manipulating the byte at offset 0x0B (The RTC B register). This byte uses bits 0 to 3 for the divisor, and bit 6 is used for generating interrupts.

When interrupts are enabled, the RTC will produce a series of timer interrupts which can be used to drive the process scheduler. IBM (1989) states that when an interrupt is generated the system must flush the RTC C register. If the system fails to do so, then the RTC will not generate further interrupts. Flushing the RTC C register is achieved by reading the contents of offset 0x0C in CMOS memory.

2.2 Memory Management

In this section the role of a memory manager is defined. A distinction is drawn between physical memory and virtual memory, as the concept of a frame, and that of a page are introduced. After outlining the basic concepts, this section turns its attention to the efficient management of memory in the context of a multi-tasking system.

2.2.1 The Role of a Memory Manager

Memory management is the treatment of memory as a resource to be allocated and to be shared between independent processes. Stallings (2005), considers memory management as one of the most important and complex activities performed by an operating system. He adds that to maximize throughput, the system should maintain as many processes in main memory as possible. The challenge of a memory manager is to meet the needs of many processes with the limited capacity of available memory. Corbató (1962), presents a multi-tasking system which later developed into the MULTICS operating system. Corbató (1962), had already recognised the following requirements for a multi-tasking system:

1 That each task must be protected against the interference of others.

2 That tasks must support relocation.

3 That no task should be allowed to monopolise a resource to the detriment of others.

Stallings (2005) summarizes these requirements with the terms: Protection, Relocation, and Sharing. Protection is required as the risk of one task overwriting the personal space of another would be catastrophic. Sharing memory as a resource is the goal of a memory manager. Relocation is necessary as unused memory must be swapped out to disk so that it can be reclaimed and repurposed. When that memory is needed in future it would be highly restrictive if the original address must be used. Addressing these issues is the topic of this section.

2.2.2 Physical Memory vs Virtual Memory

Physical memory refers to the physical capacity of memory available to a computer system. In this memory model, there is a one-to-one correspondence between the physical address of a byte, and the location of that byte in main memory. Physical memory is made up of bytes which are grouped into frames. A frame of memory is treated as a unit when managing physical memory. Intel (2018) states that, Physical memory provides no protection, the processor has unrestricted access is to the global address space. Only the Kernel should have direct access to the entire physical address space.

Virtual memory is a memory model which uses logical addresses to refer to physical addresses. Intel (2018) states that, logical addresses are translated to physical addresses by the Translate Lookaside Buffer (TLB). This remapping between logical addresses and physical addresses is performed on a frame by frame basis, mapping virtual pages to physical frames. These remapping's are contained in a data structure called a page directory. There can be any number of page directories defined, but only one can be active at any time. Each of these page directories defines its own logical address space, which spans the entire addressable range, which is 4 GB on 32-bit processors.

The TLB facilitates both protection and relocation in systems which use paging. Virtual memory provides relocation, since the logical address of a page may differ from the physical address of its associated frame. Intel (2018) states that, virtual memory is a memory model in which each running process is given its own isolated address space. This provides protection, since each task can access its own pages, but not the pages of its neighbours. Only those pages which have been assigned to a physical frame can be dereferenced. Intel (2018), guarantees that accessing pages which have no associated frames results in a page fault. By ensuring that there is no overlap between the page directories of each running process, the Kernel can ensure that each task is protected from overwrite by malicious or negligent neighbours.

2.2.3 Partitioning

The heart of memory management is the problem of effectively allocating memory. The simplest scheme for managing memory allocation is called partitioning. In this scheme available memory is divided into blocks called partitions. Each partition is allocated to a specific process. When the process completes, then the partition is made available for use by other processes. If all blocks are equal in size, then it is a Fixed Partition scheme, if the blocks can differ in size, then it is a Dynamic Partition scheme. This simple approach allows for efficient algorithms, though it does have weaknesses.

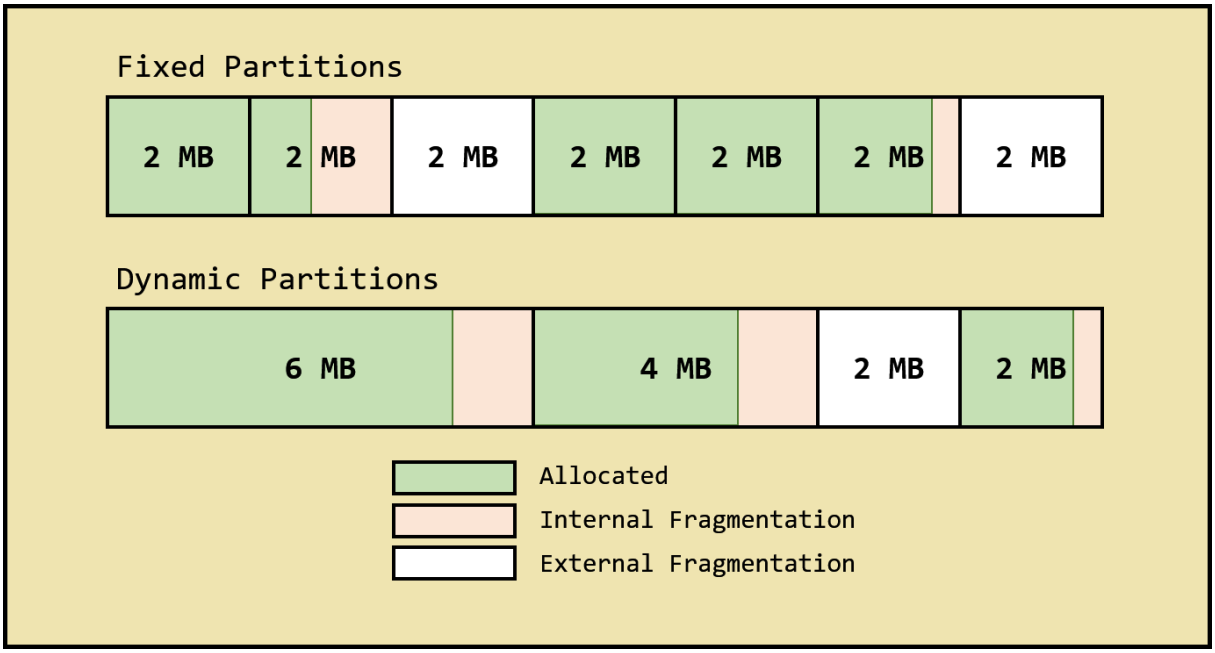


Figure 2.6: Partitioned memory highlighting the effect of fragmentation.

Stallings (2005), highlights that partitioning schemes suffer from fragmentation, making it unsuitable for modern operating systems. Fragmentation is illustrated in figure 2.6. This occurs when memory is unused either because the allocation unit is larger than the requested size, resulting in unused slack space called internal fragmentation, or when isolated fragments of available memory exist, but are too small to meet any request. This latter form is called External fragmentation. These isolated partitions are essentially wasted. Both cases must be avoided by the frame allocator. Because of these limitations, partitioning is not used by modern systems.

2.2.4 Best Fit, First Fit and Next Fit Algorithms

When allocating storage for a task, it would be best to select a partition which is just large enough to contain the task, but no larger. This minimizes wasted memory due to internal fragmentation. This is the idea behind the Best Fit algorithm. This approach requires scanning all available partitions, then choosing that which best fits the size of the task.

The Best Fit algorithm eliminates external fragmentation by always selecting the optimal partition. However, both Tanenbaum (1987), and Stallings (2005), state that this is a slow process. It is faster to simply locate the first partition which is sufficient for the size of the task, this is the First Fit algorithm. The First Fit algorithm produces external fragmentation but has the benefit of superior performance. According to Lions (1976), the First Fit algorithm was used to great effect in early versions of UNIX and gave satisfactory results.

Stallings (2005), notes the drawback to the First Fit algorithm is that the scanning always begins from the start of memory. The memory manager must scan through previously allocated partitions before it locates the first partition which fits the request. A better approach is to record the location of the previous allocation and to begin scanning from that point forward. This approach is the Next Fit algorithm, which results in much faster allocation.

When analysing the efficiency of these partitioning strategies it should be noted that the memory map requires a collection of partition descriptors, of which there are N elements. Scanning these elements is performed iteratively, and in the worst case every element must be visited. Therefor all operations are linear and carry a running cost on the order of $O(n)$.

Fixed partitioning strategies can be effective in systems which use paging. By applying this strategy to allocate physical memory, the Kernel can stitch together a plurality of isolated frames to form larger partitions. The page directory can be manipulated such that pages appear to be contiguous, even though the frames may be scattered and isolated.

2.2.5 The Buddy System

Knuth (1968) presents an efficient memory allocation algorithm called the Buddy System. The buddy system is widely used, and its continued application gave rise to the term “Heap” as a synonym for memory allocation, since it is commonly implemented by the heap data structure.

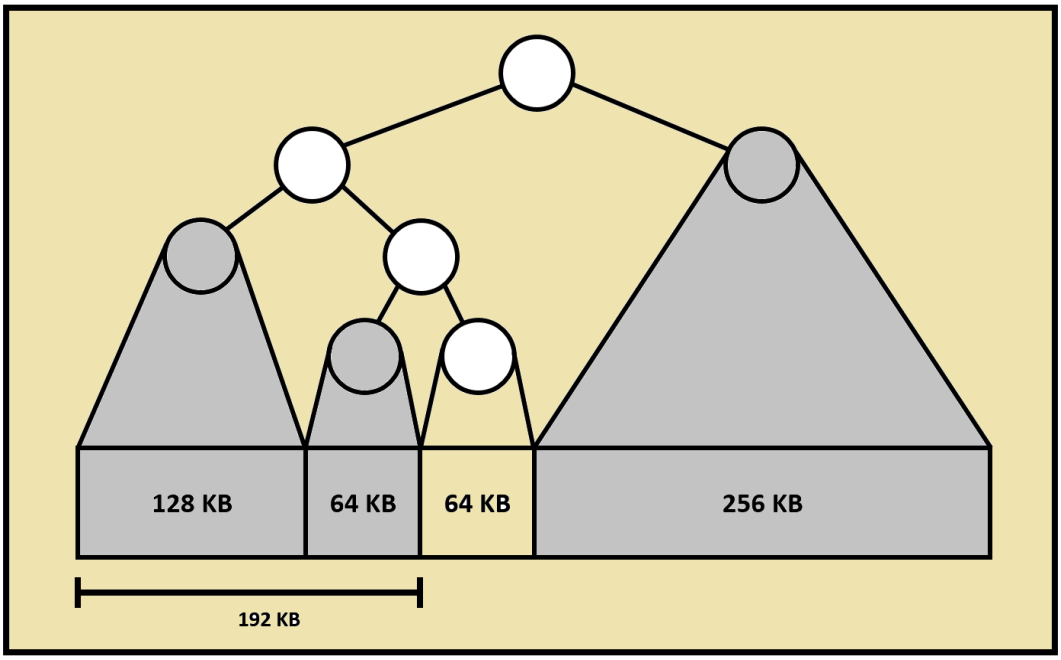


Figure 2.7: The buddy system approach to allocating memory.

The Buddy System represents the global address space as a binary tree. In this system the allocator starts at the root whose footprint is the entire address space. By successively splitting regions in half, eventually a block is obtained which fits the needs of a request. Knuth (1968), shows that by combining adjacent blocks from different subtrees, the allocator can provide allocations which are not a power of two. Figure 2.7 demonstrates that a combination of 128 KB and 64 KB blocks, when taken together meet a request for 192 KB.

When analysing the buddy system, it is can be seen that, given its basis on a binary tree, all operations carry a complexity of: $O(logN)$, in the worst case. The buddy system was the earliest example of an allocation strategy which achieves logarithmic complexity. This makes the buddy system the strategy of choice for managing virtual memory allocations.

2.2.6 Page Replacement Strategies (Disk Swapping)

Physical memory is a finite resource, eventually the time will come when every available frame has been assigned to some task. At this point some page will need to be reclaimed and reassigned. This is achieved by adopting a page replacement strategy, more commonly known as Disk Swapping since the page is retired to disk so that it can later be restored.

László Bélády (1969), found that as the total number of pages increases, so too does the number of page faults. This effect is known as the Bélády Anomaly and is counter-intuitive as it is assumed that an abundance of pages should alleviate the pressure. In fact, an abundance of allocated pages implies that the systems memory is being maximized. Allocating further pages has the effect of applying increased pressure on a limited resource. Bélády (1969), found that the FIFO system was a poor choice in this regard, with long running tasks suffering most. For this reason, the FIFO system is no longer considered a viable solution.

2.2.7 Demand Paging (Lazy Loading)

Aho (1970), found that the demand paging algorithm, or lazy loading, is the optimal paging policy carrying the minimum possible cost, measured in page fault occurrences. Demand paging achieves optimal fairness by following one simple rule: Do not allocate any physical memory unless it is used. For instance, when spawning a new process, the Kernel may neglect to allocate the required memory. The Kernel can rely on the fact that once the process begins execution it will generate a page fault. Only then will a portion of the process be mapped into memory, such that execution may proceed. After an initial flurry of page faults, the algorithm converges on the minimum working set of pages sufficient for the needs of the process. All modern systems use demand paging, since it allows running more tasks then the total memory can accommodate. Aho (1970) states that allocating pages on demand is optimal but remains reserved on the matter of choosing the page to be replaced.

2.2.8 Least Recently Used

Bélády (1966), observed that policies which attempt to minimize the number of required page replacements perform more poorly than expected. Bélády (1966) argues that a displacement in time, rather than a count of replacements, is a superior criterion. Bélády (1966) outlines the optimal page replacement algorithm, called “Clairvoyant Replacement”. This is an example of a God Algorithm which cannot exist. It requires knowing in advance which page will be needed at the farthest point in the future, and then choosing such a page. Bélády (1966), uses this rationale to develop a good approximation; the Least Recently Used (LRU) algorithm.

Bélády (1966) assumes that a page which has not been accessed recently will likely not be needed soon. This information can be gathered, since Intel (2018) declares that the hardware will indicate which pages have been accessed by setting a bit in the page directory. The job of the scanner is to find unused pages by toggling these bits. Pages which have not been accessed since the last visit are reclaimed by adding them to a list of available pages. By ordering this list according to timestamps, it is possible to estimate how recently a page has been accessed. The page allocator then selects the optimal page from this list.

As was the case with the Best Fit algorithm, LRU shows that making optimal choices can be an expensive mistake. Stallings (2005), argues that the LRU approach is difficult to implement efficiently. The algorithm requires a running cost on the order of $O(n)$ simply to identify unused pages. LRU also requires memory on the order or $O(n)$ for book keeping such as time stamp recording. Ignoring the issue of maintaining a sorted list, the LRU approach carries a linear cost both in terms of memory footprint and process time. This amounts to significant overhead. Stallings (2005), refutes this approach, showing that the benefits of this policy are insufficient to negate these running costs. For this reason, Stallings (2005) considers Clock replacement a superior choice.

2.2.9 Clock Replacement

Clock replacement approximates the LRU policy and is analogous to the First Fit improvement to the Best Fit algorithm. Clock does not try to find the optimal page and is satisfied to select any suitable page. This simplifies the role of the scanner, giving it the option of an early out, and allowing the scanner to be invoked only on demand (in the event of a page fault).

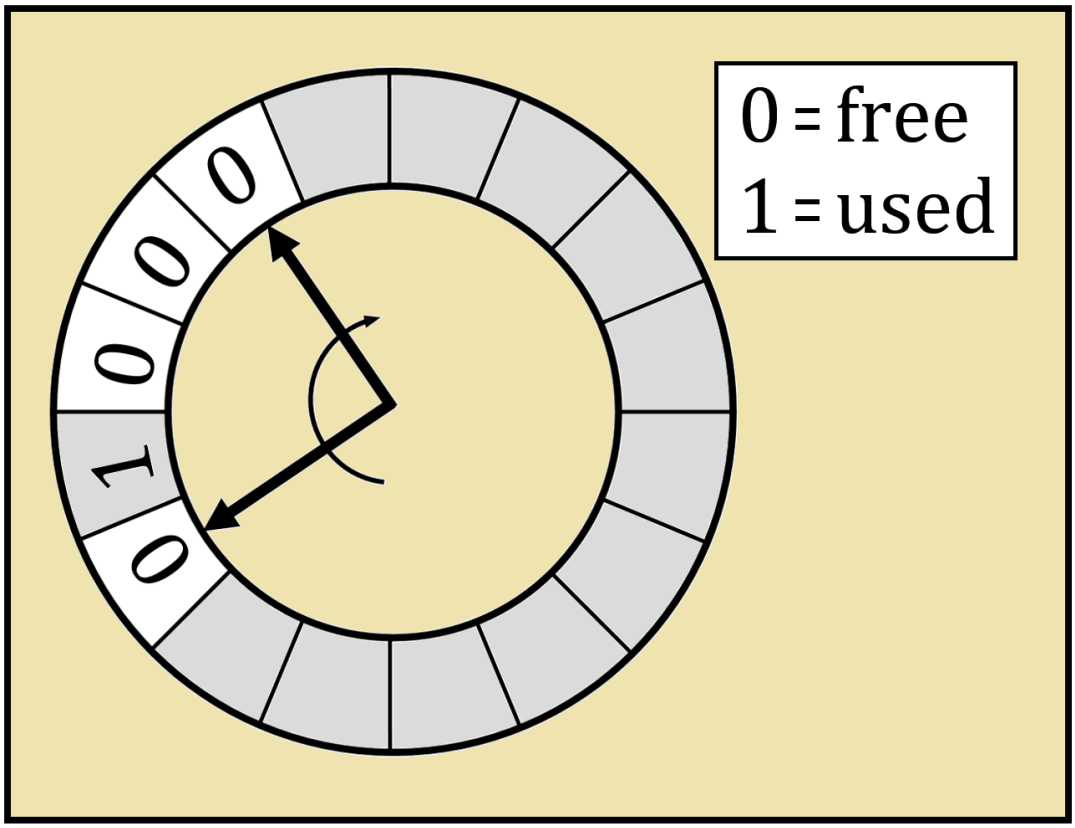


Figure 2.8: Second chance clock replacement policy.

Clock replacement treats the page directory as a circular buffer. The only overhead required by Clock is a single pointer called the clock hand. The clock hand cycles through the page directory in clockwise order, toggling the accessed bits as it proceeds. Once the first suitable page is encountered, it is immediately reclaimed. These refinements eliminate the linear running costs and perform on the order of $O(C)$ on average. Given that constant time overhead is optimal, Stallings (2005), considers Clock replacement as the most simple replacement policy which can be realised efficiently.

The Second Chance Clock algorithm is an improvement over Clock which minimizes the number of page faults incurred. The rationale behind this approach is to give each page a second chance before being reclaimed. This is achieved through the addition of a second hand which follows the leading hand around the Page Directory. The front hand is solely concerned with marking pages as unused, while the back hand reclaims those pages. Figure 2.8 shows the basic components of the Second Chance Clock replacement algorithm; where the front hand can be seen actively marking pages as unused, while the back hand reclaims those pages which still meet this requirement.

The Second Chance Clock replacement policy requires only two pointers, which is a significant reduction in memory footprint over the LRU policy. Clock only considers a small window of pages directory entries when identifying unused pages, this can be completed in significantly less CPU time compared to the LRU policy. Further, the Second Chance enhancement to the basic Clock approach, significantly reduces page fault events. Due to the minimal overhead, and close approximation to the least recently used policy, the Second Chance Clock algorithm is the replacement policy used by the Linux Kernel.

2.3 Process Scheduling

Stallings (2005), states that the goal of an operating system is to execute other programs called applications. This is achieved by the Process-Scheduler, whose role is to manage time as a resource. Time is a commodity which must be shared fairly amongst tasks. In one sense time is a boundless resource of infinite supply, whilst in another sense, time is in short supply when a user is waiting for feedback. In a multi-tasking system each running task increases time pressure on the rest of the system. Managing time is a major consideration in operating system design. This section outlines the mechanisms used by the process scheduler to achieve its goal.

2.3.1 Overview of the Process Scheduler

Stallings (2005) outlines the basic components of a scheduling system, such as the one depicted in figure 2.9. These systems have a ready queue and a blocked queue. Part of the system called the High-Level Scheduler (HLS), is responsible for admitting processes to the system. The HLS performs the strategic process scheduling algorithm. Another part of the system, called the Low-Level Scheduler (LLS), is responsible for taking processes from the ready queue and moving them into a running state. This allows other processes in the ready queue to advance.

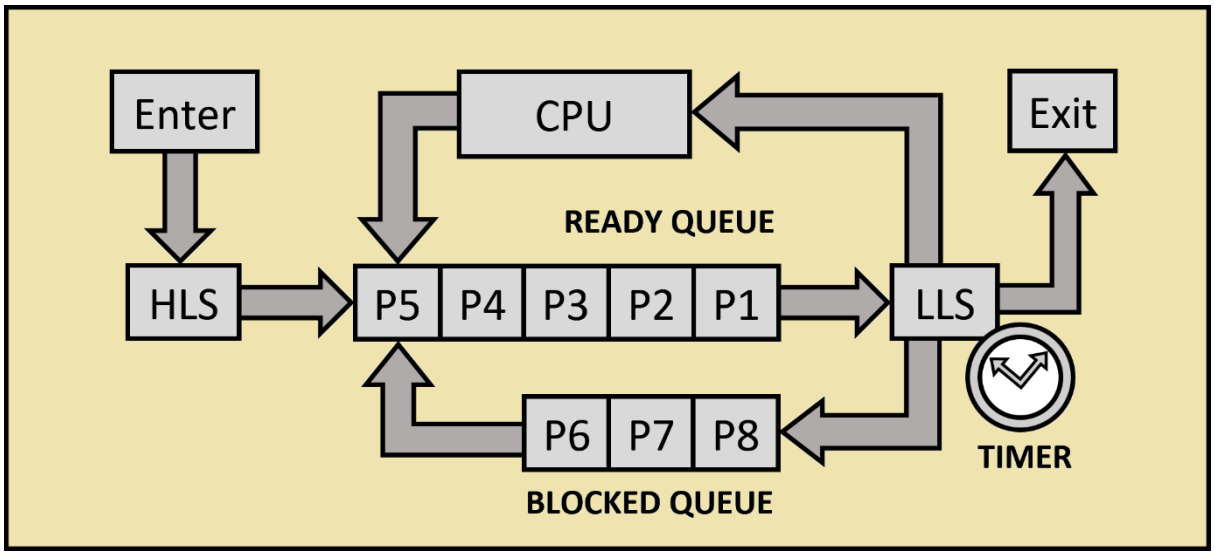


Figure 2.9: The main components of a process scheduler. The High-Level Scheduler (HLS) is the strategic scheduling, The Low-Level Scheduler (LLS) performs Context Switching.

When a process has completed, it can exit the system. If a process cannot continue, because it is waiting for input or output, it is moved by the LLS into the blocked queue. When a blocked process gets the services that it was waiting for, it can re-enter the ready queue. In a Round Robin scheduling system, each process gets a time slice in the running state, after which it is placed at the end of the ready queue. Stallings (2005), shows that these process state transitions can be realised by a Finite State Automata giving rise to the graph depicted in figure 2.10.

A timer, analogous to an alarm clock, will indicate to the LLS when a processes time slice completes. At this point the LLS must save the state of the process, so that it can later be resumed. To do this, the contents of the CPU registers are saved to the Process Control Block (PCB). Each process has its own Process Control Block. When the next process in line enters the running state, its previous state is restored from the PCB. This is called a Context Switch. Context switching is the most expensive operation performed by the Kernel. Therefor the length of a time slice must be chosen such that it outweighs the overhead of the scheduling system. When this is the case, then it is entirely possible for a modern CPU to perform thousands of Context Switches per second.

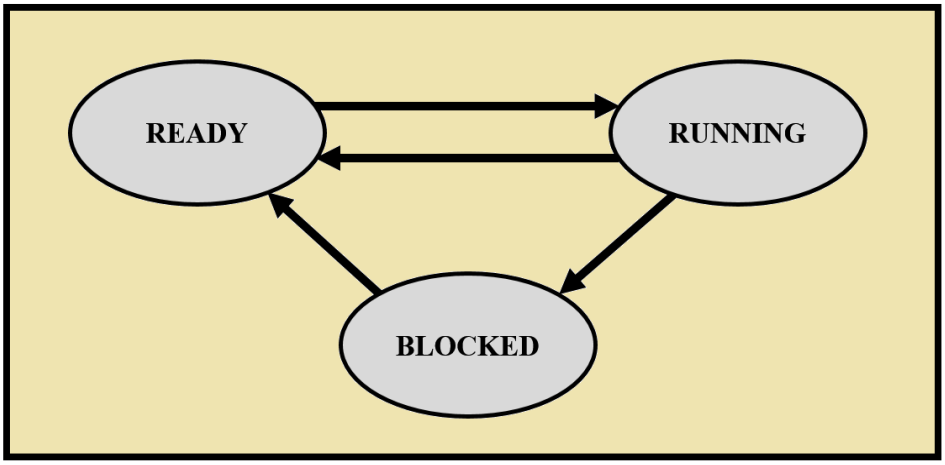


Figure 2.10: Process State Transition Diagram

2.3.2 Context Switching

Intel (2018), defines mechanisms which can achieve Context Switching. The 80386 processor requires a data structure called the Task State Segment (TSS). The TSS is used by the processor to save the state of the CPU registers. Intel (2018) requires that each execution core shall have its own TSS. The Scheduling System must ensure that the TSS always remains valid, since an interrupt can occur at any time and will cause a Context Switch.

The TSS contains four stack pointers corresponding to four rings of protection. Ring 0 is reserved for kernel mode execution and offers unrestricted access to the system. Ring 3 is used by applications who cannot access privileged instructions, special CPU registers, or physical memory. Operating systems only make use of these two rings since Rings 1, and 2 are specific to Intel processors. When the CPU is operating in Ring 3 the registers contain state appropriate for a user mode application. This state is saved automatically to the Ring 3 stack as defined by the TSS. The processor jumps to Ring 0, changing its mode of execution and disabling protection. Changing rings of protection is the most expensive operation of a Context Switch and cannot be avoided.

When the CPU enters Ring 0 the system has unrestricted access to the contents of the TSS. By manipulating the Ring 3 stack pointer the LLS can direct the CPU to restore the state of a different user mode application then was previously executing. This is called a Stack Switch and is the most common means of implementing a Context Switch. Intel (2018) also provides hardware support for achieving this. Time can be saved by being selective when saving and loading state. For this reason, Stack Switching is preferred since it gives fine grained control to the LLS.

2.3.3 Queueing Theory

Managing time in a Scheduling System is equivalent to managing the contents of the ready queue. This is the responsibility of the High-Level Scheduler. Stallings (2005) remarks that the goal of the HLS is to ensure that time is divided fairly between all running tasks. This requires defining criterion under which fairness can be judged. To achieve this the HLS can employ queueing theory. Queueing theory is a branch of statistical analysis which studies waiting lines, and the behaviour of a system in which customers must queue for a service. Queueing theory allows a manager to gather and analyse statistics such as average waiting time and throughput. Little (1960), shows that the number of customers in the system changes over time and is a

function of the arrival rate, and the average waiting time. This is called Little's Law and can be obtained by the following formula:

$$customers = arrivals \times waiting\ time$$

By transposing Little's Law, one can obtain the throughput (response time) of the system:

$$throughput = \frac{customers}{waiting\ time}$$

By applying queueing theory when admitting tasks into the system, the HLS can gather the metrics required to make informed decisions. The HLS must choose to employ a scheduling policy, and there are many options available including: First Come First Served (FCFS), Round Robin (RR), Shortest Process Next (SPN), Shortest Remaining Time Next (SRTN).

2.3.4 Scheduling Algorithms

First Come First Served is a FIFO (First-In First-Out) system. In this system each task executes in the order of arrival, and each task executes to completion without interruption. When the system can interrupt tasks, then it is said to be pre-emptive. Round Robin is a form of FCFS which is pre-emptive and is the simplest multi-tasking policy. Corbató (1962), employed this policy for MULTICS.

Shortest Process Next and Shortest Remaining Time Next are cases which emphasize deadlines and throughput. Under these systems applications may declare their total required running time. When the system cannot rely on honest co-operation from the running tasks, then these metrics must be taken to be averages according to Little's Law. This approach allows a scheduling system to be driven by deadlines and is dominant in Real Time Systems. Stallings (2005) notes that these systems favour short running tasks, but risk starving important processes which take longer to complete. This may be compensated for by the improvement in throughput, which empties the ready queue at a faster rate.

2.3.5 Priority Queues

Priority queues can be used in addition to a Scheduling Algorithm. In a priority-based system each running task is assigned a priority which indicates how important the task is. Tasks which have a higher priority advance through the ready queue at an accelerated pace. Stalling (2005) outlines such a system whereby each level of priority has its own queue. This is illustrated by figure 2.11 in which multiple priority queues are used to populate the final ready queue. Under this system tasks which have a higher priority are selected more frequently. This system avoids starvation by ensuring that even low priority tasks will eventually be scheduled.

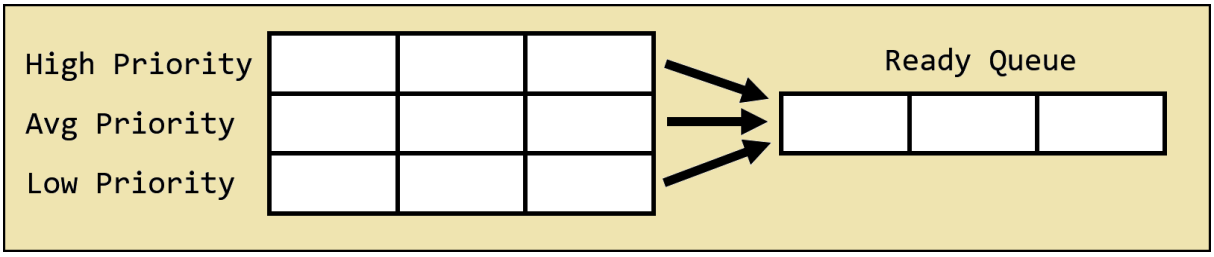


Figure 2.11: Priority Based Ready Queues.

2.4 Conclusions

Device Management is governed by a series of industry standard protocols. This chapter presented protocols for communicating with user-oriented devices: the keyboard, mouse, VESA Video display adapter, and Real Time Clock. The analysis of Memory Management has provided some surprising results: That virtual memory is essentially an unlimited resource which has completely solved the problems of process isolation and relocation in multi-tasking systems. Demand paging was found to be optimal in converging on the minimal working set of a running process. That the Second Chance Clock replacement policy can be used to achieve near optimal page replacement. Finally, the role of the Process Scheduling was defined, and the Round Robin algorithm was presented in detail.

Chapter 3 – The Shell

Thompson (1974) defines the Shell as a command line interpreter, it reads user typed commands and interprets them as requests to execute programs. Each command is terminated by a newline character and can contain references to multiple programs which together perform a unit of work. These programs form a pipeline through which data flows. Figure 3.1 illustrates a typical command line request and its division of labour among sub tasks. In this example the keyboard provides a stream of data which passes through two programs to produced output destined for the screen.

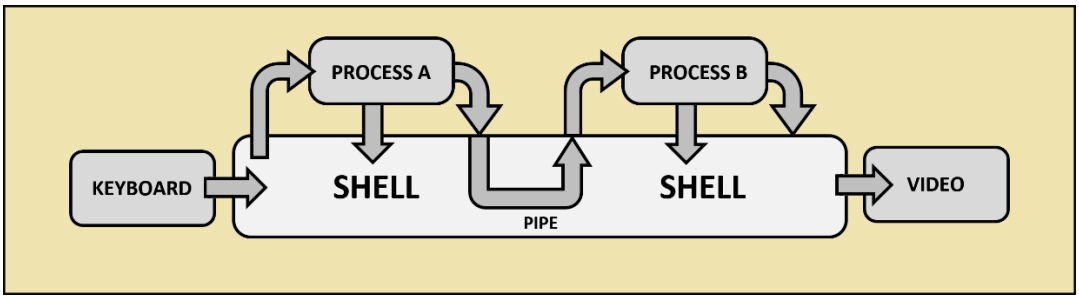


Figure 3.1: The UNIX Shell pipeline.

The Shell has evolved over time but remains the single point of contact through which a human interacts with the system. The Graphical User Interface (GUI) sits on top of the Shell and converts mouse clicks to commands. This chapter focuses of human interaction. The usefulness of the system is discussed with an emphasis on those features which have the greatest impact on productivity. Section 3.1 defines the role of the Command Line Interface (CLI), and why it remains an essential component in modern systems. Section 3.2 highlights the role of inter-process communication (IPC) in a system of interconnected parts. Section 3.3 investigates the Graphical User Interface (GUI), highlighting the benefits it provides. Finally, an approach is presented which implements a stacked windowing GUI system which minimizes latency.

3.1 Command Line Interface

The Command-Line Interface (CLI) was brought to its full potential with the introduction of the UNIX system in the early 1970's. The UNIX system of generic pipelining created a new programming paradigm which has become standard. This section outlines the approach which made UNIX the most popular operating system of its time.

Ritchie (1974), promotes the use of generic text-based streams. Apart from executable files, which must be binary, the UNIX system is averse to the use of binary formats. All processing is performed on textual data. This has the effect of promoting inter-operability between utility programs. Streams were exploited to great effect by the UNIX Shell pipeline. UNIX style pipes allow redirection of data through a chain of utilities. These utilities act as building blocks which can be pieced together to solve more complex problems. Piping allows users to create their own solutions to problems, without having to learn how to write programs.

UNIX files are simply a sequence of bytes, the system hides details such as the size of disk sectors. Ritchie (1978), notes that this approach was extended to include programs. This allows UNIX to treat everything as a file. No distinction is made between devices and programs. In this system data is homogenous. All data transfers and synchronization are handled by the same piping system, which maintains predictable behaviour.

Ritchie (1978), considers pipes as central to the effectiveness of a CLI. The ability to route the output of one program to the input of another creates endless opportunity to solve new tasks from existing utilities. Programs which perform simple tasks can be co-ordinated by the user to solve larger tasks, without needing to write new programs. Since the Shell too operates on text, any sequence of commands can be written to files called scripts, which can later be executed by the Shell as though the commands came from the keyboard. This is made possible by the generic nature of textual data, and the concept of piping.

3.1.1 Command Line Interpreter

The main responsibility of the Shell is to act as an interface between the user and the system. Thompson (1974) states that the Shell is a command line interpreter, reading lines typed into the keyboard and interpreting them as requests to execute programs. These commands are constructed from a few simple rules. A command consists of a program and a list of arguments:

```
program.exe arg1 arg2 arg3
```

The Shell separates this list and uses them as inputs to the specified program. When multiple programs appear in a single command it is necessary to define how the pipeline should be formed. The Shell reserves four special characters for this purpose:

```
<   >   |   ;
```

When multiple programs are executed in sequence, but not interconnected, then the commands can be entered on different lines, or a semi-colon may be used as a separator:

```
first.exe; second.exe
```

When the output of one program is connected to the input of another, then the vertical bar is placed between them, and the data flows from left to right:

```
here.exe | there.exe
```

When a program should take its input from a source (which might not be another program), then the left-angled bracket follows the program and precedes the source of its input:

```
there.exe < here.exe
```

When the output of a program is redirected, then the right-angled bracket is used instead:

```
here.exe > there.exe
```

When a program is executed in isolation, its stdin is connected to the keyboard and its stdout is directed to the screen. This combination of keyboard input and screen output is called a terminal, console or Shell. When data is transferred to or from a device, such as the keyboard, one end of the pipe is connected to a program called a driver. The driver takes care of communicating with the device, and the rest of the system only sees a stream of characters.

3.1.2 Pipes and I/O Blocking

Pipes are simple data structures. Each pipe has a read head, a write head and a capacity to store data implemented as circular arrays of fixed size. The circular arrangement of a pipe gives the illusion of an infinite capacity of storage. As bytes are read, space is reclaimed and made available for overwrite. Figure 3.2 illustrates the main components of a pipe.

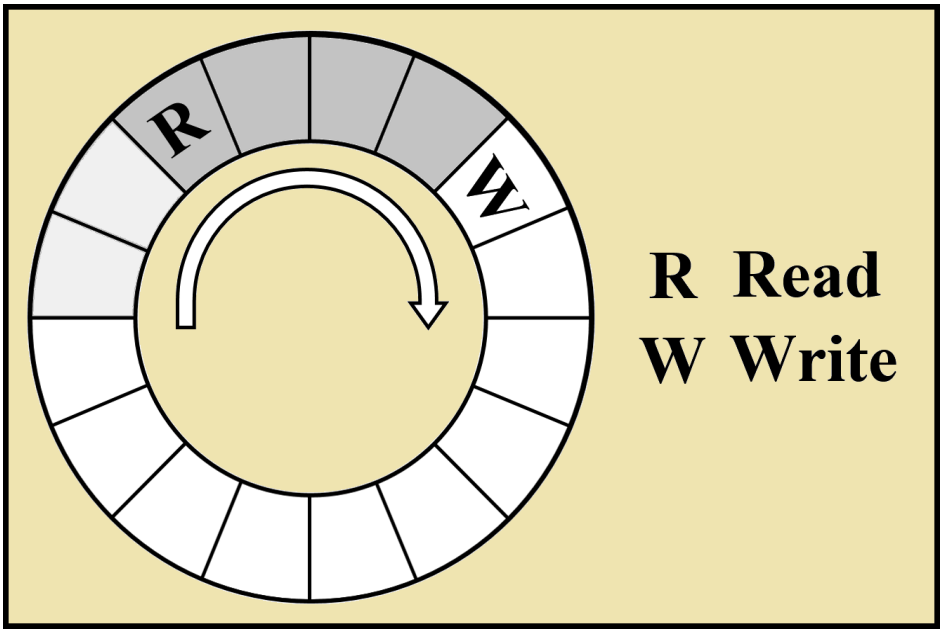


Figure 3.2: A Pipe with read and write heads.

The read and write heads are often given to different programs. Pipes synchronize data transfers between these programs through blocking. Reading from an empty pipe, or writing to a pipe which is full, will cause a program to block. When this happens, the program goes to sleep. When data, or space is made available, the program wakes up and continues execution.

3.2 Inter-Process Communication (IPC)

Inter-Process Communication (IPC) is the method by which data is transferred from one process to another. Each process executes in its own private address space. Only the Kernel has complete visibility of the global address space. Transferring data between tasks was traditionally performed by the Kernel. Liedtke (1995), showed that this approach is inefficient and necessary. Liedtke (1995) avoided this by allowing tasks to share their own memory with other tasks. By allowing both tasks to access the same memory, the Kernel no longer needs to intervene. Liedtke achieves optimal IPC with zero copy overhead. Liedtke (1995), used this as the basis for the L4 Micro-Kernel, which has established a niche in the telecommunications industry. Today this approach is called shared memory and is illustrated in figure 3.3 which compares traditional IPC to the L4 approach. Setting up a shared page requires modifying only a single pointer in the page directory. This minimal overhead of writing a single pointer is paid for only once during initialisation.

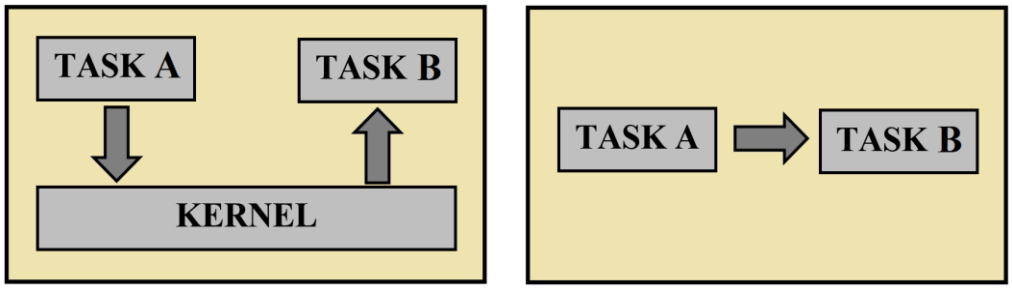


Figure 3.3: The L4 approach to Inter Process Communication.

Härtig (1997), extended the L4 Micro-Kernel producing a version called L4Linux which could be substituted for the Linux Kernel. This allows direct comparisons to be made between the Mach, L4, and Linux Kernels. Härtig (1997), found that L4Linux outperformed Mach by an order of magnitude. The Linux Kernel was slightly outperforming L4Linux on all fronts. However, the native L4 Kernel was in the worst-case outperforming even Linux in Piping and RPC (Remote Procedure Calls) which are essential to IPC.

These findings raised the question why L4Linux performed poorly compared to the native L4 Kernel. Härtig (1997), found the Syscall interface responsible for this discrepancy. System calls are implemented through interrupts. This causes a Context Switch from user mode to kernel mode, and a return trip back to user mode. A Context Switch is a costly operation. L4 avoids Context Switching by allowing tasks to communicate directly. Figure 3.4 compares these two approaches, showing that the L4 Micro-Kernel approach remains in user mode. There are clear benefits in avoiding kernel mode. User mode is the normal mode of execution. It is desirable to remain in this mode for as long as possible. Every trip to kernel mode requires a return trip to user mode. This is a costly operation which must be paid for twice.

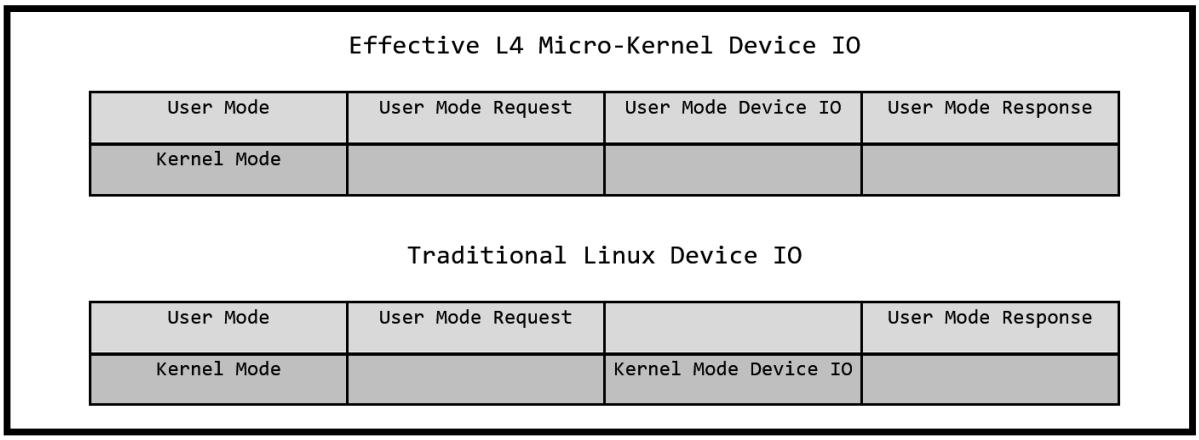


Figure 3.4: Context Switching overheads of Mono-Kernels and Micro-Kernels.

This rationale can be applied to device drivers. Leslie (2005), allows applications to issue device IO requests through normal user mode function calls. Tasarov (2015), found that this approach eliminates overhead in a key component of a Kernel. He states that file system IO represents 30% of Linux kernel mode activity. Tasarov adds that maximizing Solid State Drive throughput resulted in significant kernel mode overhead which dominated system usage. Linux has since adopted this approach with its user space device driver framework. This marks a partial return of the Micro-Kernel design to the forefront of Kernel development, as this is a current hot spot in the Linux Development community.

3.3 Graphical User Interface

The user interface is the part of the system which a human interacts with. When an interface is comfortable and intuitive, human effort is minimized. By reducing physical and mental effort, productivity is increased. Graphical User Interfaces take these principles and apply them to operating systems. The main limitation of the command line interface is its barrier to entry. Proficient use of the CLI requires prior knowledge of a wide range of utilities. Further, this approach requires effort to figure out the appropriate commands. This is unsuitable for the public who have limited interest in such technical pursuits. A more intuitive approach is required to meet the needs of the public. This problem inspired researchers from the Palo Alto Research Centre (PARC) to develop a system which placed the user first. This resulted in the first graphical user interface, the Xerox Alto, which was release in 1973.

Alan Kay (1987), states that the goal the Alto was to design a system which could be used by children, this brought intuition to the forefront of their requirements. PARC invited children into the office to operate the system, so that they could observe their interactions. This led to a system which was easy to learn, easy to use, and engaging even to non-technical users.

Icons play a heavy role in these systems. Its more natural and intuitive to manipulate a screen full of icons, then a screen full of text. It takes seconds to show a user how to point and click but requires training to explain the same operation through a command line. Kay (1987) states that there is something both intimate and engaging about a GUI, that invites its user to immerse themselves in the experience. While a screen full of obscure technical jargon intimidates new users. This section addresses the issue of human input output, and highlights some of the areas which dramatically improve the user experience, minimize effort, reduce strain and improve usability.

3.3.1 Readability on Screen Displays

Reading on-screen text for extended periods of time results in eye strain. This is partially due to the rasterization process which converts text into pixels. Weisenmiller (1999), considers rasterization of fonts in on-screen displays, and outlines the factors which improve of hinder readability. He found that printed text and digital displays have different requirements and limitations which impact the speed and accuracy of reading. Weisenmiller found that font size and line height have the biggest impact on readability, with clarity being the next biggest factor. The guidelines for typographic style are not considered here, but it is worth noting that sans serif fonts were found to reduce noise and improve accuracy.

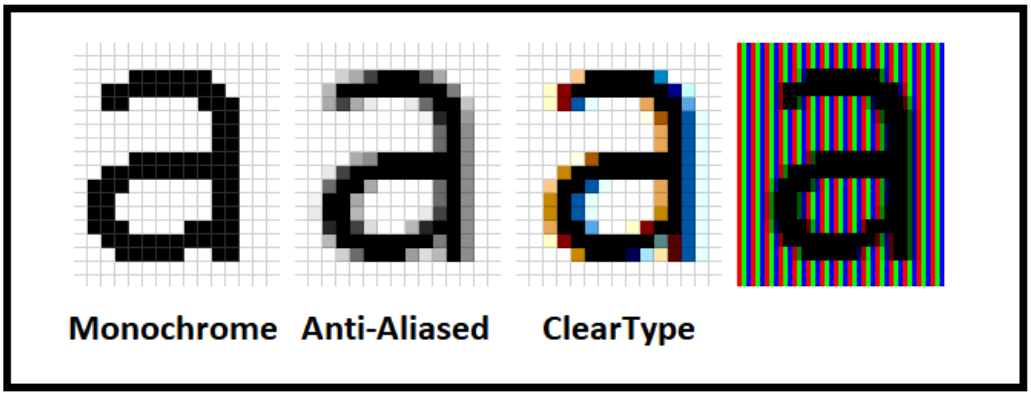


Figure 3.5: Comparison of text rasterization techniques.

Monochrome rasterizers make no attempt to smooth out edges and are optimal for low resolution displays where any attempt to smooth out edges would result in reduced clarity. Anti-Aliased rasterizers improve readability on higher resolution displays, where the text has a larger pixel footprint. ClearType is the sharpest, and best suited to on-screen displays. ClearType is a patented technology developed by Microsoft which renders text on a sub-pixel basis. Figure 3.5 shows how pixels are formed by combining red, green, and blue sub-pixels. ClearType manipulates these sub-pixels to form sharper edges. This doubles the horizontal resolution and has a significant impact on both speed and accuracy of reading. As a result, ClearType increases comfort and limits fatigue over extended periods.

3.3.2 Fitts's Law

Most engineering professions have the benefit of mathematical results which provide a means to precisely measure and quantify aspect of design. Software engineering is still very much in it's infancy in this regard. However, there is one notable exception in usability.

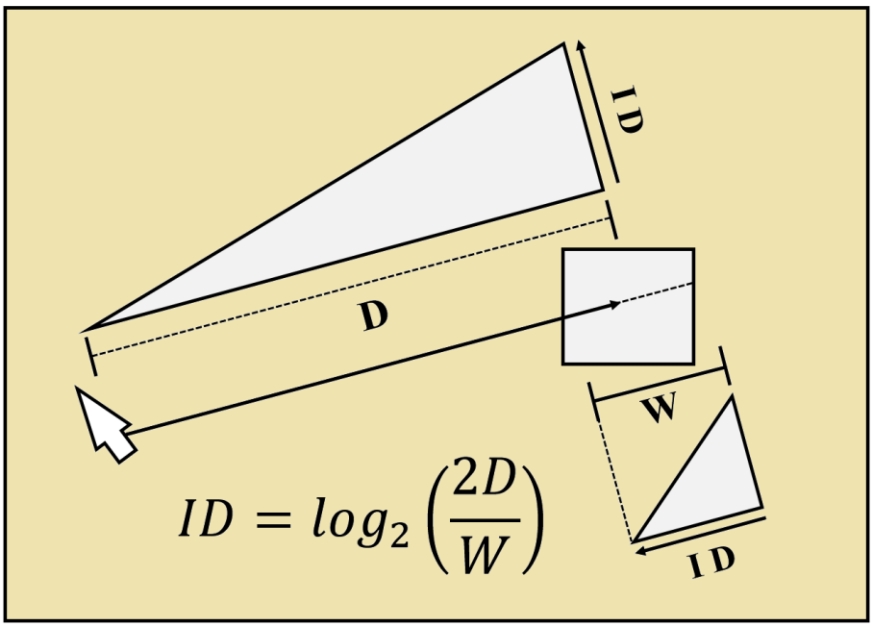


Figure 3.6: Fitts's Law demonstrating the index of difficulty.

Fitts (1954), researched the efficiency of human operators on a production line, with the intention of optimizing operator throughput. Fitts studied the ability of a human to move an object from one point to another. He found that the time required to move to a target area is a function of the ratio between the distance and the width of the target. This metric is referred to as the *index of difficulty* (ID), and is given by the following formula:

$$ID = log_2\left(\frac{2D}{W}\right)$$

This formula can effectively be used to minimize the difficulty of using a system in a controlled manner. The function is illustrated in figure 3.6 where the difficulty increases with distance but decreases as the target grows. From Fitts (1954), there are two ways to reduce difficulty, by increasing the size of the landing zone, or reducing the distance travelled.

Fitts (1954), has implications both on the size of icons, and their placements. In the context of a monitor screen, which has well defined boundaries, the movement of a cursor can be limited to the screen area. In this context, Fitts (1954) can be exploited to create regions around the edges of the screen which have infinite area. This makes the corners of the screen ideal for placing items which experience a high volume of traffic

To apply Fitts (1954) to travel distance requires building momentum into the cursor movement. A fixed movement step feels mechanical, and "sticky". This forces the user to move farther to cover ground. Building acceleration into the cursor reduces the movement required from the user. This has the effect of reducing the distance to the target. Adding momentum to the cursor creates a responsive interface which feels more natural. This allows the user to rely solely on muscle memory and reduces the need to anticipate how the mouse will respond.

3.3.3 Stacked Windowing System

This section presents an efficient implementation of a stacked windowing system which is designed to minimize latency. The system supports overlapping windows and relies on the natural tendency of the topmost window to receive most of the interaction. This results in a general case, which happens to be the optimal scenario, which allows efficient pixel transfers. The less efficient operation of restoring a partially obscured window to the topmost position requires the use of the painter's algorithm. This costly operation is only performed when switching between windows, and so can be viewed as a once off penalty. The windowing system presented here has been optimized to reduce latency. This system achieves seamless transitions and is highly responsive.

The system is designed around a stack of windows. The desktop manager treats this stack as a layered collection. The stack effectively maintains the required z-order of its elements. The stack also serves the additional purpose of recording which window should

receive user input. This simplifies the process of book-keeping and allows for efficient message passing. All non-topmost windows are effectively put to sleep until the user brings them to the topmost position. When this event occurs, it requires performing the painter's algorithm to ensure that the screen is consistent with the reshuffled stack. The painter's algorithm is a slow operation and is avoided when possible by using an acceleration structure, called the shadow buffer. The shadow buffer is equal in area to the framebuffer. It contains a duplicate of the framebuffer, minus the topmost window.

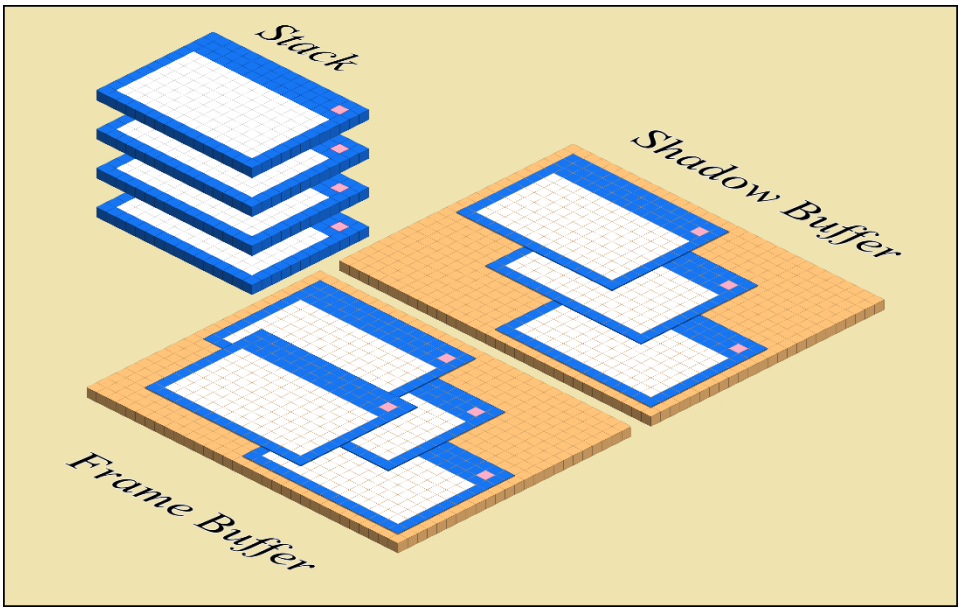


Figure 3.7: The Stacked Windowing System.

The main components of the windowing system are illustrated in figure 3.7. The left side of the diagram illustrates the window stack, which maintains the z-order of the various windows. The bitmap images on the right are the frame buffer, and shadow buffer. For efficiency, the colour depth of these buffers must be consistent with that of the windows. Updating the frame buffer, and shadow buffer is necessary only under certain conditions. There are three cases which must be addressed: Pushing a window to the top of the stack, popping a window from any position within the stack, and moving or resizing the topmost window. See listing 3.1 for an efficient implementation of each scenario.

Pushing a window to the topmost position requires writing the previous topmost window to the shadow buffer. This saves a copy of the previous window, so that it may be restored when necessary. Then the new topmost window is drawn to the framebuffer.

Moving the topmost window is very similar to pushing, but it does not require writing to the shadow buffer. Moving a window requires first restoring the obscured pixels from the shadow buffer. Then the window is drawn at the new position in the framebuffer.

Popping a window from the stack is the most costly operation and requires performing an iteration of the painter's algorithm from the bottommost window to the topmost. This invalidates the exposed region of both the framebuffer and the shadow buffer.

To maximize throughput of these operations, a highly optimized Blit function is required for the block transfers, such a function is given in listing 3.2. This block transfer should operate on spans of pixels, bounded by the clipping region. These spans must be aligned to the native alignment of the target architecture. In the case of the Intel 386, this means that each pixel should be aligned to a four-byte boundary. This alignment is critical to the efficiency of the Blit function. The Blit function presented in Listing 3.2 operates on spans of pixels. Only the start and end points are plotted, and pre-clipped to the minimal area required. This significantly reduces overhead and avoids the need to plot pixels independently. Experiments have shown that this operation must be completed within 33 milliseconds to be perceived as instantaneous. Any latency incurred in this process will severely hinder, rather than enhance, the user's ability to interact with the system.

3.4 Conclusions

User interfaces are the most important part of an operating system. They are the point of contact through which humans communicate with machines. The best systems place their users first. They make the human-machine dialog easy, by allowing people to speak to the machines on human terms. Minimizing effort is critical to productivity and ease of use. Once the user is operating within their comfort zone, even complex tasks can be solved with ease. When the human is productive, so is the machine. The achievements of the pipelined Shell, and the Graphical User Interface have brought humans closer to machines, by bringing machines closer to humans.

Over the past eighty years computers have continued to evolve, but their purpose remains the same. Computers exists to improve human productivity. Technology moves at an alarming rate, though it seems that advances in human computer interaction proceeds slowly. The Graphical User Interface first appeared in 1973. Only now are we beginning to see the emergence of its successor, the natural user interface, in which computers respond to gestures and spoken commands. Today we are surrounded by smart devices. The interfaces of today are insufficient for a world where all things can think for themselves. This technology will bring with it new challenges and discoveries.

PART II

Software Design

Chapter 4 – Requirements and Design Prerequisites

The purpose of this chapter is to declare the project goal statement, to outline the project requirements, to define the scope of this project, and to establish an effective development process such that progress can be made towards this goal.

4.1 Goal Statement and Project Scope

The goal of this project is to develop a Multi-Tasking Operating System, referred to as Spartan or simply the system. The system shall provide a Graphical User Interface as its primary Human-Machine interaction model. The system shall operate in 32-bit protected mode and shall execute on Intel 386 Processors running on a PC Compatible Motherboard. The system shall host a plurality of Applications, each of which executes in its own isolated virtual address space. The system shall also support at a minimum those devices which humans depend upon in order to interact with a machine, namely the keyboard, mouse and the video display.

The focus of the project is on the Human Machine Interaction and not on designing a full featured runtime environment in which to host applications. The system must support the keyboard, mouse and video display. Secondary storage support is out of scope, instead the system shall use a Ram Disk which is a region of memory that is treated as though it were a disk. Multi-Tasking and Memory Management are requirements, the efficiency of these systems is not the focus of this project. Grub shall be used as a bootloader and the bootstrap process shall be restricted to establishing an effective execution environment.

4.2 Functional Requirements

- Be deployed to a bootable CDROM.
- Execute on Intel 386 processor running on a PC Compatible motherboard.
- Setup a 32-bit protected mode execution environment.
- Provide Multi-Tasking support where each process gets a fair share of execution time.
- Provide Memory Management support in which each user mode application operates within its own isolated virtual address space.
- Support the following devices: Keyboard, Mouse, and Video Display adapter.
- Provide a Ram Disk based File System.
- Provide a Graphical User Interface in the form of a Stacked Windowing System.

4.3 Non-Functional Requirements

- Fail Safe, the system shall not corrupt computer resources. This is achieved by using a Ram Disk and avoiding any form of disk IO to persistent secondary storage.
- Error Recovery, the system must continue to operate in the event that a user mode application should crash, or in the event that system data structures become corrupt.
- Intuitive, easy to learn and easy to use. This shall be achieved by implementing a Graphical User Interface.
- Responsive, the latency between user input and visual feedback should be minimized.

4.4 Development Process

The Spartan system is developed using a rapid prototyping methodology based on Agile. The approach employs incremental development. Demonstrations of progress utilized prototyping to showcase functionality. The Kernel must be in a mature and stable form before development of the Shell is possible. Demonstrating finished work is only possible once the Shell has also reached a mature state. Prior to this milestone, rapidly prototyped demos are used to provide visible feedback. Incremental development methodologies allow parts of the Shell to be developed as Kernel functionality is made available. This forms a feedback loop which allows the system to be developed on a “needs driven” basis.

4.4.1 Development Environment

The Microsoft Visual Studio C Compiler is used as a Cross-Compiler for the Spartan system. The system is mainly written in Pure C and requires some features to be implemented directly in assembler with the Flat Assembler (FASM) being used for this purpose. The Grand Unified Bootloader (GRUB) is used as a Boot-Loader. There is zero runtime support. Once in operation, the Spartan system is completely self-sufficient for its needs.

Pure C lacks support for Object Oriented Programming. To overcome this issue objects and interfaces have been modelled using plain old data structures. Early in development objects were modelled such that both the data and methods were embedded within the same data structures. It soon became apparent that this was a mistake, making it more difficult to instantiate an object and risking the accidental overwrite of method pointers. The solution to this problem was to separate object behaviour from state. Behaviours are encapsulated in an associated interface; these interfaces are realised as data structures which contain only function pointers. For the purposes of this design document the object class and its associated interface are conceptualized as a single unit.

4.4.2 Deployment Process

The Spartan system cannot execute from within another operating system. Spartan must be executed directly on the hardware in order to operate correctly. It is necessary to test early and often on real hardware. An effective build environment and deployment process are pre-requisites and must be established before development of Spartan can begin. Some tools were developed in-house to simplify the build and deployment process. These custom tools integrate seamlessly into Visual Studio as a post-build stage. These custom tools produce a bootable CDROM image in addition to the executables. The executables themselves cannot run under windows, but the CDROM image can be loaded into a Virtual Machine or burnt to disk and executed on real hardware. This CDROM image can be thought of as the release executable.

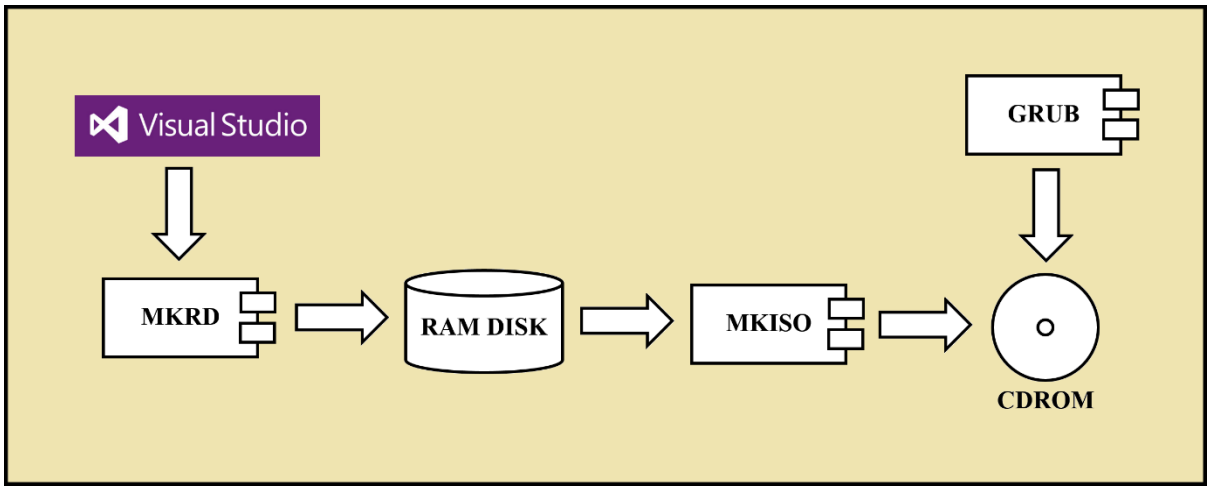


Figure 4.1: Build Process and CDROM authoring.

The build process is illustrated in figure 4.1 where the output from Visual Studio is piped through a chain of custom utilities to produce a bootable CDROM image which has GRUB pre-installed. Burning CDROM disks during development is extremely wasteful and time consuming. It is preferable to also support USB deployments which makes testing on real hardware much faster and more convenient.

The bootable CDROM contains a Ram Disk which in turn contains the Kernel. Spartan does not support secondary storage devices, instead the system uses a Ram Disk. A Ram Disk is a region of memory which is treated as though it were secondary storage. A File System is layered on top of the Ram Disk to provide file and directory hierarchies. The Ram Disk does not contain sub-directories, it has a single root directory which contains all files. A convention is used whereby files are organised using separators embedded within the file name. Thus, all file names are stored fully qualified. The File System driver abstracts away this detail so that applications are presented with a directory hierarchy.

A program called mkrd was developed to create Ram Disk images from the contents of a directory. Another program called mkiso was developed to convert the Ram Disk into a bootable CDROM image. The Ram Disk contains executables in the Microsoft Portable Executable (PE) format. The benefit of choosing PE instead of ELF is that PE files can be made identical in-memory to the on-disk representation. These executables are stored in a ready-to-run state so that there is zero runtime overhead for Kernel dependencies. The mkrd tool is responsible for this transformation. It decides where to place the executables in the Ram Disk and relocates these executables so that all internal pointer references are correct. Once all internal pointers have been resolved, then mkrd can resolve external dependencies. This is achieved by parsing the import tables which specify DLL's and declare imported functions.

Executables are ready-to-run once all pointer references have been resolved. At this stage a bootable CDROM image can be authored. A second program called mkiso was developed for this process. Mkiso accepts a blank CDROM ISO file which has GRUB pre-installed. The Ram Disk is inserted into the ISO image which requires manipulating the CDROM ISO9660 file system. The GRUB.CFG file is updated so that GRUB will load the Ram Disk into memory. Once these arrangements have been made, then development of the main system may proceed.

Chapter 5 – Software Design

The Spartan system is made up of the Kernel, the Shell, the Desktop and a plurality of Applications. Figure 5.1 illustrates the main components of the System and highlights their dependencies. Before discussing these components in detail, it is useful to define the scope boundary of this project. This will help to define the Spartan system.

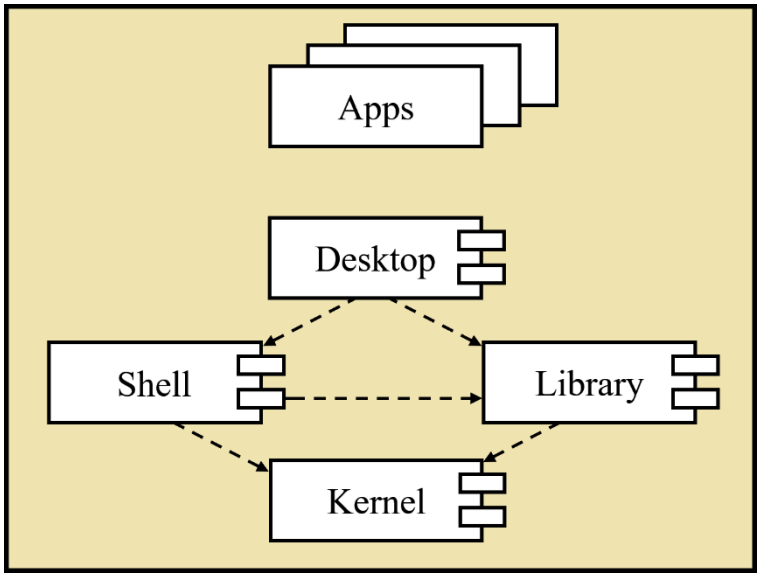


Figure 5.1: Deployment Diagram representing the System Overview.

The Library contains functionality which was required but out of scope. Some Library functionality was sourced from the public domain, such as implementations of Binary Search and string functions sourced from GLIBC. Only the Library contains public domain code and for this reason the Library should be considered a dependency and not an integral part of the project. Applications are developed for demonstration purposes only. They are not documented in this work. The three remaining components (The Kernel, Shell and Desktop) are all integral components. In addition to these core components some build utilities were developed and are part of the Spartan system, though they only execute at compile time.

5.1 System Overview

The three main components of the Spartan System are the Desktop, the Shell, and the Kernel. These components contain namespaces which provide interfaces. Figure 5.2 illustrates the system at the namespace granularity. Detailed discussion of these interfaces will be deferred to a later point in this document, this section provides a high-level overview of the Spartan system.

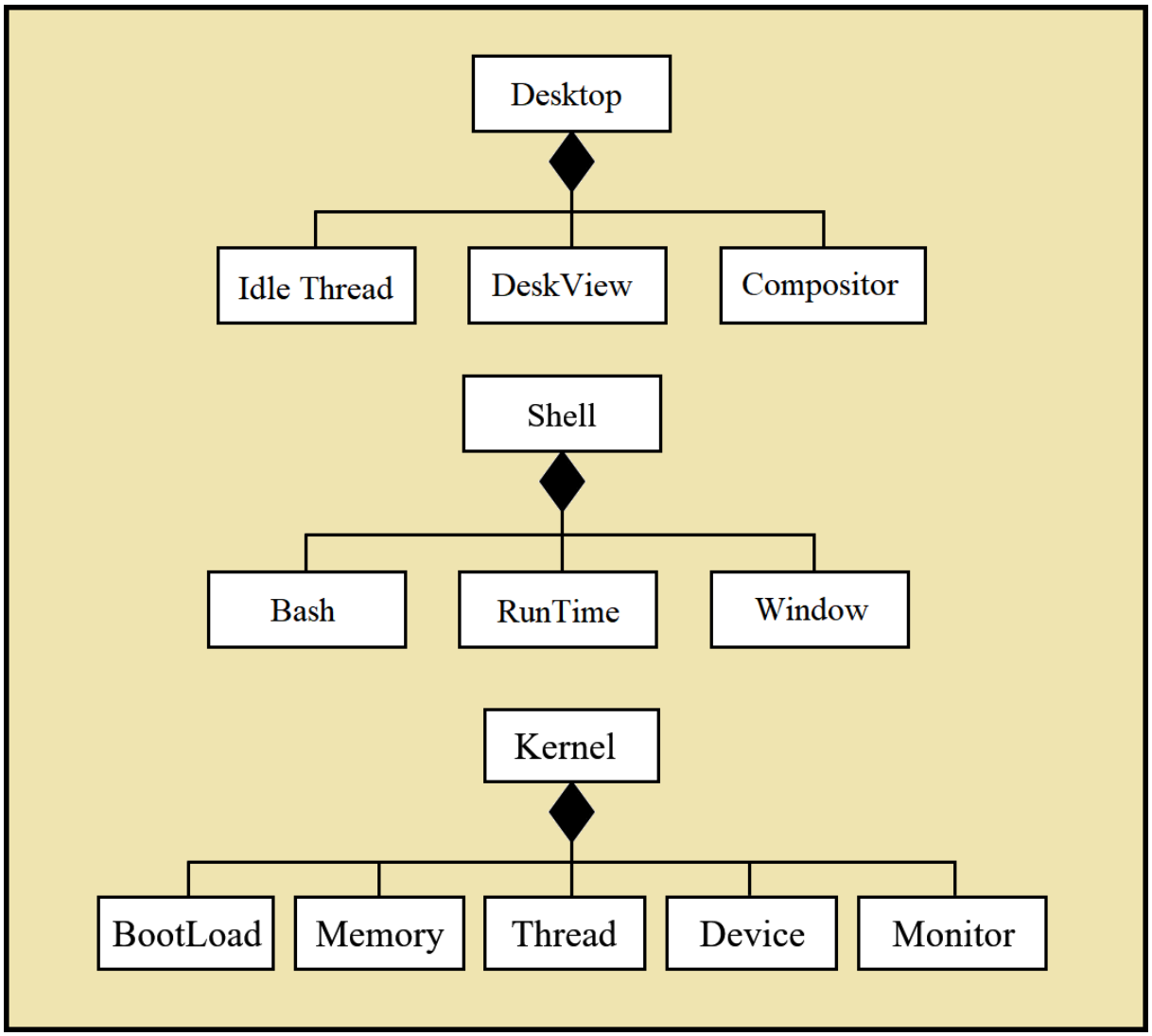


Figure 5.2: Major namespaces of the Spartan System.

The namespaces contained within the Shell and the Kernel form the runtime support which is made available to user mode Applications, while the namespaces contained within the Desktop are internal to the Desktop and not made available to user mode applications.

5.1.1 Desktop Namespaces

The Desktop contains a DeskView, an Idle Thread, and a Compositor. The DeskView manages the Stacked Windowing System. It displays the desktop, taskbar, mouse cursor and keyboard caret. The DeskView monitors keyboard and mouse events and dispatches them to the currently active window. The Compositor is responsible for managing the video display. One of the Compositors responsibilities is to combine the independent window canvases into a single image. The Idle Thread waits for signals such as SIGKILL and processes these events.

5.1.2 Shell Namespaces

The Shell provides runtime support for Applications. The Bash namespace is a command-line interpreter which implements the command-line interface. The Window namespace provides support for the Graphical User Interface and contains interfaces for creating and managing Widgets. It also provides implementations for generic Widgets such as a Button, List View and Text Box. The Window namespace also contains the Event Queue which processes events such as On Click and On Key Down. The RunTime namespace contains additional runtime support such as font rendering, bitmap rendering and text processing routines.

5.1.3 Kernel Namespaces

The Kernel contains low level runtime support. The Thread namespace is responsible for process management and task scheduling. The Memory namespace is responsible for Memory Management and Paging. The Device namespace contains device driver implementations and also contains the File System Manager. The Monitor and BootLoad namespaces are used internally by the Kernel. The BootLoad namespace is responsible for system initialization and is executed once at start-up. The Monitor centralizes Logging and is responsible for Error Detection and Recovery, it allows the system to recovery from error states and can restore the system to its “Last Known Good” state in the event of system data corruption.

5.2 Expanding the Desktop Namespaces

By expanding the view of the Desktop namespaces, one immediately arrives at the interface granularity. The Desktop contains six interfaces. The Idle Thread and the Compositor are namespaces which contain only a single interface each: IIdleThread and ICompositor. The DeskView namespace contains four interfaces: ICaret, ICursor, IDeskView and ITaskBar. Figure 5.3 depicts the Desktop namespace topology at the interface granularity.

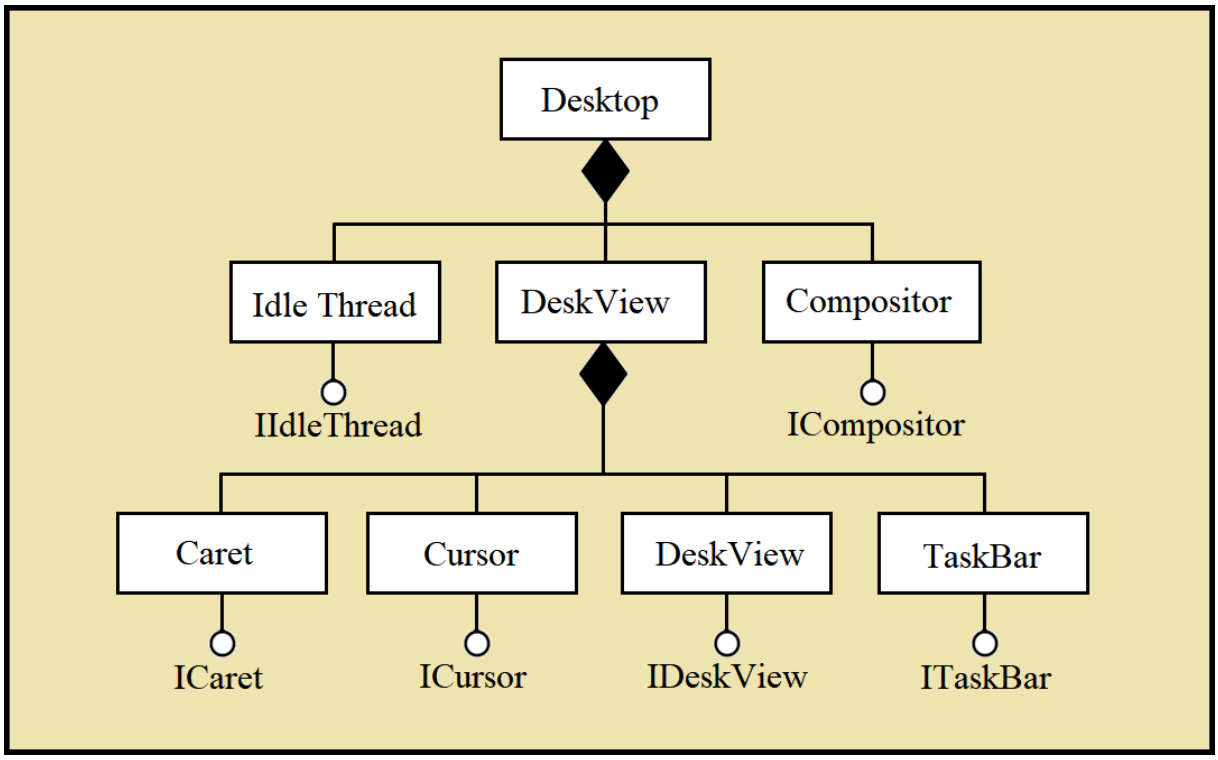


Figure 5.3: The Desktop Interfaces.

The Idle Thread processes signals from applications, the Compositor interfaces with the display adapter, the Caret interfaces with the keyboard and the Cursor interfaces with the mouse. The TaskBar acts as a short-cut to useful applications, and the DeskView is simply a background. These interfaces are used internally by the Desktop itself. The Caret, Cursor, DeskView and TaskBar are all Widgets which act as an interface between the user and the system.

5.3 Expanding the Shell Namespaces

The Shell namespace topology is more involved than that of the Desktop. The Shells namespace has three major branches: The Bash namespace, the Window namespace and the RunTime namespace. As can be seen in Figure 5.4, it is possible to depict in a single image the expanded view of the Shell namespaces at the interface granularity, this will not be the case when describing the Kernel.

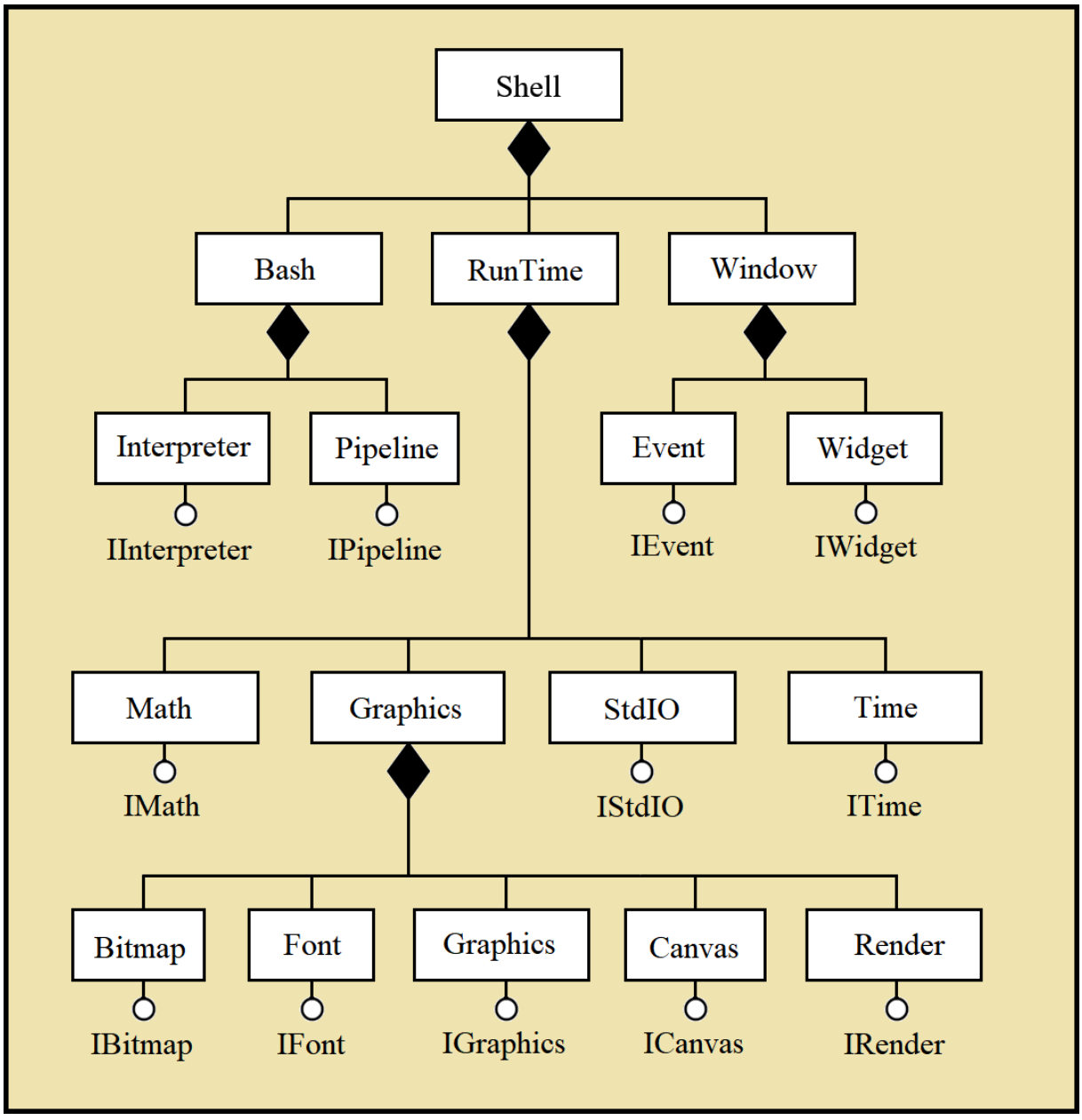


Figure 5.4: The Shell Interfaces

The Bash namespace is responsible for the command-line interface. It contains an IInterpreter interface which is a command-line interpreter. The IPipeline interface is responsible for IO redirection. The Window namespace is the center of the Graphical User Interface. It contains the IEvent interface which drives the Event Queue and provides default event handling for common events such as On Key Down, On Draw, and On Click events. The IWidget interface provides support for window creation and provides implementations for generic widget such as Buttons and List Views.

The RunTime namespace houses the system services. The IStdIO interface unifies the Kernel's Device Drivers with the File System and provides convenience functions for processing textual data which is streamed to these devices. The IMath and ITime interfaces provide convenience functions for supporting applications, these interfaces are not discussed in detail as they are not related to operating system development. The Graphics sub system contains a number of interfaces which are used by the Widgets. The IBitmap interface provides support for displaying icons and images, the IFont interface is responsible for rendering text, the IGraphics interface provides support for rendering vector graphics such as lines and polygons.

The ICanvas interface acts as a drawing surface, every application has its own private canvas for drawing its visual output. The IRender interface is a virtual machine which batches rendering tasks as part of the Rending Pipeline. The terminal end point of the Rendering Pipeline must synchronize its updates with the refresh rate of the display adapter, and as such it is incapable of responding immediately to requests made by applications. Later when it has the opportunity, it will recreate the commands requested by the applications such as moving a rectangle of pixels from one region of the screen to another. These operations are encoded as opcodes and are processed by the IRender virtual machine. This arrangement is easy to manage and extend. The Rendering Pipeline subsystem will be discussed in more detail at a later point.

5.4 Expanding the Kernel Namespaces

The Kernel is the foundation of the Spartan system and this is evident by the complexity of its namespace. Figure 5.5 depicts an expanded view of the core namespaces: The Thread namespace provides support for Process Management and the Memory namespace implements Memory Management.

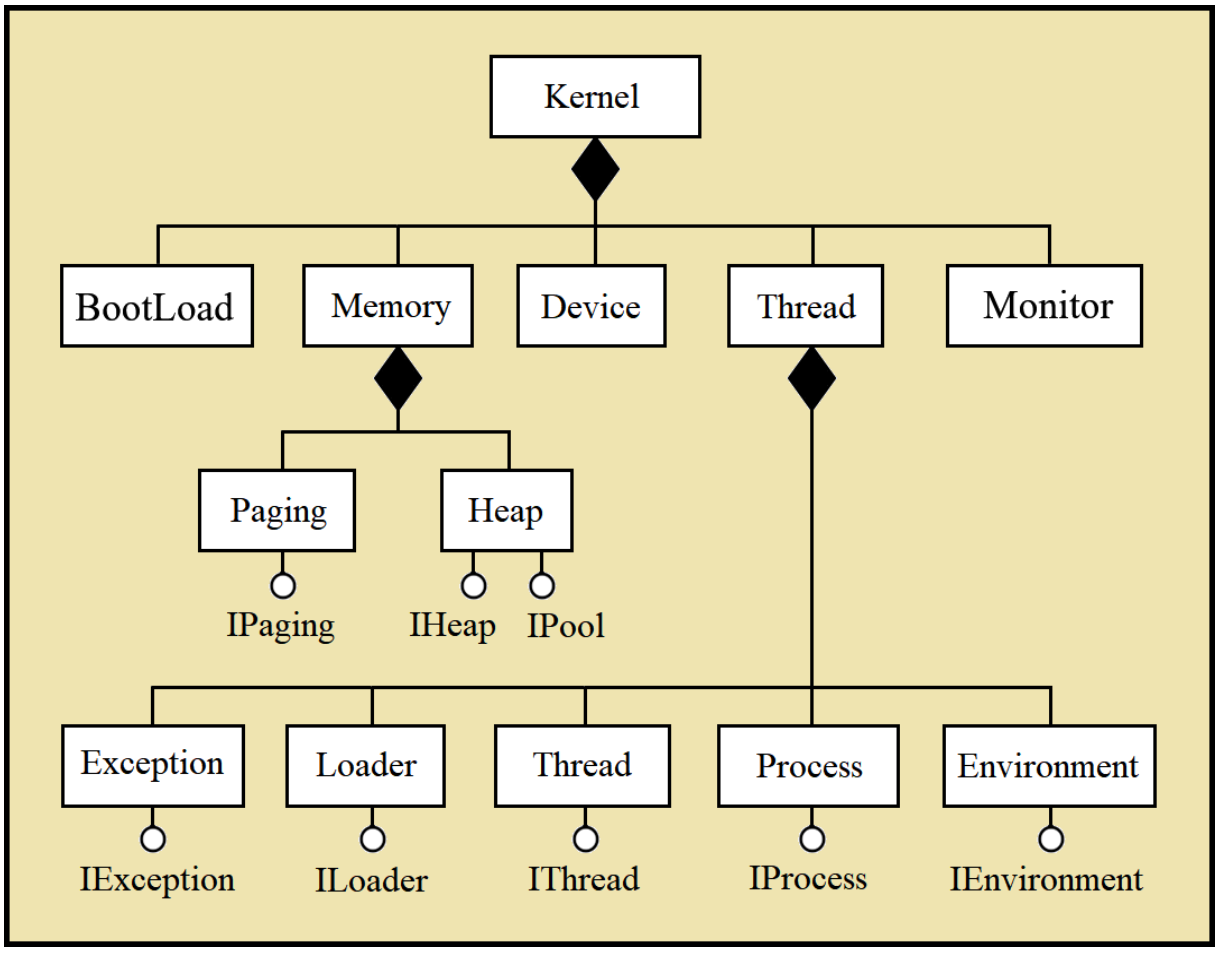


Figure 5.5: The Kernels Memory and Thread Interfaces

The IHeap interface provides Memory Allocation support. It implements is the Next Fit algorithm. The IPool interface is a generic container used throughout the system and is a special case of the IHeap interface. The IPaging interface provides support for virtual address spaces. The IPaging interface implements the Clock Replacement and Lazy Loading algorithms.

The Thread namespace is responsible for Process Management. The IThread interface implements the Process Scheduler and implements the Round Robin scheduling algorithm. The IThread interface is also responsible for Blocking and Unblocking a thread. The IProcess interface manages processes. A process is defined as an executive which has its own private virtual address space. A process may contain multiple thread of execution which all share the same virtual address space. The IProcess interface provides support for managing these threads as a unit.

The IEnvironment interface is closely coupled to the IProcess interface and was separated out after it had grown sophisticated enough to warrant its own interface. The IEnvironment interface is responsible for establishing an effective execution environment in which to host user mode applications. It sets up the applications Heap and Pipe systems. These systems form part of the RunTime execution environment and the IEnvironment interface initializes the services which the Kernel provides. Additional initialization is performed later by the Shell as part of the RunTime namespace. The IEnvironment interface also has the responsibility of mapping the various sections which make up the applications address space.

The ILoader interface is responsible for loading an application from the file system and placing the executable in a “Ready to Run” state. This involves relocating the executable and resolving external dependencies. This interface is used by the mkrd utility when authoring the RamDisk image. The IException interface acts as a trap in the event that an application performs an illegal operation such as attempting to divide by zero or attempting to reference non-existent memory. Kernel mode exceptions are not tolerated but applications are expected to raise exceptions from time to time. The Spartan system does not provide runtime support for high-level try-catch style exception handling, though this component should ideally provide support for these language features.

The Kernels Device namespace houses the file system manager and a plurality of device drivers. Figure 5.6 illustrates the Device Management sub system and shows that each device is exposed to the rest of the system as a pipe. This follows the UNIX tradition that everything is a file.

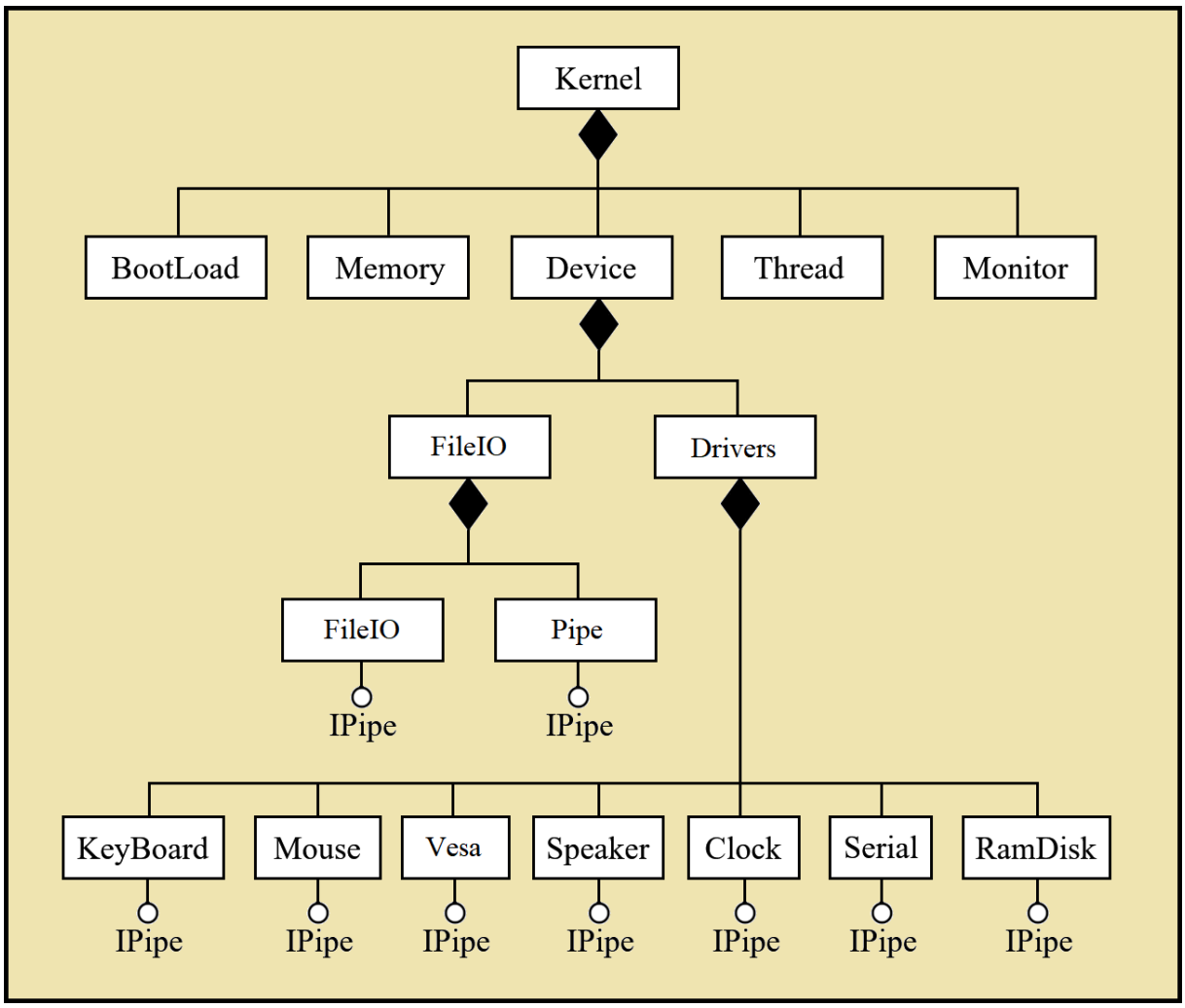


Figure 5.6: The Kernels Device Interfaces

One of the major roles assumed by the Kernel is to act as an intermediary between the hardware and the rest of the system. Because of this, many of the Kernels components implement well defined specifications. A consequence of this is that there is little freedom in the implementation of these specifications. The BootLoad and Device subsystems are particularly tied to industry standard specifications.

Figure 5.7 depicts an expanded view of the Monitor and BootLoad namespaces. The Monitor acts as a supervisor to all other Kernel components. All objects in the Spartan system are stateless and they process the state they are provided. It is the Monitors responsibility to manage the state associated with each of the Kernel components. The Monitor centralizes Logging and is also responsible for Error Detection and Correction. The Monitor is capable of rolling back any operation and can recover from otherwise fatal errors.

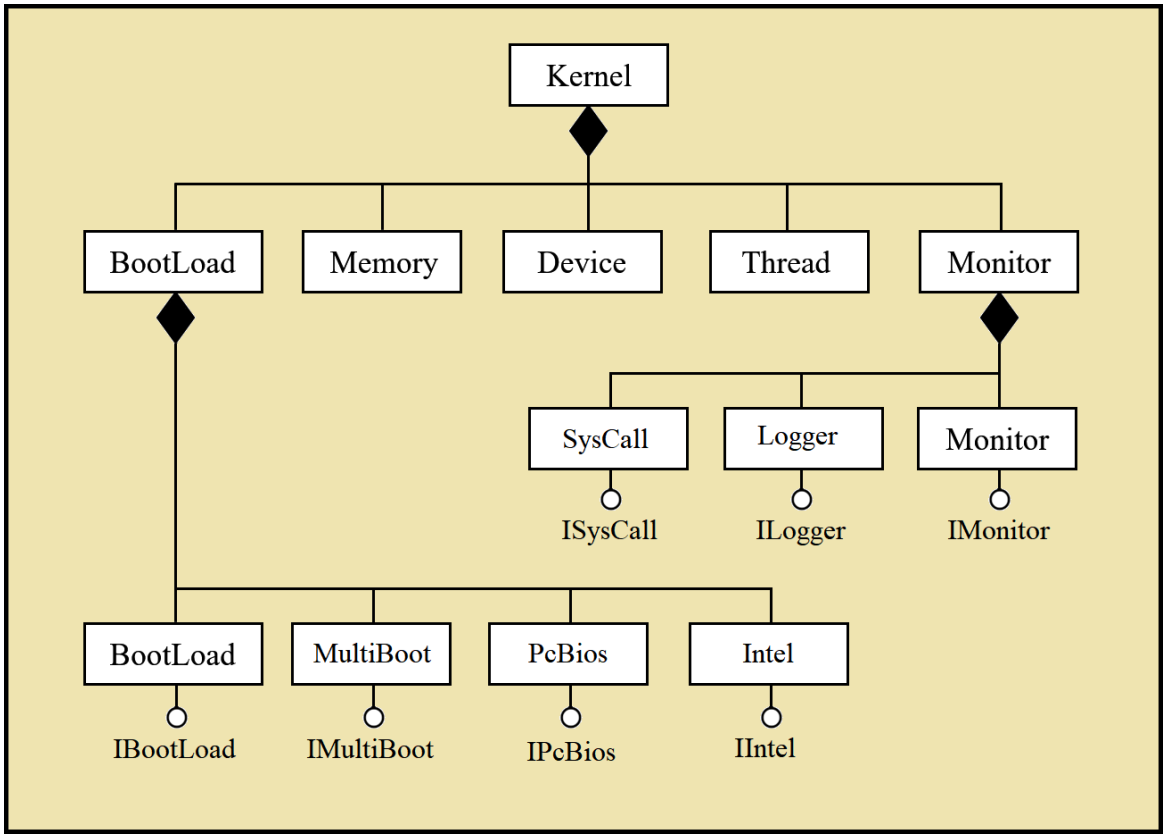


Figure 5.7: The Kernels BootLoad and Monitor Interfaces

The BootLoad namespace is responsible for initialization and is executed once at start up. This process involves parsing information provided by Grub, initializing the processor and device driver initialization. The BootLoad system liaises with the Monitor to ensure that each system component prepares its data without corrupting the rest of the system. The final step in the bootstrap process is to execute runtime unit tests to ensure that the Kernel behaves correctly before the system goes live.

Chapter 6 – Detailed Design

This chapter provides a detailed analysis of the design of the Spartan system. Operating Systems are a large problem space and there is neither enough time nor space within this document to fully detail the design of the entire Spartan system. The discussions contained within this chapter shall exemplify the design process as a whole. This chapter is not intended to be an exhaustive account of the design of the entire Spartan system. The author has selected a number of subsystems which are documented in detail with the remaining subsystems receiving only the high-level overview provided in chapter 5.

Before proceeding with the analysis, it should be noted that there is a limitation to the design of this system. The Spartan system was developed in the Pure C programming language which lacks built in support for Object-Oriented Programming. To address this issue an approach was adopted whereby objects are modelled through Pure C data structures. This approach has its limitations and requires great effort to implement polymorphism and class hierarchies. With this approach the developer assumes the responsibility of ensuring that the various implementations remain interchangeable. This is an intricate task best performed by a compiler, and when performed manually it results in a system which is difficult to modify and must be extended carefully. Because of this, OOP features such as polymorphism and inheritance are employed sparingly and only in those cases where the benefits are greatest, such as Device Drivers and Widgets.

In addition to Object-Oriented design, the design of the Spartan System was also driven by Data Oriented design practices. DOD considers it important to have full disclosure of the topology of a data item while OOP hides this information through encapsulation. Apart from

this divide the two approaches complement each other with OOP being more effective at simplifying system-wide, high-level design and DOD being more effective at simplifying individual units. Readers unfamiliar with Data-Oriented design may need to note some terminology used under this discipline. In Data-Oriented design, “Problems” are modelled as data and “Solutions” are modelled as code. Data-Oriented Design (DOD) analyses solutions to determine the high frequency use cases and may redesign the data to make these functions as simple and efficient as possible.

6.2 Memory Management Subsystem

The Memory Management subsystem provides three interfaces: The IPool interface, the IHeap interface and the IPaging interface. This section describes the IHeap and IPool interfaces in details and outlines the mechanisms used to realize memory allocation and data storage. The IPaging interface is not described here as its implementation is largely defined by the Intel Architecture Reference Manual Volume 3, chapters 2, 3, 4 and 5. The definitions provided by this reference manual are precise and there is no benefit in repeating that information here.

6.2.1 Pool and Heap Problem Analysis

The Spartan system uses a variety of abstract data types including Stacks, Queues and Linked Lists. The Spartan system is developed in the Pure C programming language which does not have the new and delete operators. An alternative to these operators is required so that the rest of the system can focus on their own responsibilities and not involve themselves in the business of micro-managing memory allocations. The Pool and Heap classes assume this responsibility. A Pool is a container which organizes a homogenous array of objects instances. A Pool has a granularity which defines the size of the objects it contains and a count which defines the capacity of the Pool. A Heap is a special case of a Pool which can merge multiple elements to meet requests for allocations of arbitrary sizes.

6.2.2 Pool and Heap Problem Model

Both the IHeap and IPool interfaces implement the NextFit algorithm to organize collections of objects. Managing these collections requires some overhead and this is implemented as an array of nodes and a next pointer which records the current position. This arrangement is depicted in figure 6.1 which illustrates the relationship between the next pointer, the node array and the pool of objects.

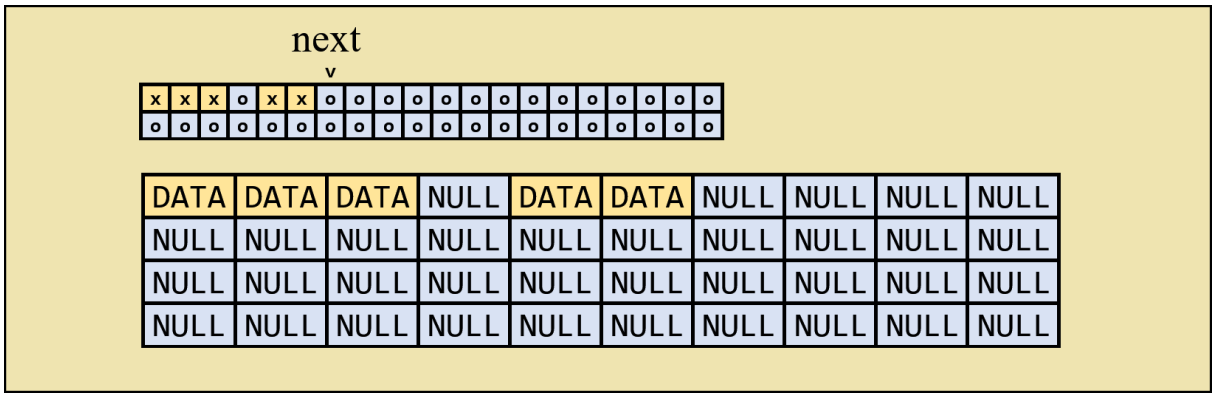


Figure 6.1: Heap and Pool Problem Model.

The system Heap is managed by the Kernel to allocate Partitions to Processes, and each Process has its own private Heap for internal allocations. In addition to its private Heap, each Process also has a Pool for its Pipes, Event Queue, and Widgets. Allocating storage for objects and later Releasing that storage are actions which are performed once per object. These operations are assumed to be less frequent and it is anticipated that the most common operations are Inserting and Removing objects from the Pool.

In order to make Insertions and Removals both simple and efficient it is specified that a Pool is an unordered array of elements. Each item in the Pool has an associated index, and this index is transient, that it is to say that the index is invalidated upon removal and it is assigned anew upon each insertion. If the order of the items in the Pool is important to the implementation, then the responsibility falls to the implementation to maintain the order appropriately.

6.2.3 Pool and Heap Solution Model

From the problem model we can derive the class hierarchy depicted in figure 6.2. The Node field points to the overhead array which records which items are allocated and which have been released. The Data field points to the Pool of object instances. The Gran field defines the size of the object instances, or the granularity of the Pool. The remaining fields and methods are named with sufficient clarity to describe their purpose.

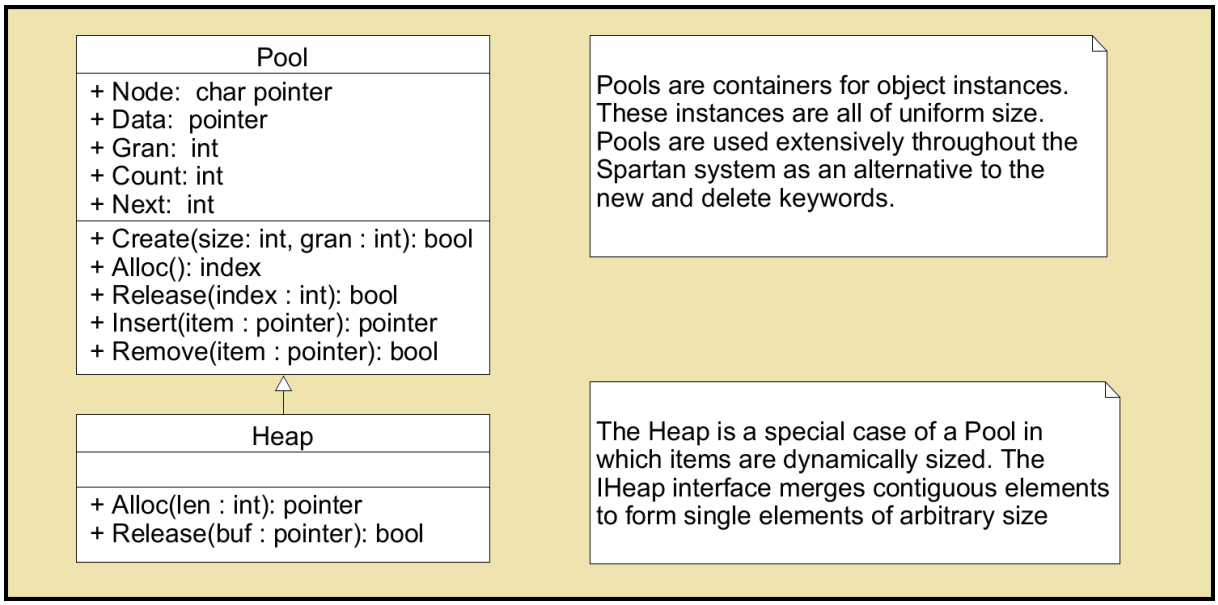


Figure 6.2: Pool and Heap Class Diagram.

It can be seen from the solution model depicted in figure 6.2 that the Heap inherits directly from the Pool class. This relationship is later expanded to include other systems which also derive from the Pool class, such as the Ready Queue used by the Process Scheduler and the Event Queue used by the Windowing subsystem. They are omitted from figure 6.2 for clarity reasons. These extensions enforce restrictions on the order of the items contained within the Pool and this is one key area where compiler support for polymorphism is particularly relevant. Since Pure C lacks this built-in compiler support for polymorphism, this model of a Pool was revisited numerous times to arrive at the generalized case depicted above.

6.3 Process Management Subsystem

Threads are tasks which the system executes. Processes are groups of Threads which share a common Virtual Address Space. A Process has a Process Control Block which is used by the system to manage a Threads as part of the Process Scheduler and during the Loading Process. Each Process also has a Process Environment Block which contains important information relating to the Process and is used to provide the Process with RunTime support for the various system services. This section describes the mechanisms used by the Spartan system to manage Processes and to manage the Process Scheduler.

6.3.1 Process Scheduler Problem Analysis

The Process Scheduler maintains a Queue of Threads to be executed. When a Thread is created, it enters the Scheduling system by joining the end of the Ready Queue. When a time-slice elapses, the Scheduler places the running Thread at the end of a Ready Queue, then takes the next Thread from the front of the Ready Queue and places it in a state of execution. This operation is called a Context Switch and is performed by the Low-Level Scheduler. Context Switching is driven by a periodic timer analogous to an alarm clock. The Spartan system uses the Real Time Clock (RTC) as the periodic timer which drives the Scheduling system.

When a Thread cannot continue because it is waiting for an external event such as Device IO, then it is removed from the Ready Queue and placed in the Blocked Queue. When a Thread receives the service it was waiting for, then it is removed from the Blocked Queue and placed at the end of the Ready Queue. When a Thread completes its task, then it is removed from the Scheduling system which makes room for other Threads to enter the system. This problem describes a finite state automaton with a Ready, Running and Blocked state and this is the approach taken for its design and implementation.

6.3.2 Process Scheduler Problem Model

The Process Scheduler is modelled as a Pool of Threads. The ability to iterate over the elements of this Pool is required functionality which a basic Pool lacks. Iterating through a Queue is similar to iterating through a Linked List. A naïve approach achieves this through "Pointer Chasing". This approach thrashes the cache by breaking data locality. The same behaviour can be achieved using an array of contiguous elements. Array guarantee data locality and maintain cache coherence. This substitute data structure is called a Ring or circular array and it is the underlying mechanism used to model the Ready Queue and Blocked Queue.

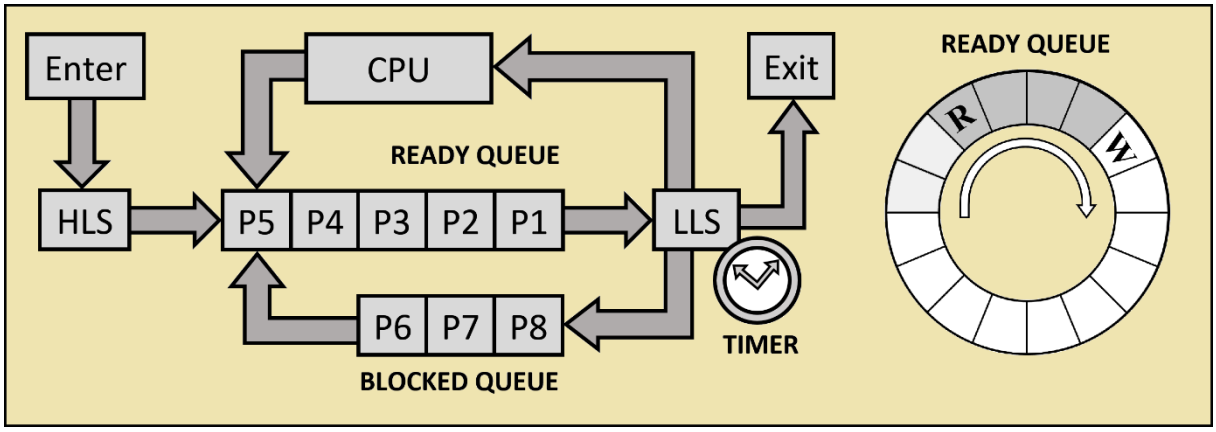


Figure 6.3: Process Scheduler Problem Model.

Both the Ready and Blocked Queue have a Head and a Tail. The Tail marks the end of the Ring and is the position where newly created Threads are inserted. The Head acts as an iterator over the elements of the Ring and the element indexed by the Head of the Ready Queue is the Thread which is in the running state. When the Head reaches the Tail, then it loops back to the start of the Ring and continues to iterate from there. When an element is removed from the Ring then it is overwritten by the element at the end of the Ring and the Tail is decremented. Iterating through the Ring is the high frequency use case which drives the design of this system. Figure 6.3 depicts such a system and resembles the problem model used for the Pool except that the elements are contiguous, and the Next pointer has been replaced by Head and Tail pointers.

6.3.3 Process Scheduler Solution Model

The Process Scheduler is modelled as a type of Pool. The Thread Pool maintains a collection of Pcbs, Pebs and Contexts. Each Process is associated with one Pcb, each Pcb has one Peb and each Peb has a Context. These three classes define a Process and when the Spartan system manages a Process it is these objects that are manipulated. Figure 6.4 illustrates the relationship between a Pcb, Peb and a Context in relation to the TaskPool.

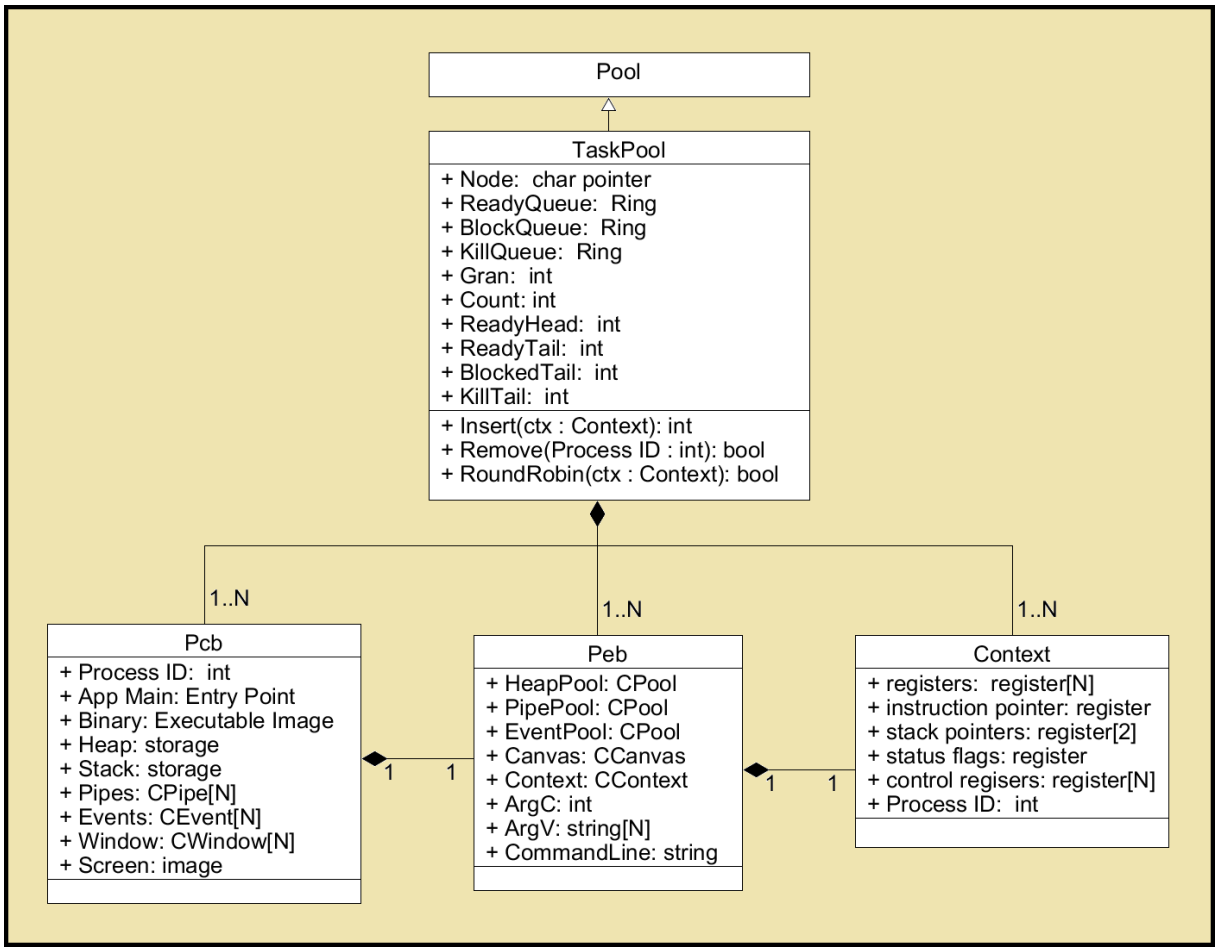


Figure 6.4: Process Scheduler Class Diagram.

The RoundRobin method is invoked by the Low-Level Scheduler and performs the Context Switch. It is during the Context Switch that the Head is iterated through the Ready Queue to obtain the Thread which is placed in the running state. The Kill Queue is used to store Threads which have been removed from the Scheduler so that the system can reclaim those resources.

6.3.4 Create and Destroy Process Problem Analysis

When the user wishes to execute a program then the system needs to load the program from the file system and place it in a ready to run state. The system must also initialize the RunTime environment which will support the newly created Process. This environment includes a private Heap, a Pool of Pipes, a Drawing Canvas, an Event Queue and a Pool of Widgets. Once the RunTime environment has been established then the Process is ready to be added to the Process Scheduling system. Later when the program wishes to exit, the reverse operation needs to be performed so that all allocated resources are released and reclaimed by the system. A Process cannot destroy its own resources as it is currently using them, such that a Process is incapable of releasing the memory partition in which it is contained. For this reason, a signal is defined called SIGKILL which notifies the system Idle Thread that the Process wishes to be destroyed. The Idle Thread responds to that signal and performs the required clean up.

6.3.5 Create and Destroy Process Problem Model

Creating a process requires initializing subsystems which cross the boundary between which services are offered by the Kernel and which are offered by the Shell. For this reason, process initialization must be separated into two parts with an end point within the Kernel called IEnvironment, and an end point in the Shell called Crt0 (C RunTime – crt0 is the traditional runtime bootstrap binary object present in all C applications).

The act of destroying a Process is carried out by the Idle Thread which has no knowledge of any internally allocated resources. Those resources are internal to the Process and are considered part of the memory partition which houses the entire Process Virtual address space. Shared resources are used by multiple Processes and must be released cleanly so that the other Processes cannot access those resources. For this reason, all shared resources must be recorded by the RunTime environment.

6.3.6 Create Process Solution Model

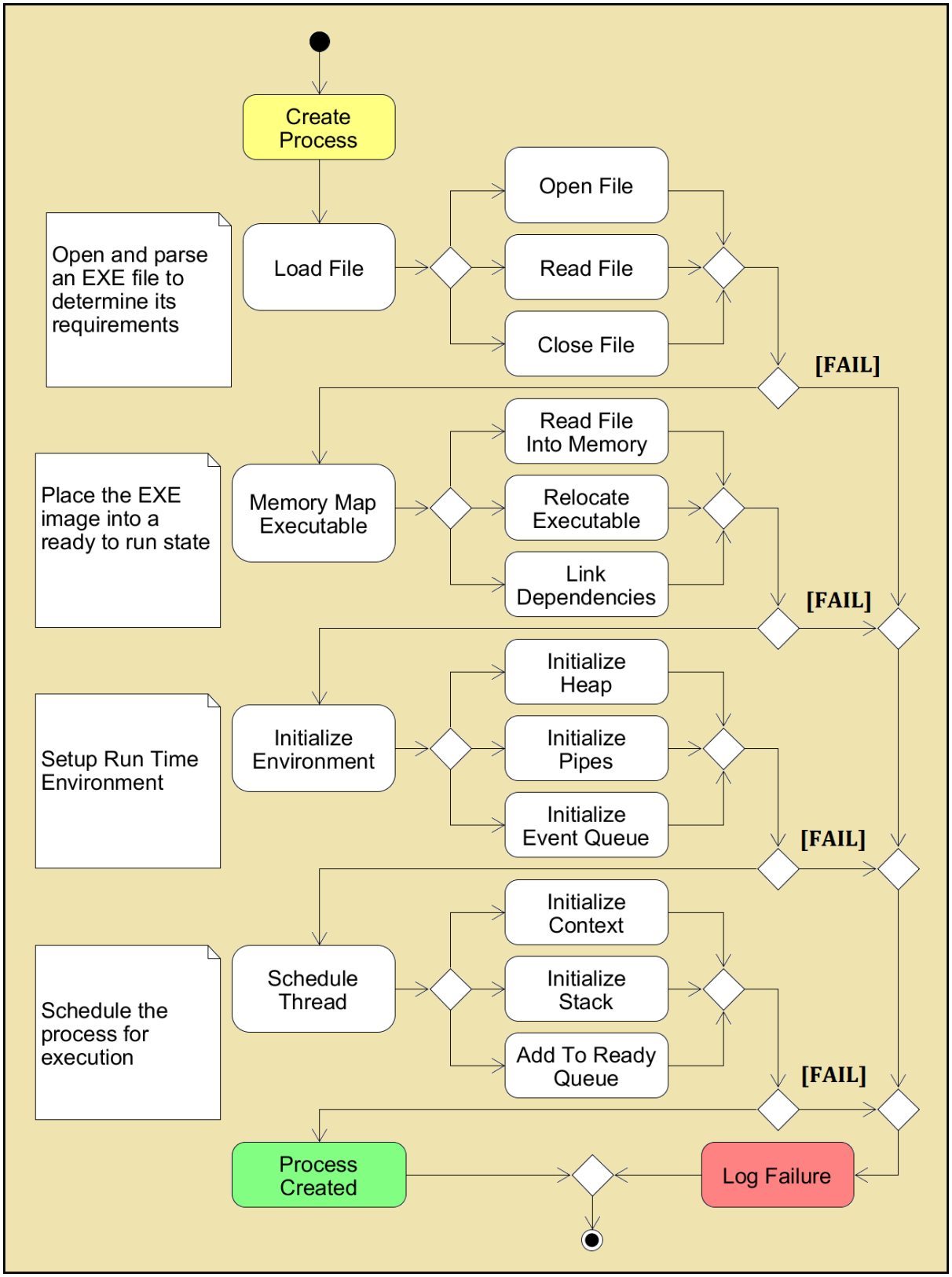


Figure 6.5: Create Process Activity Diagram.

6.3.6 Destroy Process Solution Model

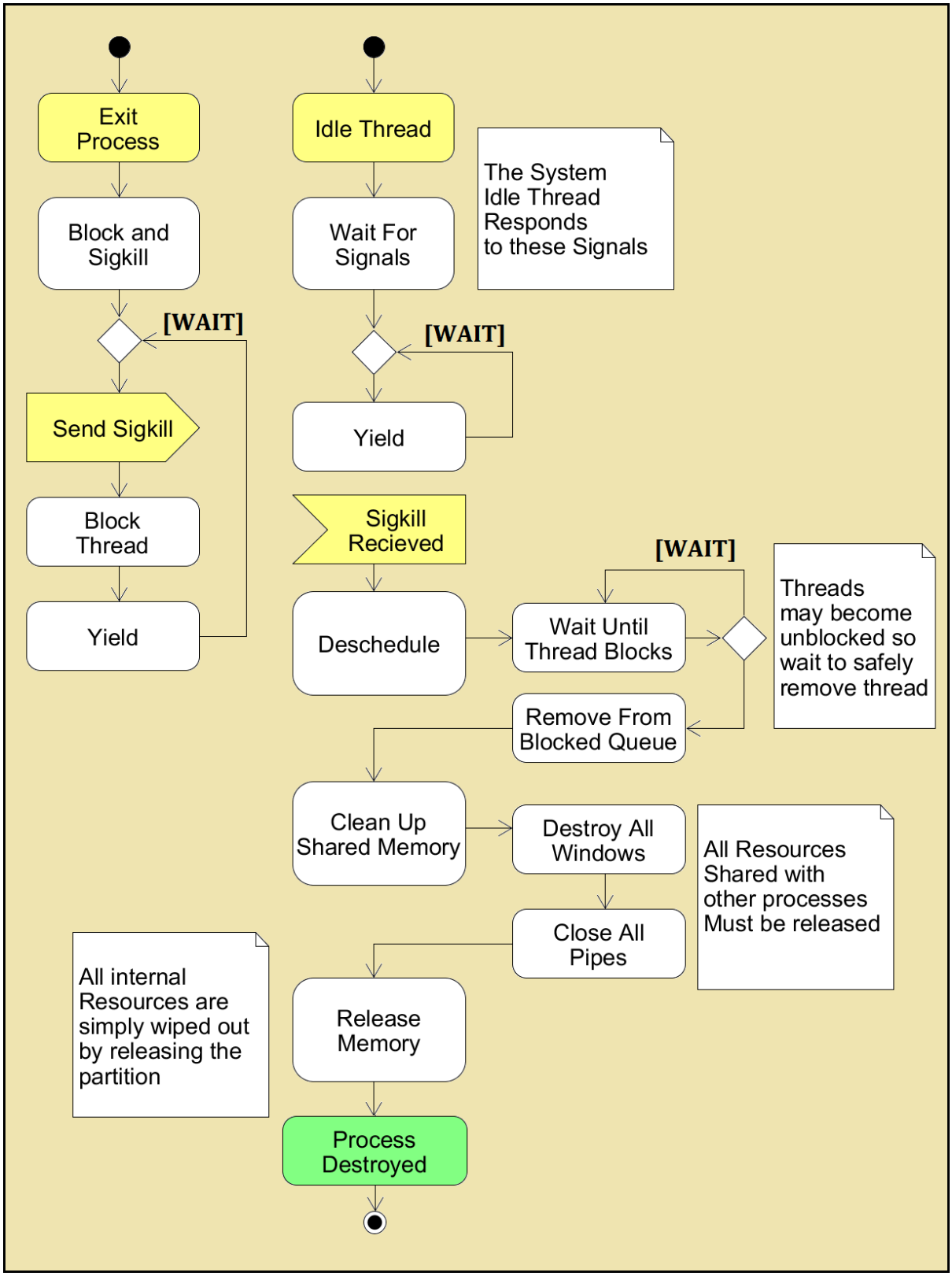


Figure 6.6: Destroy Process Activity Diagram.

6.3.7 Process Management Solution Model

The Process interface is the interface which the Kernel exposes to the rest of the system. It provides functionality for creating and destroying Processes. The activity diagram in figure 6.5 outlines the Process creation, and the activity diagram in figure 6.6 details the Process destruction. These operations are much more involved then would appear in those diagrams, and so this functionality has been split amongst multiple classes. Figure 6.7 depicts the Process class and two helper classes. The Loader class is responsible for reading the executable from the file system. It is also responsible for relocating the executable image so that all internal references are consistent with the new image base address. And the Loader class is also responsible for resolving external references through a process called linking. The Environment class is the Kernel end point for RunTime environment initialization.

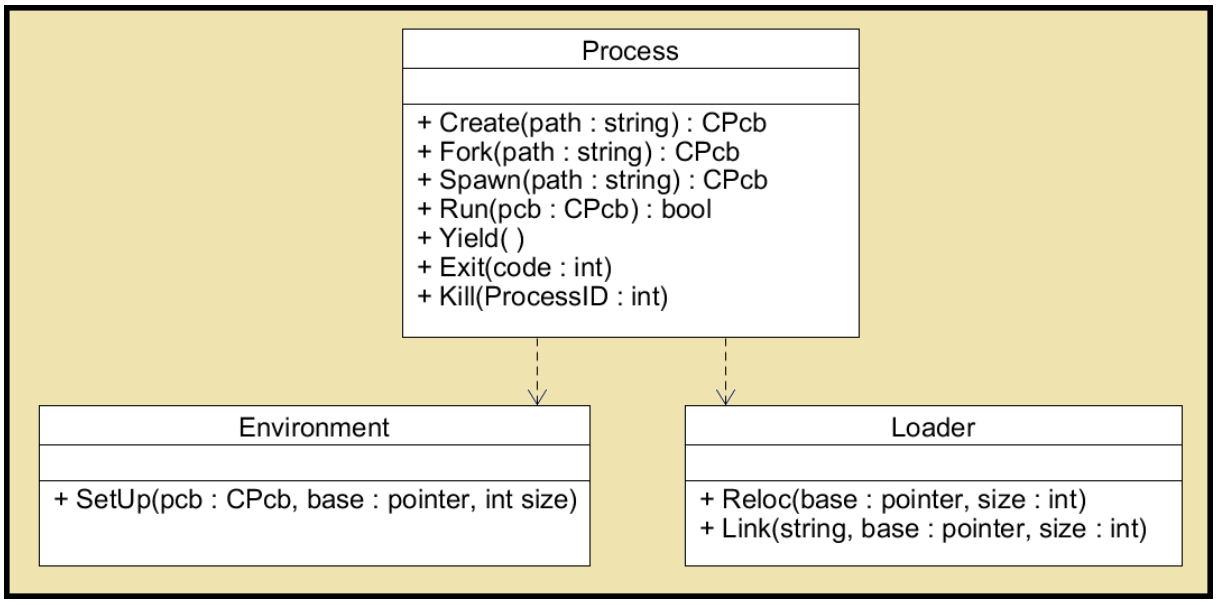


Figure 6.7: Process Interface Class Diagram.

Kill and Exit methods send a SIGKILL signal to destroy a Process. The Process Create, Fork and Spawn methods are used for Process Creation and differ in that Fork only creates a Process and initializes its RunTime environment, in addition to this Spawn also adds the Process to the Blocked Queue. Create does the same as Fork but also Runs the Process.

6.4 Device Management Subsystem

In the Spartan system, Device Management is achieved through the use of Pipes. Devices are controlled by Device Drivers which act as an intermediary between the Device and the rest of the system. These Device Drivers are exposed to the rest of the system as a Pipe end-point and communication with the Device can be achieved by writing to or reading from the associated Pipe. This section will restrain the discussion the design of the Piping subsystem and will not discuss the various protocols used to communicate with the various devices. Inquisitive readers wishing to study those protocols are deferred to the relevant documents, namely Intel (2018), IBM (1981, 1988, 1989, 1990), and VESA (2018).

6.4.1 Pipe Problem Analysis

Applications communicate with each other through Pipes. Since each Process has its own private virtual address space, and from Liedtke (1995) we know that this requires that both Processes share the memory assigned to a Pipe. Pipes have a read-end and a write-end, each end of the Pipe is given to an Application which acts as either the producer (write-end), or the consumer (read-end). By default, Pipes are blocking, if there is no space available to write to the Pipe then the producer is blocked. The act of reading from a Pipe will unblock the producer. The consumer will become blocked if they attempt to read from a Pipe which is empty. The act of writing to a Pipe will unblock a consumer. This describe normal blocking Pipes.

In addition to blocking Pipes, there are also non-blocking an asynchronous Pipes. A non-blocking Pipe will never block a producer of a consumer. If the producer writes to a non-blocking Pipe which is full, then the oldest bytes are discarded. If a consumer reads from an empty non-blocking Pipe, then the read fails and returns false. Asynchronous Pipes implement a call-back feature in which the producer and consumer install function pointers that will be invoked whenever data becomes available.

6.4.2 Pipe Problem Model

Pipes are a FIFO queue of bytes. In the Spartan system they are modelled as Rings which have a read head and a write tail. Figure 6.8 illustrates the main components which make up a Pipe highlighting the relationship between the head, the tail and the data contained within the Pipe.

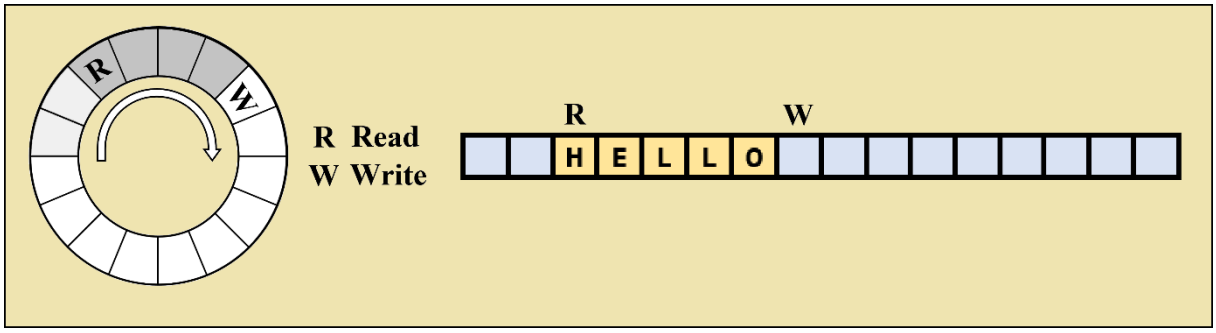


Figure 6.8: Piping Problem Model.

6.4.3 Pipe Solution Model

Pipes are basic objects but used extensively throughout the entire Spartan system. It is therefore essential that they are implemented as efficiently as possible. Since the memory used by a Pipe is shared memory, we are utilizing the zero-copy optimizations introduced by Liedtke (1995) for efficient Inter-Process Communication. Given the above problem model one can derive the solution model depicted in figure 6.9, which highlights the Pipes central role in the Inter-Process Communications subsystem.

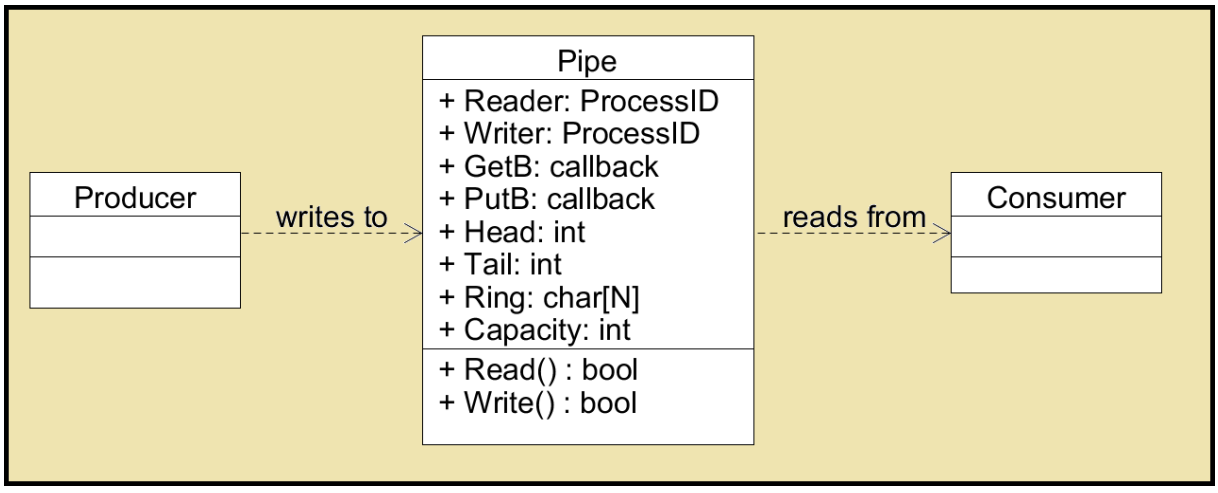


Figure 6.9: Piping Class Diagram.

6.5 BootLoad Subsystem

The BootLoad module is executed once at start up. This module is responsible for initializing the system and for establishing an effective execution environment. Most of the functionality the BootLoad module contains has been written to specification. These specifications have been noted within the source code with URL's directing the reader to the relevant documents. Figure 6.10 illustrates the BootLoad subsystem.

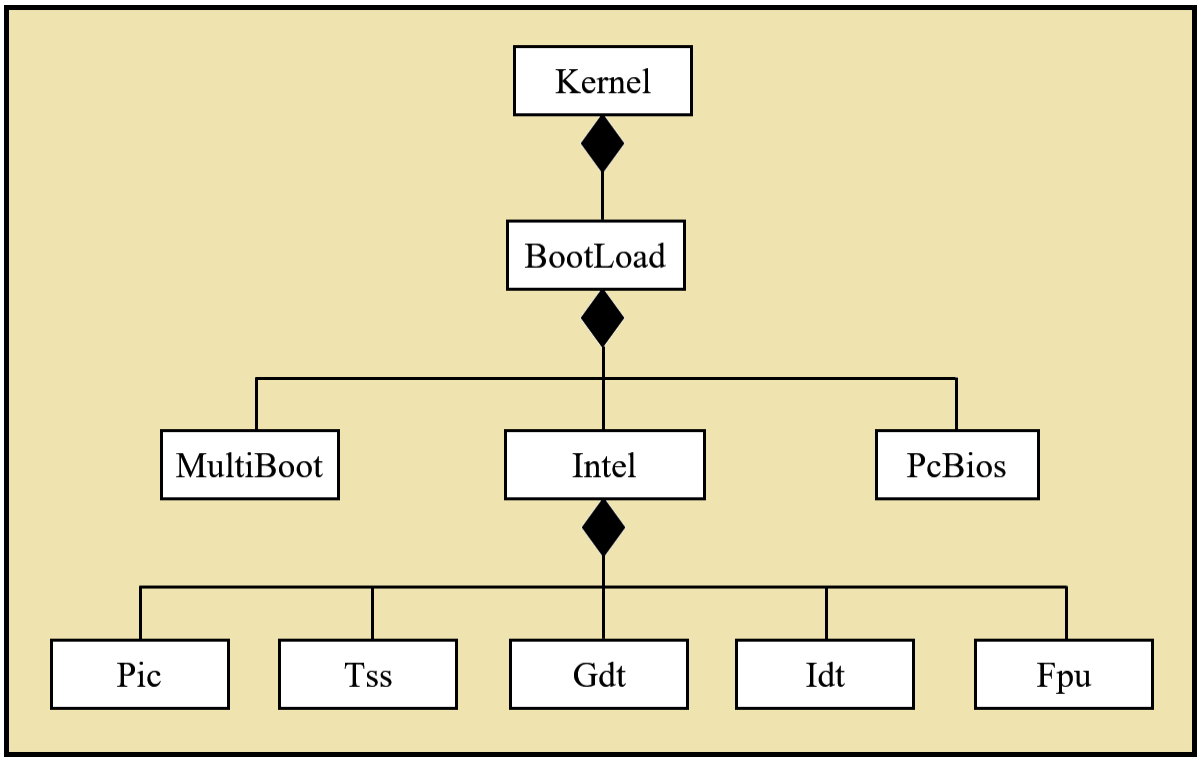


Figure 6.10: The BootLoad module and its subcomponents.

GRUB provides the Kernel with a MultiBoot header which contains information gathered by GRUB from the PC BIOS. The MultiBoot module parses this information. The Intel module is responsible for initializing the CPU itself and is the most involved component in the entire BootLoad subsystem. The Intel module has been subdivided into separate modules according features of the processor. The PcBios module is responsible for initialization related to the IBM PC BIOS specifications. The PcBios contains separate modules for each specification, which are not illustrated in figure 6.10 for clarity reasons.

6.6 Monitor Subsystem

The Monitor is a key component of the Spartan Kernel. It is responsible for error detection and recovery and allows the Kernel to recover from error states. This hardens the Spartan system and ensures that the system is protected from accidental or malicious corruption. The Monitor acts in a supporting role to the other Kernel components and is responsible for error recovery and logging system activity. This provides centralized system-wide logging and simplifies all other Kernel components which benefit from transparent logging and error recovery support. Figure 6.11 illustrates the Monitors central role in the Spartan Kernel, where the Monitor centralizes both logging and error recovery.

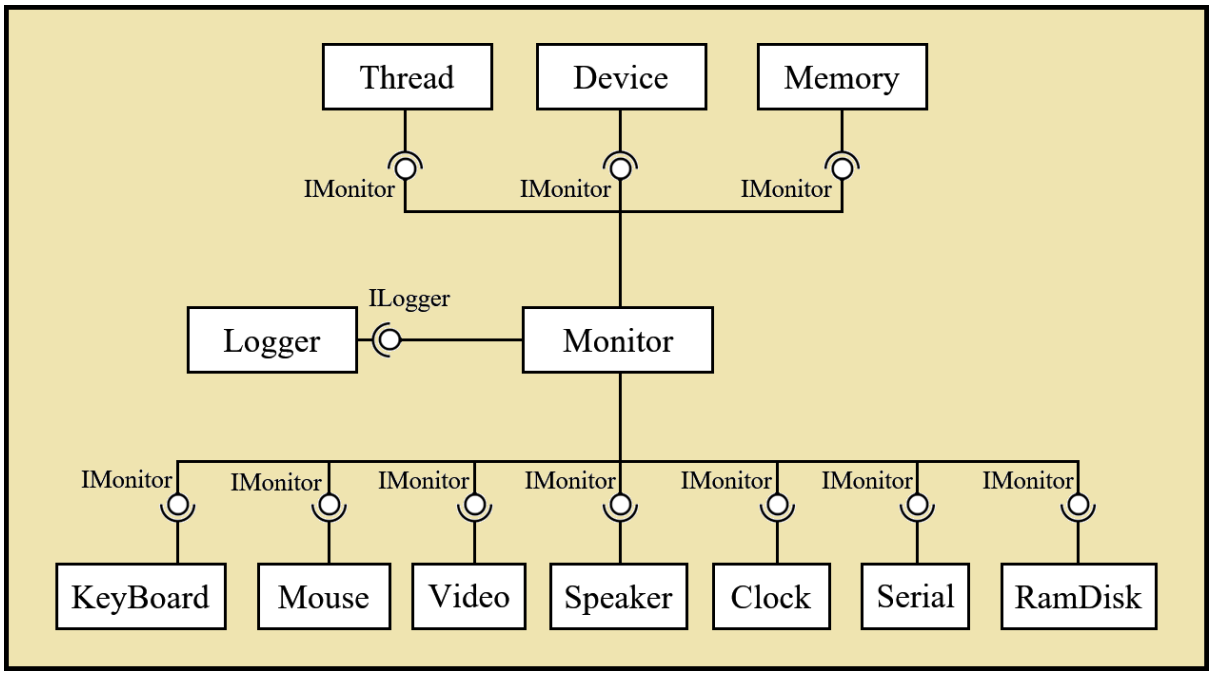


Figure 6.11: The Monitor servicing the Kernels Subcomponents.

6.6.1 Error Detection and Recovery Problem Analysis

The integrity of Kernel data is critical since the Kernel cannot simply crash and restart as though it were a user mode Application. The Monitor achieves error detection through cycle redundency checks, and error recovery is achieved through backups and restore points.

6.6.2 Cycle Redundancy Check (CRC) Problem Model

A good Cycle Redundancy Check (CRC) is effective at spreading the influence of bits throughout the checksum. To achieve this the system uses 2x2 matrix multiplication and bitwise rotates. Figure 6.12 illustrates how matrix multiplication spreads the influence of bits within a four-byte group. Bitwise rotates and exclusive-or is used to retain these bits so that they have an accumulative effect over the entire buffer. Taken together this approach ensures that flipping a single bit will affect all 32 bits of the checksum.

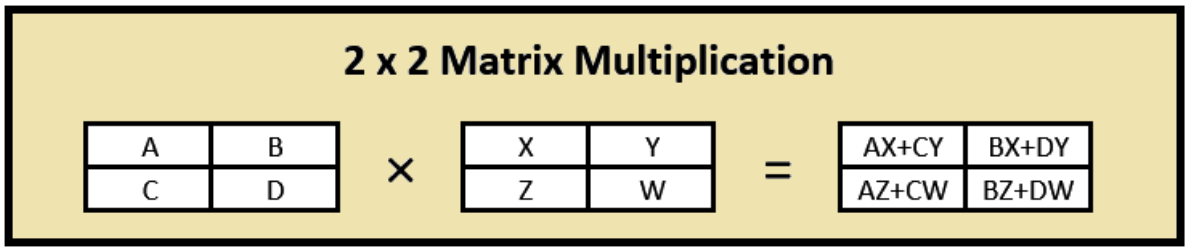


Figure 6.12: Matrix multiplication.

To calculate a checksum the system must interchange how it represents the data buffer and the checksum. When it is performing bitwise rotates, items are treated are 32-bit integers. Matrix multiplication treats each item as a 4-element char array. This process is illustrated in figure 6.13, where the same 32-bit unit of data is represented in various forms. In addition to this data representation, the Monitor also maintains an associative array which names modules as "owning" a region of memory.

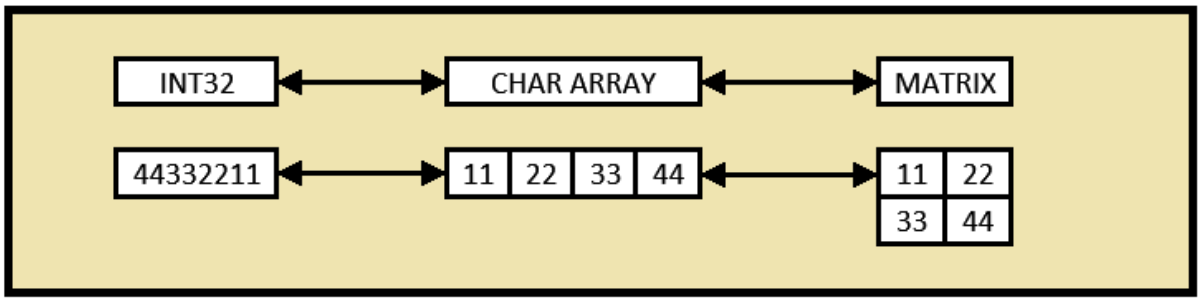


Figure 6.13: Mixing data representations for Cycle Redundancy Checks.

6.6.3 Error Detection and Recovery Solution Model

Kernel modules are realised as singleton interfaces which process associated data. The Monitor is responsible for assigning ownership of memory to these modules. Modules request access to data, and in response to such requests the Monitor calculates a checksum to verify the integrity of the data, if the data is valid then it is presented to the module. When processing has completed, the module returns the data to the Monitor. The check sum is recalculated, and the data is duplicated to create a restore point. Figure 6.14 illustrates the sequence of steps taken by the Monitor to verify the integrity of a region of memory, and to recover from errors if necessary.

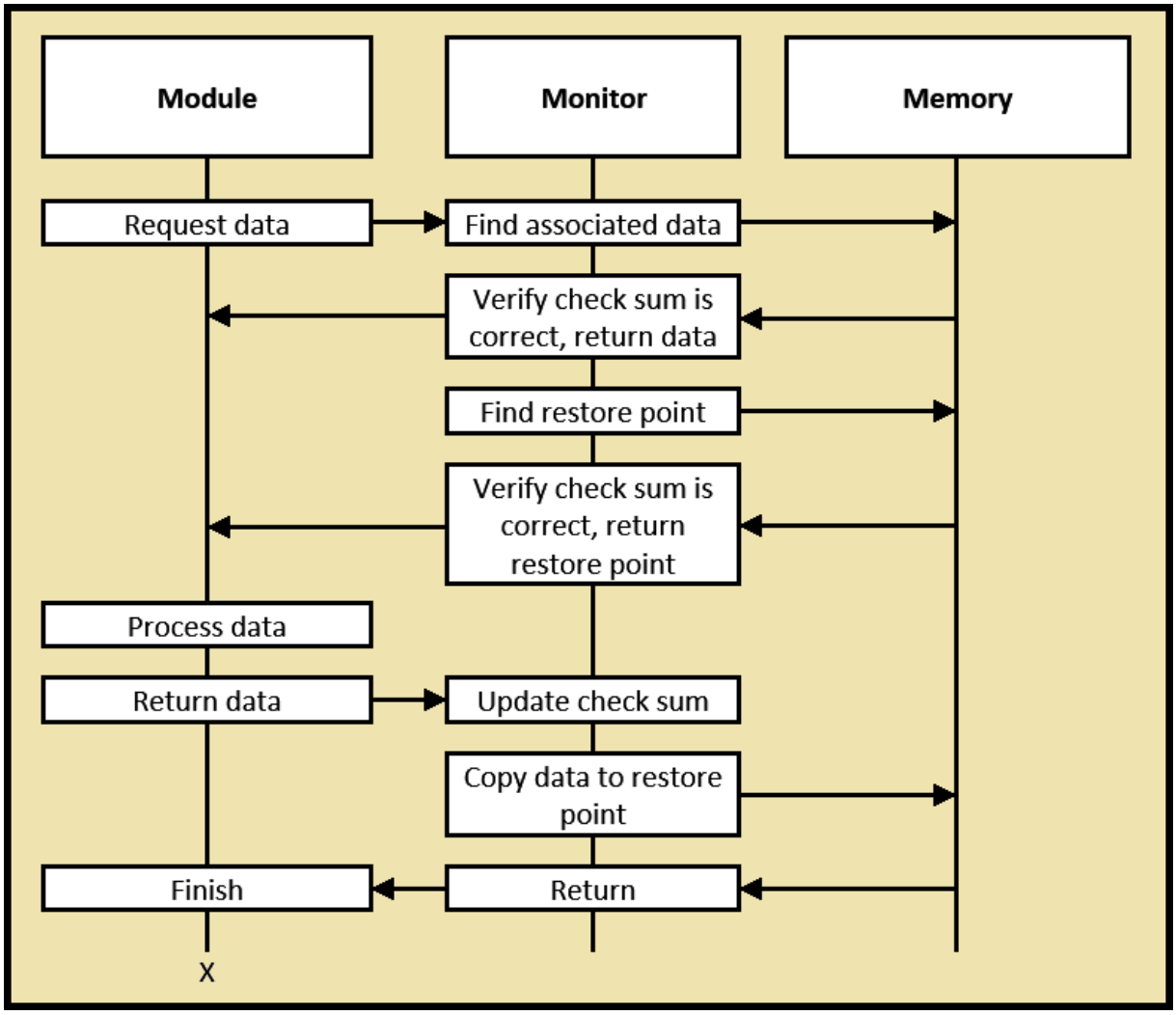


Figure 6.14: Error Recovery Sequence Diagram.

6.7 Rendering Pipeline Subsystem

The term “Rendering” describes the act of converting an image into pixels. The flow of pixels from the applications drawing buffer to the screen is called the “Rendering Pipeline”. Responsiveness is one of the non-functional requirements of the Spartan system. This section will address this requirement beginning with an analysis of monitor refresh rates and outlining the problems which must be overcome to ensure the system maintains a responsive feel.

6.7.1 Rendering Pipeline Problem Analysis

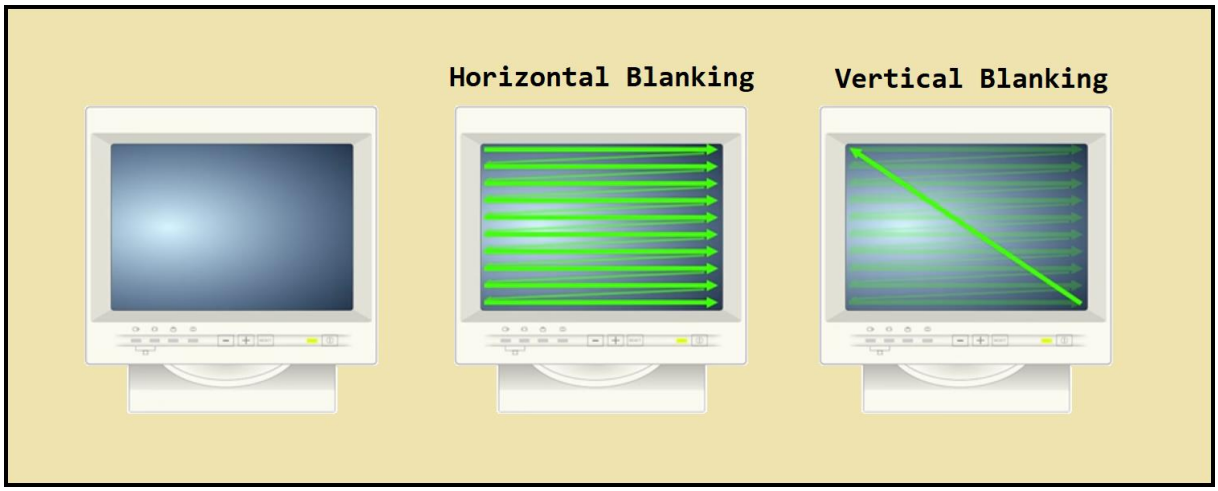


Figure 6.15: Horizontal and Vertical Blanking intervals.

Monitor refresh rates are applicable to VESA displays and refer to the speed at which the screen is updated. Figure 6.15 illustrates the refresh rate of a Cathode Ray Tube (CRT) monitor. A CRT has an electron gun which traces an image in scanlines from left to right, from the top of the screen to the bottom. This produces a charge which excites the phosphor dots on the screen, causing them to illuminate. When the beam completes a scanline, it returns to the left edge of the screen to trace the next scanline, this period is called horizontal blanking. When the beam reaches the bottom of the screen it travels diagonally back to the start of the first visible scanline. This period is called vertical blanking and it occurs at regular intervals, for example 60 times per second which gives a refresh rate of 60 Hz.

It is desirable to synchronize updates to the vertical blanking interval. This avoids unwanted rendering artefacts such a tearing and flickering. These artefacts are illustrated in figure 6.16, they can cause eye strain and at best appear displeasing to the eye.

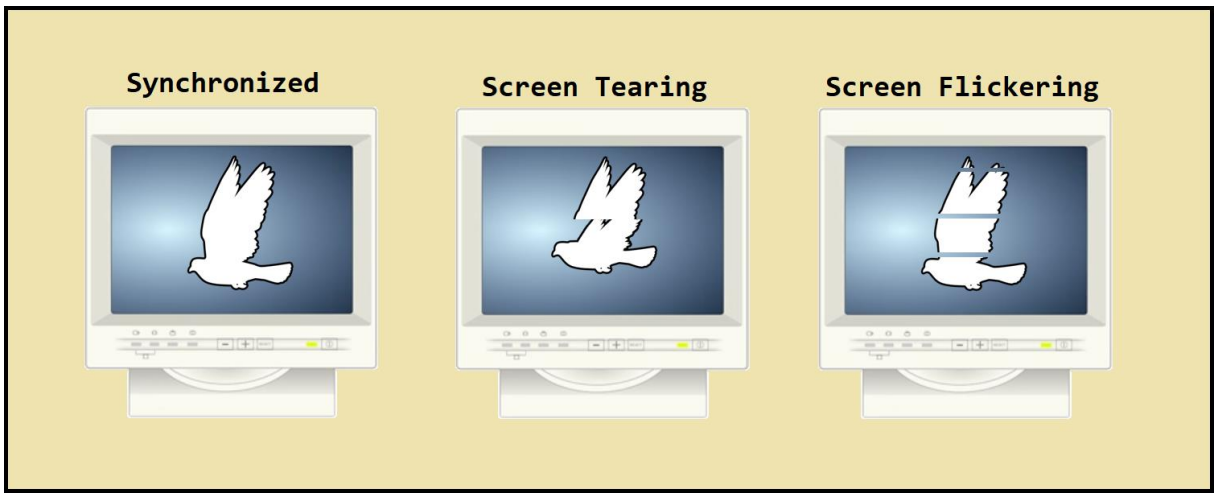


Figure 6.16: Rendering Artefacts.

Thirty frames per second appears to the human eye as continuous motion. However, the difference between thirty frames per second and sixty frames per second is perceptible to a human. To maintain responsiveness the screen must be updated at sixty frames per second, this is an ambitious target when everything must be performed in software. The Spartan system relies on the VESA standard and does not benefit from the hardware acceleration which a GPU provides. However, maintaining 60 frames per second is attainable if the resolution is lowered.

The throughput of the Rendering Pipeline is memory limited. The memory which is updated happens to be memory mapped IO, which is slower than writing to RAM. If the resolution of the video mode is 800 x 600 and each pixel is 32-bits, then the system needs to transfer 1,920,000 bytes per frame. This requires a bandwidth of 115,200,000 bytes per second to maintain a refresh rate of 60 Hz, that's almost 110 MB per second. Also, the system must not consume too much of the CPU time or else there would be no time for the applications to update their images, and the system would appear to be much slower than 60 Hz.

The display adapters memory mapped IO is called the Framebuffer. When an application renders a frame, it sends it to the Framebuffer and the display adapter displays it on the screen. This simple approach is called single buffering and is illustrated in figure 6.17. This approach can cause tearing or flickering if the application cannot keep pace with the monitors refresh rate. This is because the application is drawing to the frame as it is being displayed.

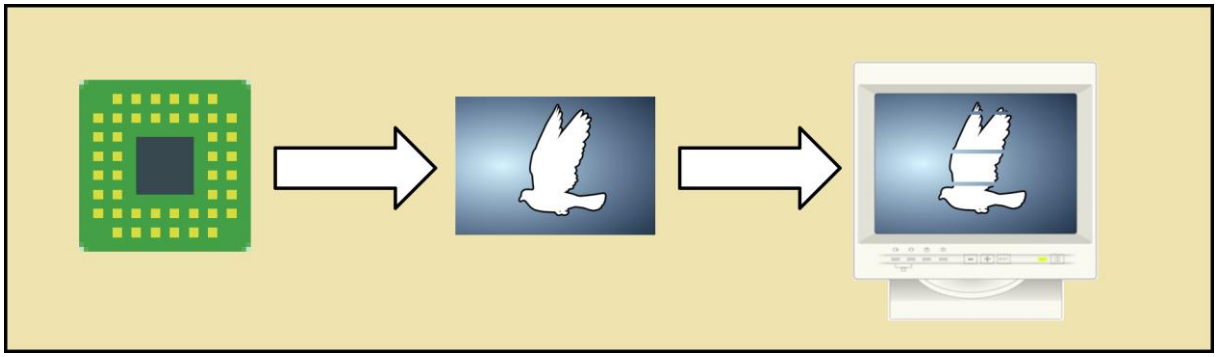


Figure 6.17: Single Buffering

In order to avoid artefacts, an application can use a second buffer to draw the next frame. This approach is called double buffering and is depicted in figure 6.18. Double buffering avoids artefacts by always providing the display adapter with a completed frame. However, when the application finishes drawing the next frame, it must wait for the display adapter to finish displaying the current frame before it can swap buffers and begin drawing the next (third) frame. This is wasteful as the application is forced to wait while the display adapter is busy.

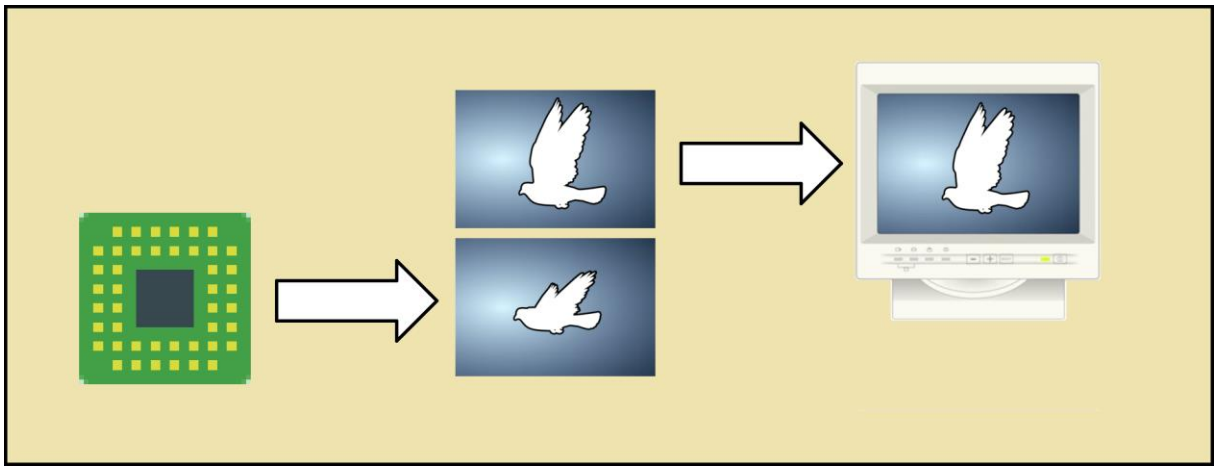


Figure 6.18: Double Buffering

Graphics-oriented applications, such as video games, tend to spend most of their processing time in the rendering phase. When these applications experience intensive load, then they might struggle to keep pace with the monitors refresh rate. If the application cannot finish drawing the frame when vertical blanking occurs, then the display adapter will redraw the previous frame a second time. When the next frame is ready, the application cannot send it to the display adapter and is forced to wait for the next vertical blanking interval. This application is likely to consistently miss the vertical blanking period and will always have to wait for the next frame, halving the frame rate and producing 30 frames per second.

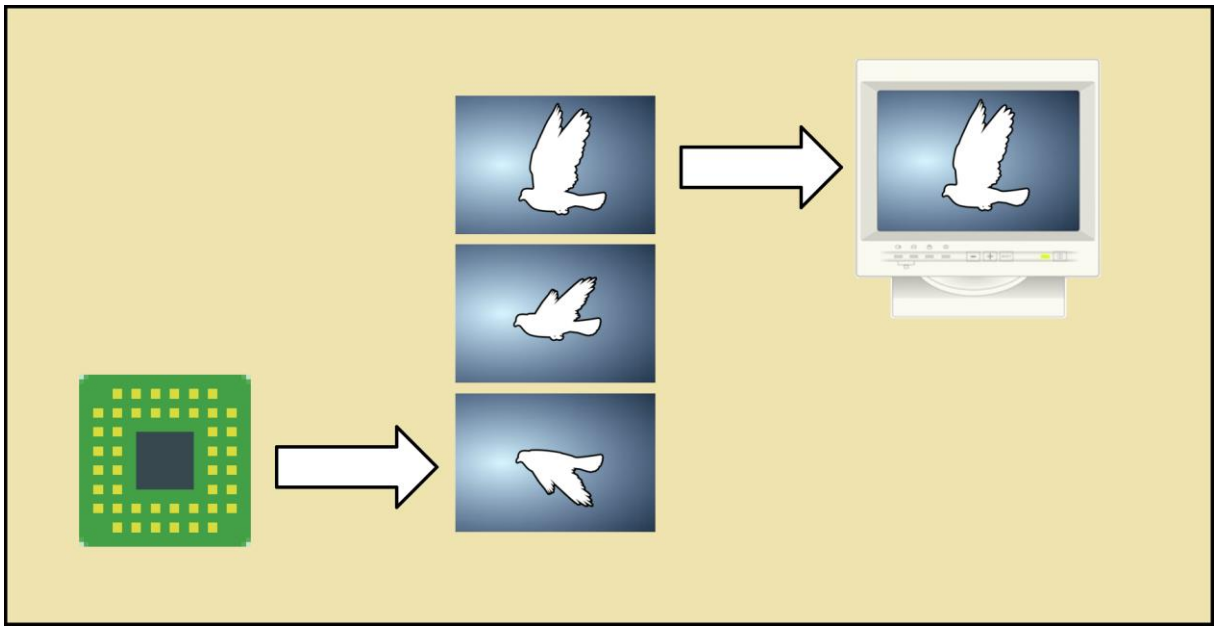


Figure 6.19 Triple Buffering

This can be avoided by introducing a third buffer, so that there is always one frame being drawn, one frame being displayed, and a tertiary buffer which is ready to either be displayed or to be updated. Triple buffering is depicted in figure 6.19 and is much fastest then the previous approaches, it allows the application to get a head start drawing the next frame but introduces a single frame of latency. It is more important that applications can produce 60 frames per second, then it is to worry about one sixtieth of a second latency. And for this reason, the Spartan system uses triple buffering as the basis for its Rendering Pipeline.

6.7.2 Rendering Pipeline Problem Model

The Spartan Rendering Pipeline manipulates drawing buffers called Canvases. A Canvas is an aggregate abstraction which implements the “Dirty Regions” concept. A Canvas is a pixel buffer which is decomposed into a grid of rectangles. The Canvas contains a plurality of Windows which may overlap. The size and positions of these Windows determine the topology of the rectangular grid. Each Application has its own private Canvas into which it draws its Windows. The system maintains a shared Canvas called the Framebuffer which merges together into a single frame the Canvases from each of the Applications. Figure 6.20 highlights the main components of the Canvas, notice that Window areas overlap within the Canvas but not within grid cells. This arrangement allows for efficient pixel transfers and accomodates zero copy operations used by the rest of the Rendering Pipeline.

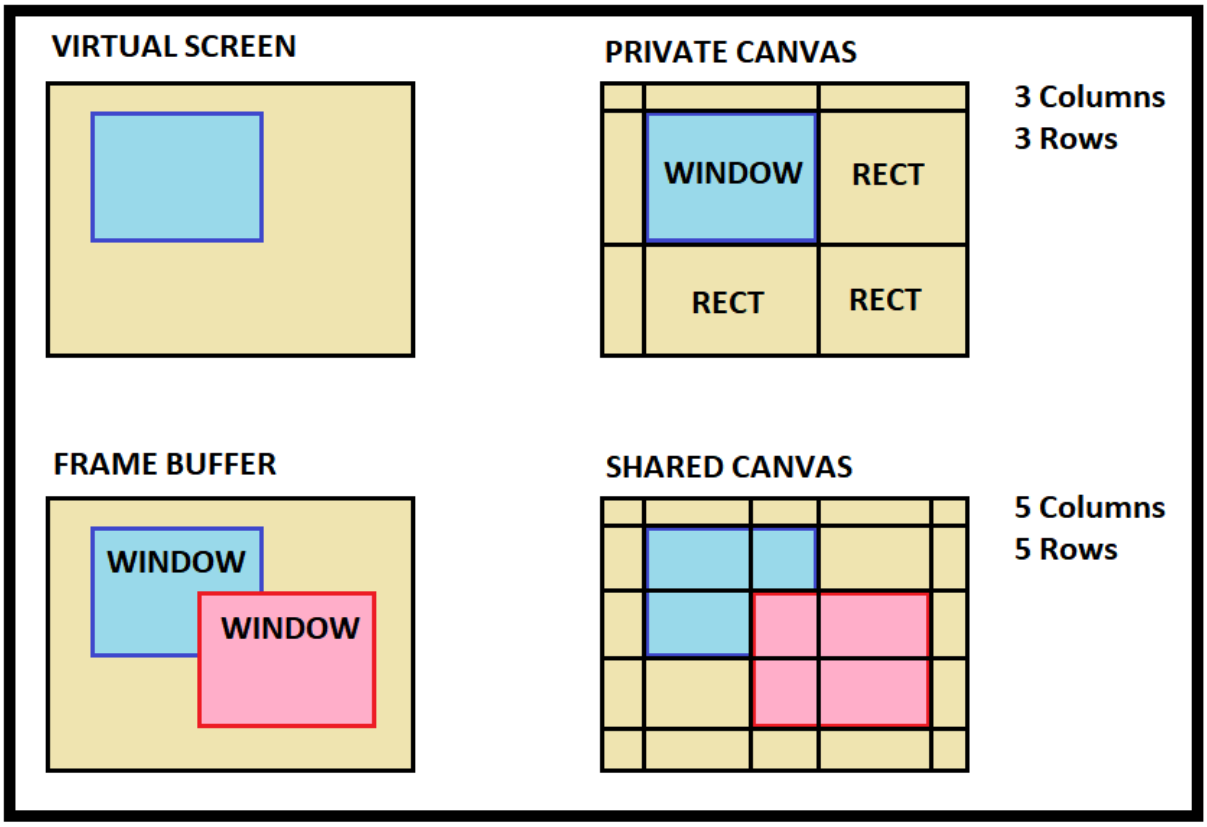


Figure 6.20: The Canvas Problem Model.

In the Rendering Pipeline, Applications assume the role of producers and a system task called the Compositor acts as the consumer. The Compositor has the responsibility of merging the drawing buffers from each application into a single frame which is then displayed on the screen. This arrangement is illustrated in figure 6.21 where three applications are producing frames which are merged by the compositor and dispatched to the display adapter.

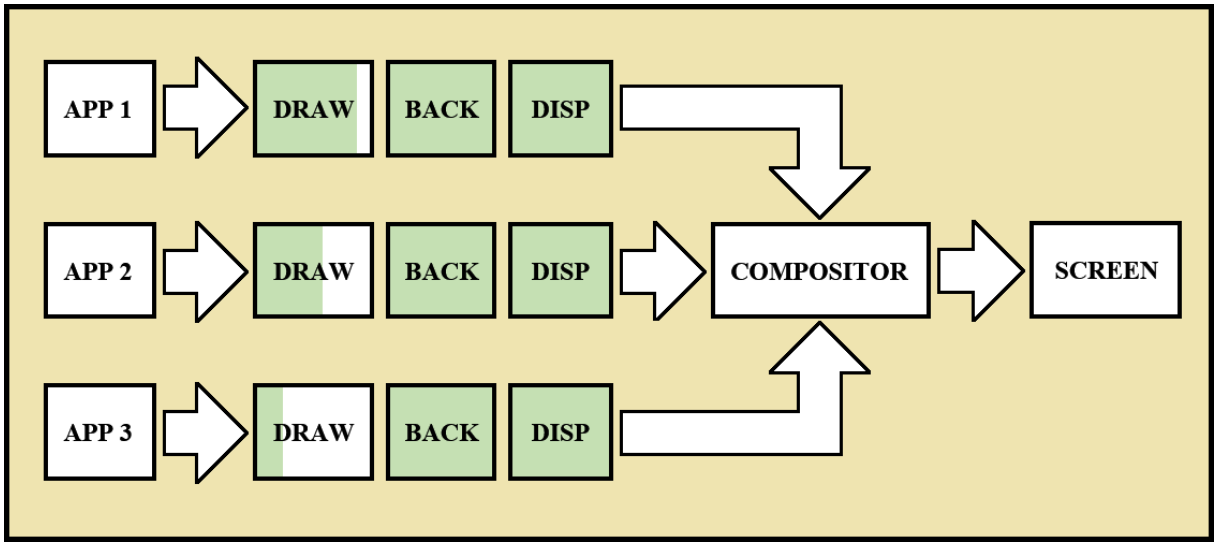


Figure 6.21: The Compositors role in the Rendering Pipeline.

The Compositor updates the Framebuffer at a rate of 60 Hz, it processes each frame one scanline at a time and synchronizes on each horizontal blanking interval. This synchronization is crucial and should be invariant to the load experienced by the Process Scheduler, so an alternative periodic timer is used: the PIT (Programmable Interval Timer). During each interval the Compositor will transfer one scanline, then will yield the remaining time back to the system. When a given scanline has not been updated, then the entire time slice is yielded.

When the video mode is 800 x 600, then the Compositor must be capable of transferring 600 scanlines 60 times a second for a total of 36,000 scanlines. The PIT has a base frequency of 1,193,180 Hz which can be lowered to 36,157 Hz. At this frequency the Compositor can synchronize to the horizontal blanking interval by skipping every 229th signal received from the PIT. This gives control over the update frequency which is invariant to overall system load.

When an application finishes drawing to its Canvas it sends a request to the Compositor. The Compositor is likely to be busy displaying the current frame and is incapable of responding immediately to the request, so it is added to a queue and will be processed later, when the Compositor is not busy. The requests in the queue can involve actions such as moving or resizing the Windows contained within the Canvas. To provide support for these operations, the requests are encoded as opcodes. When the Compositor gets a chance to process these requests then it invokes a virtual machine called the Renderer which decodes the opcodes and performs the requested operations.

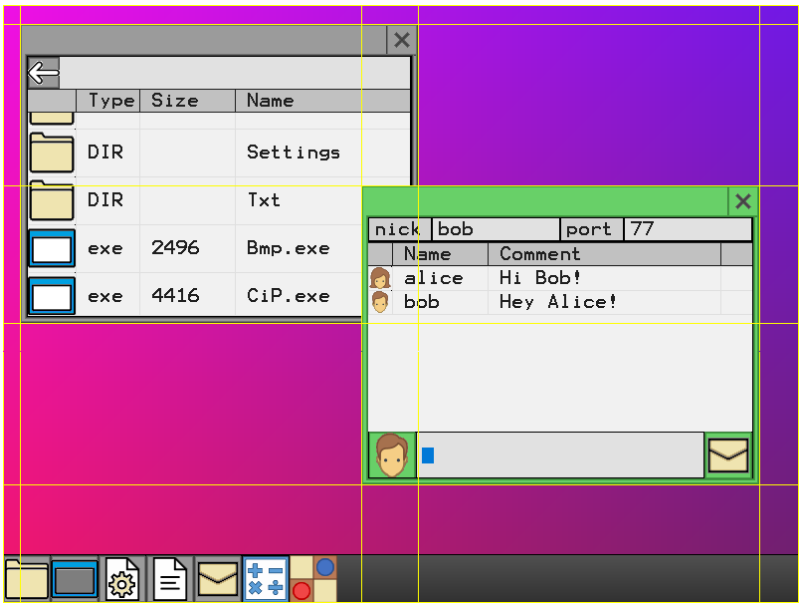


Figure 6.22: Framebuffer with overlapping Windows.

As can be seen in figure 6.22, Windows can overlap within the Canvas but not within the grid of rectangles. This allows the Renderer to treat each rectangle as a unit which can be manipulated independently without moving the original pixels in the Canvas. Since the Compositor is already fetching pixels from the Canvas and transferring them to the Framebuffer we can modify one of those pointers to get the Compositor to perform the requested operations for free. This allows the Renderer to process requests using zero copy operations by leveraging the work which would already be performed by the Compositor.

6.7.3 Rendering Pipeline Solution Model

The Rendering Pipeline is made up of a Compositor, a Renderer, a plurality of private Canvases, and a single shared Canvas called the Frame Buffer. Each Canvas is made up of a plurality of Windows and many Rectangles. Figure 6.23 illustrates the composition of the Rendering Pipeline and highlights the relationship between these components.

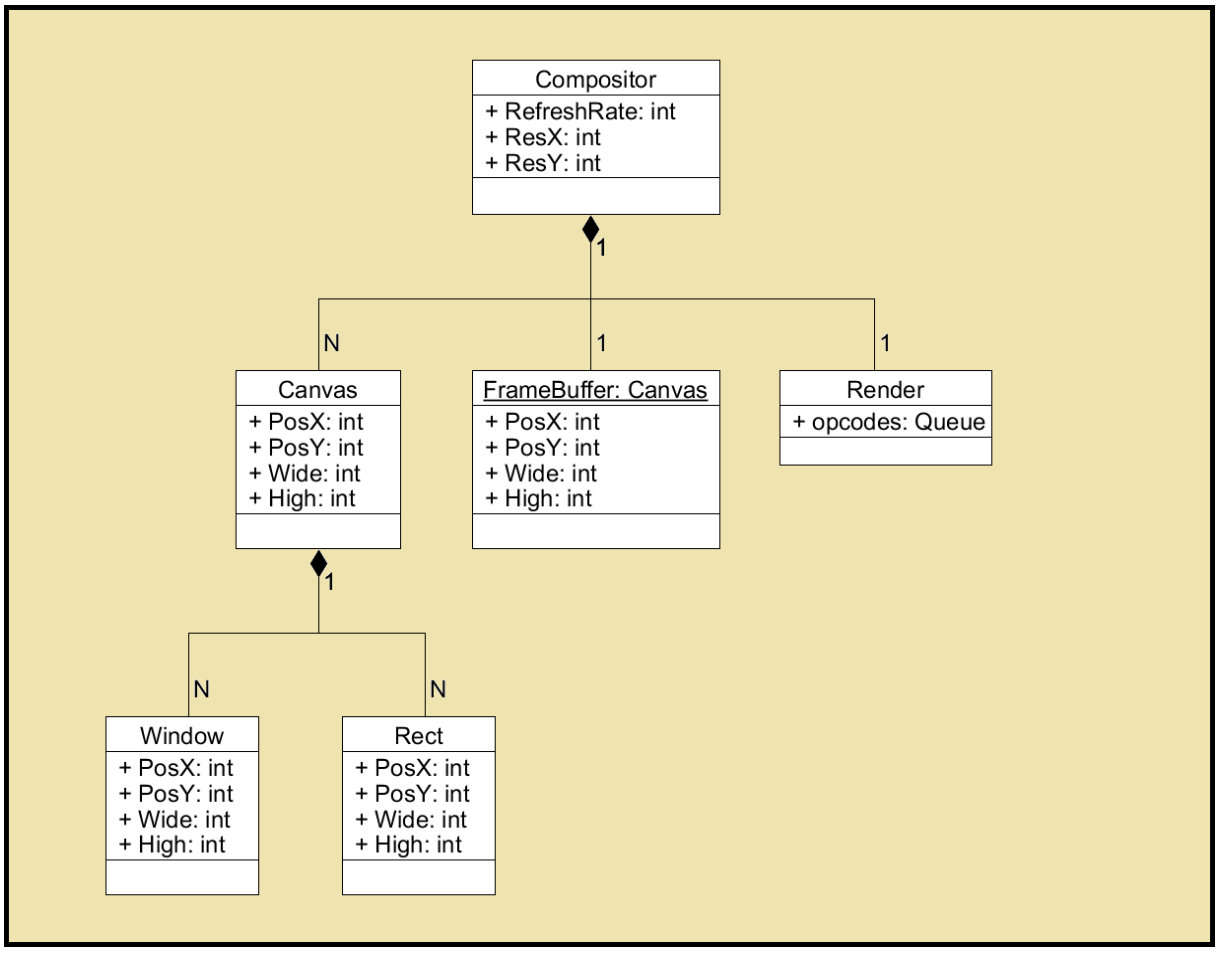


Figure 6.23: Rendering Pipeline Class Diagram.

It is important to note that the private Canvases must exist in shared memory. These regions of memory are accessed by the Applications themselves and the Compositor. As with all shared memory, it is necessary to ensure that this memory is freed accordingly upon the destruction of an Application. Therefore the Compositor must be designed such that any Canvas can be removed at any point in time.

6.8 Windowing Subsystem

Windows are the embodiment of an Application in a Graphical User Interface. Windows are rectangular regions of screen space which contain objects called Widgets which the user may interact with. A Graphical User Interface presents a point and click interface to the user which is more natural and intuitive then the historic command-line interface. This section outlines the Spartan Windowing system and the technologies which drive the user experience.

6.8.1 Windowing Problem Analysis

The screen can contain any number of Windows and these Windows may overlap within the screen. In a Stacked Windowing System there is a stack which maintains the z-order of the Windows. The topmost Window acts as a single point of interaction for the user and all other Windows are effectively put to sleep until the user chooses to bring them to the topmost position. The Window in the topmost position is said to have the focus and all keyboard and mouse input are directed to this Window. This simplifies the process of device IO redirection, which was historically dealt with through a process called multiplexing. In a Stacked Windowing System there is no need for multiplexing.

Windows contain Widgets which are objects such as Buttons, Text Boxes and Scroll Bars. A Widget may contain a number of child Widgets and will act as their parent. Child Widgets may inherit the behaviours and properties of their parent. The user may interact with Widgets by pointing and clicking. Events such as pressing a key on the keyboard or moving the mouse cursor will result in an Event being raised which is added to an Event Queue. Each Application monitors its own Event Queue and will dispatch the Event to the intended Widget for processing. Widgets may also raise their own Events such as the “On-Action Event” which may prompt a reaction from other Widgets. If a Widget does not handle a given Event, then the Event is processed by a default Event handler.

6.7.2 Windowing Problem Model

It can be seen from the Windowing problem analysis that the Windowing system requires: A Window Stack, an Event Queue, a collection of default Event Handlers, a collection of Widgets, and a means by which a Widget may inherit the properties of their parent. The Window Stack and Event Queue are implemented using the typical FILO and FIFO data structures as depicted in figure 6.24. Windows are inserted (pushed) to the topmost position of the Window Stack but can be popped from any position within the Window Stack. The Event Queue contains a sequence of Events and is processed in FIFO order. The real difficulty with implementing an effective collection of Event Handlers is in converting basic mouse click and mouse move Events into something more meaningful such as a resize Window Event. There is unlimited scope for refinements in the area of Event Handling and describing these components in detail is a task which warrants its own design document and are not describe herein.

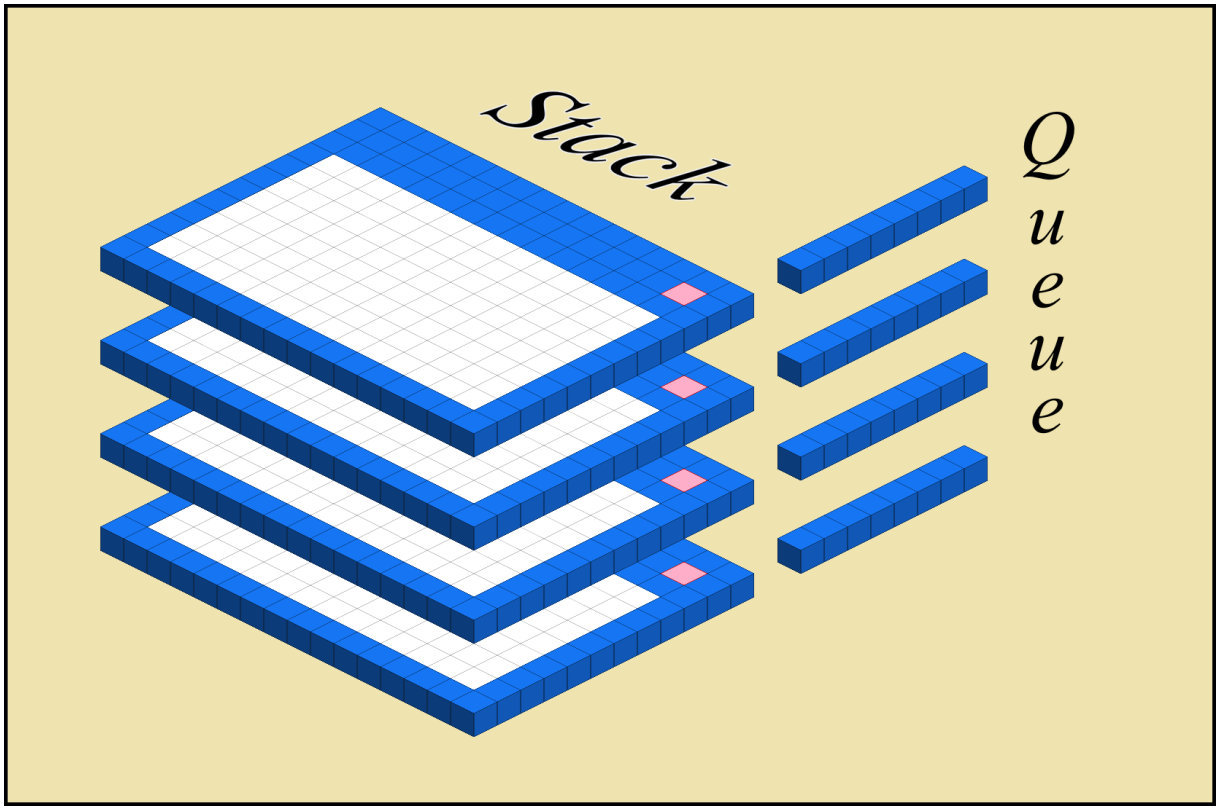


Figure 6.24: Window Stack and Event Queue Problem Model

6.7.3 Windowing Solution Model

The Windowing system is made up of Widgets and Events. Widgets are a form of Window in that they are represented as a rectangle of pixels. Widgets differ from Windows in that they have behaviours which react to Events such as mouse clicks, and also Widgets have custom properties which support these behaviours. An Event is an abstract base class which are generic messages that are passed through the Event Queue. Events can take many forms with some examples being the OnLoad, OnQuit, OnSize, OnDraw and OnClick events. Figure 6.25 outlines the relationship between Widgets and Windows, and between the abstract base Event class and some of the possible derived Event classes. Both the Widget and Event classes are examples where polymorphism is not only beneficial but essential to the design and modelling of a manageable and extensible system.

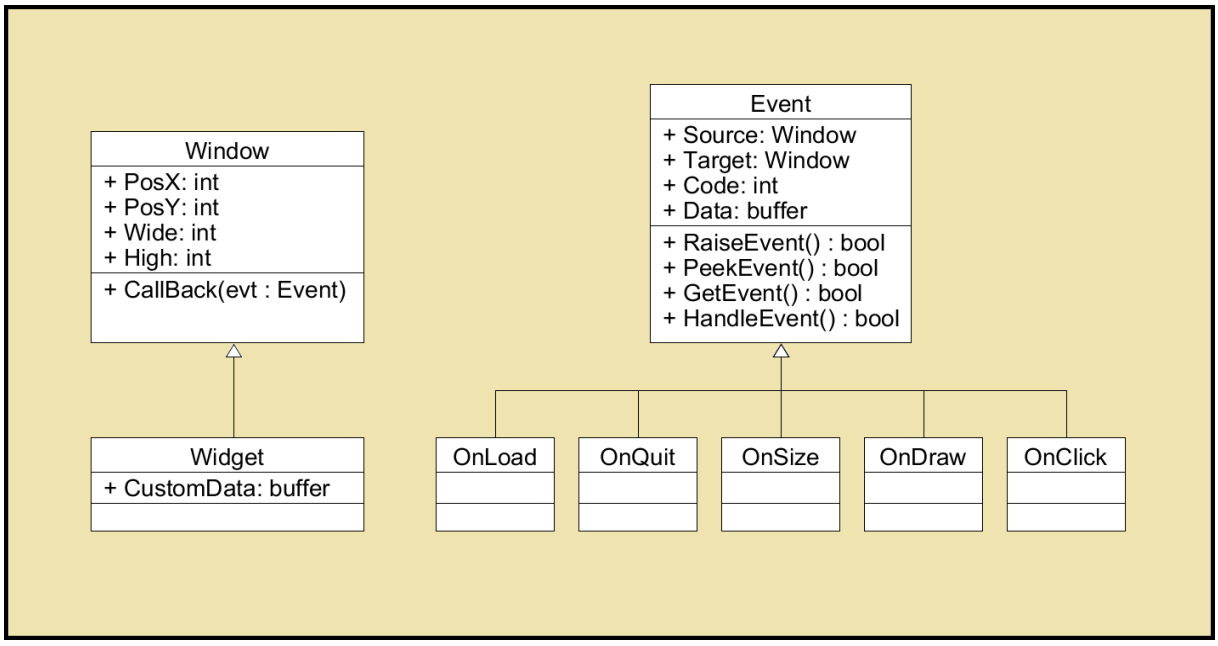


Figure 6.25: Window System Class Diagram.

This concludes the detailed design. The remainder of this chapter shall present some screenshots from the finished product which illustrate the wide variety of Widgets which are possible in the Spartan system.

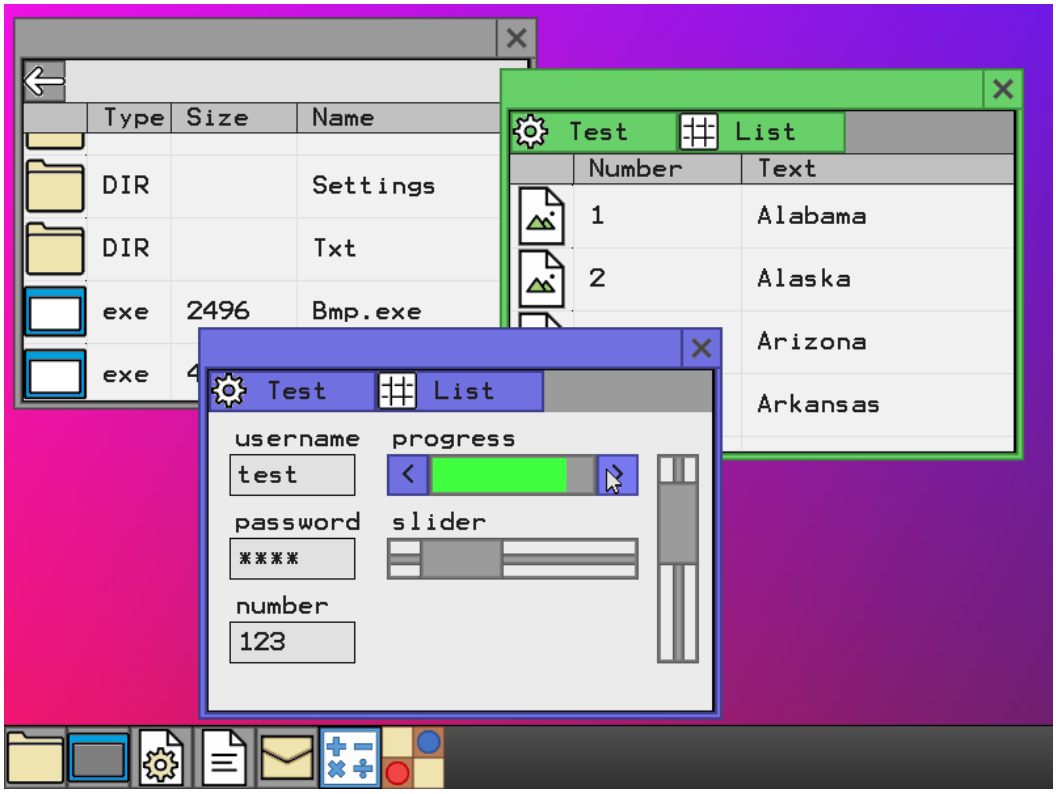


Figure 6.26: Various Widgets such as Progress Bars, Text Boxes and List Views.

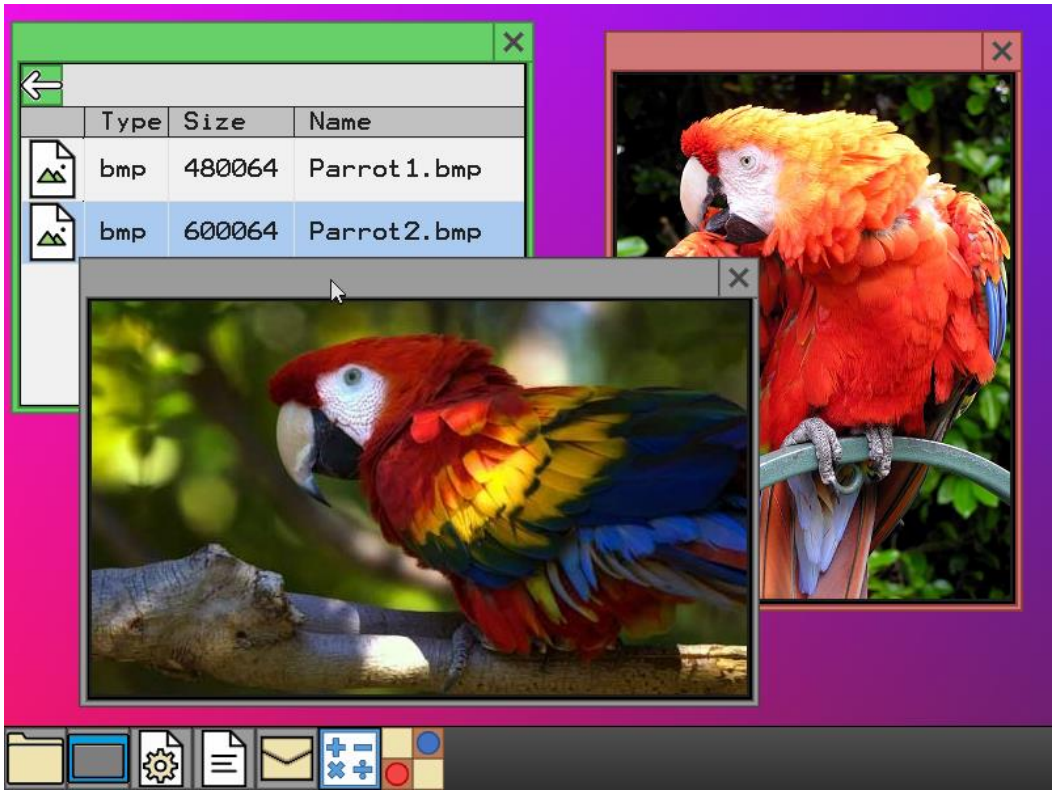


Figure 6.27: File Associations where BMP files are opened with the image viewer.

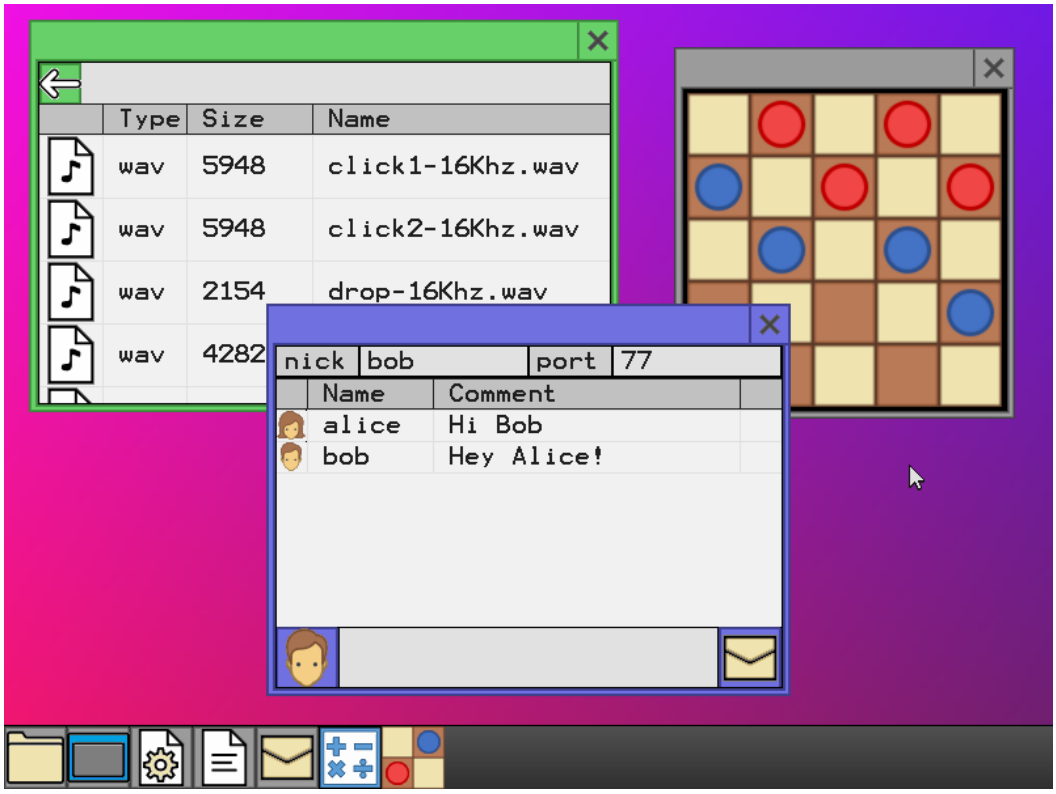


Figure 6.28: Multi-Tasking with three Applications running simultaneously.

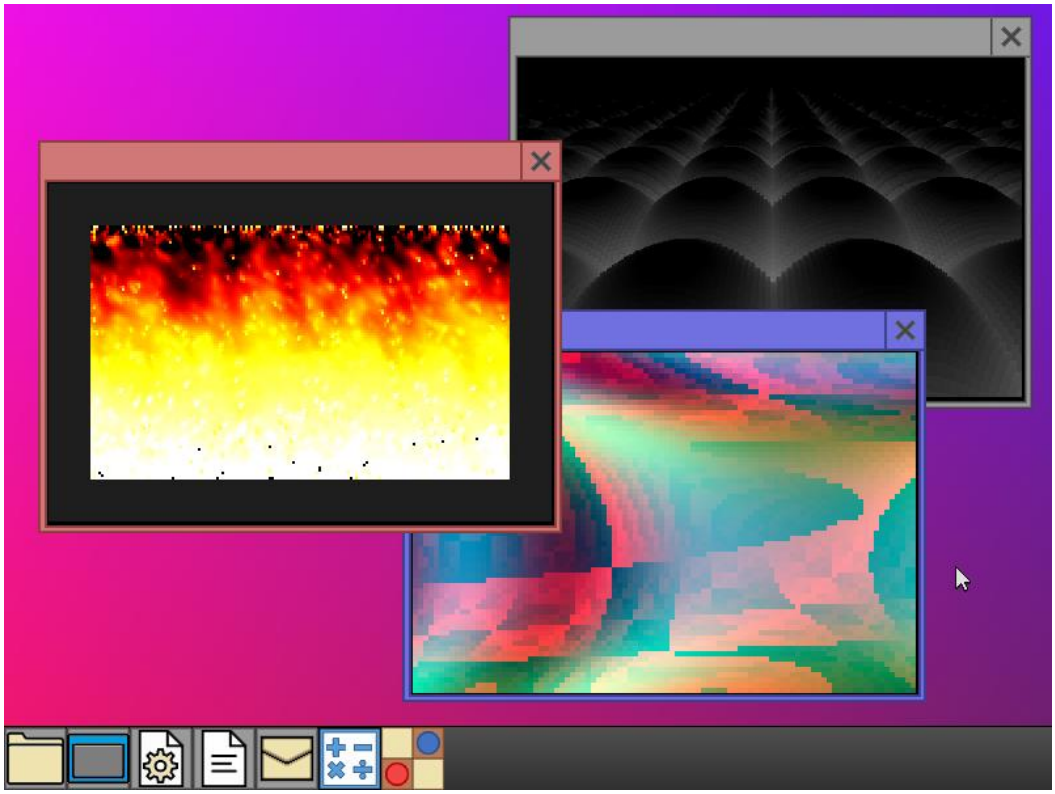


Figure 6.29: Multi-Media Applications stress testing the Rendering Pipeline.

Chapter 7 – Conclusions

This project began from a truly blank canvas and delivered an operating system which features a Graphical User Interface as its primary form of Human-Machine Interaction. It is uncommon for modern software engineers to begin from a truly blank slate. This endeavour has shown that anything is possible, and that an engineer should not be daunted by a lack of support systems but should enjoy the opportunities which a blank canvas provides. The author feels that decades of abstractions have buried not only the details but also the original sense of adventure that inspired developers to embark on this career path. There are few, if any, projects which can match the educational value inherent in operating system development. The author feels that this is an exercise which every computer science major should undergo as a rite of passage. The inner workings of an operating system are shrouded behind a veil of mystery and this project has attempted to lift that veil and discover its secrets. There is no magic making it all happen. As with any other software project: big problems are broken down into little problems, until one can go no further. This project is not complicated, it is simply sophisticated.

The project is far from finished and provides enormous scope for further exploration. The author has merely scratched the surface of where this project could be taken. Two areas which, due to time constraints, did not receive enough attention are the Widget Toolkit and Device Drivers. The list of implemented Widgets includes Buttons, Text Boxes, List Views and Scroll Bars. What has not been implemented is a seemingly unending list of user conveniences. The supported Devices include the Keyboard, Mouse, Video display adapter, Serial port, and PC Speaker. Notable omissions include secondary storage, the PCI bus and USB support. Given more time, then these are areas which the author would like to explore further.

The Spartan operating system may not have enough features to replace an established operating system for every-day use, that was not the intention. But it is exactly those missing features which sets the imagination on fire with endless possibilities. It is inhuman to fully comprehend the complete workings of the Linux, Mac OS and Windows operating systems, though it is entirely possible to fully understand the Spartan system. This encourages exploration and promotes a sense of adventure because it is within the reach of an individual, and that is something which has been lost in modern operating systems. There is tremendous potential to expand this project in any number of directions. That is an exercise which is left to the reader and hopefully in time a future student will take on that task. The author has thoroughly enjoyed every aspect of the research, design and implementation of this project, and there is a great sense of pride in developing Applications in an environment which you yourself have provided.

Appendix

Listing 3.1: Stacked Windowing System Core

```
int MoveWindow(WINDOW *Window, int NewX, int NewY, int NewW, int NewH) {
    if (!Window) return 0;
    if (topmost) {
        Blit(shadow.Canvas, topmost->PosX, topmost->PosY, topmost->Width,
            topmost->Height, frame.Canvas, topmost->PosX, topmost->PosY);
        if (NewW != topmost->Width || NewH != topmost->Height) {
            // TODO: Resize window, and issue a paint command
        }
        topmost->PosX   = min(max(NewX, 0), frame.Width-NewW);
        topmost->PosY   = min(max(NewY, 0), frame.Height-NewH);
        topmost->Width  = NewW;
        topmost->Height = NewH;
        Blit(Window->Canvas, 0, 0, Window->Width, Window->Height,
             frame.Canvas, Window->PosX, Window->PosY);
    }
    return 1;
}

int PushWindow(WINDOW *Window) {
    if (!Window) return 0;
    if (topmost) {
        Blit(topmost->Canvas, 0, 0, topmost->Width, topmost->Height,
             shadow.Canvas, topmost->PosX, topmost->PosY);
    }
    Window->Prev = 0;
    Window->Next = topmost;
    if (topmost) topmost->Prev = Window;
    topmost = Window;
    Blit(Window->Canvas, 0, 0, Window->Width, Window->Height,
         frame.Canvas, Window->PosX, Window->PosY);
    return 1;
}

int PopWindow(WINDOW *Window) {
    if (!Window || Window == bottom) return 0;
    WINDOW *Prev = Window->Prev;
    WINDOW *Next = Window->Next;
    if (Window->Prev) Prev->Next = Next;
    if (Window->Next) Next->Prev = Prev;
    if (Window == topmost) topmost = Next;
    Window->Prev = Window->Next = 0;
    for (WINDOW *Stack = bottom; Stack != topmost; Stack = Stack->Prev) {
        if (!Stack || Stack == topmost) break;
        Blit(Stack->Canvas, 0, 0, Stack->Width, Stack->Height,
            frame.Canvas, Stack->PosX, Stack->PosY);
    }
    Blit(frame.Canvas, 0, 0, frame.Width, frame.Height, shadow.Canvas, 0, 0);
    Blit(topmost->Canvas, 0, 0, topmost->Width, topmost->Height,
         frame.Canvas, topmost->PosX, topmost->PosY);
    return 0;
}
```

Listing 3.2: Optimized Blit Routine and IRender Virtual Machine

```
void Blit(BMP *Source, int SrcX, int SrcY, int Width, int Height,
          BMP *Target, int TrgX, int TrgY) {
    int x1 = min(max(TrgX, 0), Target->width);
    int y1 = min(max(TrgY, 0), Target->height);
    int x2 = min(max(TrgX+Width, 0), Target->width);
    int y2 = min(max(TrgY+Height, 0), Target->height);

    int x3 = min(max(SrcX, 0), Source->width);
    int y3 = min(max(SrcY, 0), Source->height);
    int x4 = min(max(SrcX+Width, 0), Source->width);
    int y4 = min(max(SrcY+Height, 0), Source->height);
    int span = min(x2-x1, x4-x3)*Target->bits/8;

    for (int y = y1; y < y2; y++) {
        int v = y - y1 + y3;
        void *s = &Source->canvas[v*Source->width + x3];
        void *t = &Target->canvas[y*Target->width + x1];
        memcpy(t, s, span);
    }
}

static bool IRender_Decoder(CCanvas *canvas) {
    static void (*opcode[])(CCanvas *canvas) = {
        IRender_Nop,    IRender_Push,   IRender_Pop,    IRender_Show,
        IRender_Hide,   IRender_Move,   IRender_Size,   IRender_Draw,
        IRender_Reveal, IRender_Refresh
    };
    if (!canvas) return false;
    while (canvas->Head != canvas->Tail) {
        u32 op = canvas->Ring[canvas->Head++];
        if (op >= elementsof(opcode)) {
            canvas->Head = canvas->Tail;
            return false;
        }
        opcode[op](canvas);
    }
    canvas->Head = canvas->Tail;
    return true;
}

static bool IRender_Dispatcher(CCanvas *canvas, u32 op, ...) {
    static u32 opcode[] = {
        RENDER_NUM_NOP,     RENDER_NUM_PUSH,    RENDER_NUM_POP,     RENDER_NUM_SHOW,
        RENDER_NUM_HIDE,    RENDER_NUM_MOVE,    RENDER_NUM_SIZE,    RENDER_NUM_DRAW,
        RENDER_NUM_REVEAL,  RENDER_NUM_REFRESH
    };
    if (op >= elementsof(opcode)) return false;
    int *x = &op;
    u8 tail = canvas->Tail;
    canvas->Ring[tail++] = *x++;
    for (u32 n = 0; n < opcode[op]; n++) {
        canvas->Ring[tail++] = *x++;
        if (tail == canvas->Head) return false;
    }
    canvas->Tail = tail;
    return true;
}
```

Bibliography

Research Papers

Ray Larner, 1987, “IBM FORTRAN Monitor System”, IBM Corporation
https://www.computer.org/csdl/proceedings/afips/1987/5094/00/50940815.pdf

Paul Fitts, 1954, “The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement”, Ohio State University.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.6753&rep=rep1&type=pdf

Fernando J. Corbató, 1962, “An Experimental Time-Sharing System”, Computation Center, MIT. http://larch-www.lcs.mit.edu:8001/~corbato/sjcc62/

Fernando Jose Corbató, 1963, Presentation of the Time-Sharing System,
https://www.youtube.com/watch?v=Q07PhW5sCEk
Ossanna, Mikus, and Dunten, 1965, "Communications and Input / Output Switching in a Multiplex Computing System", Joint Computer Conference, http://multicians.org/fjcc5.html

László Bélády, 1966, “A study of replacement algorithms for a virtual-storage computer”, https://www.researchgate.net/publication/224102735_A_study_of_replacement_algorithms_for_a_virtual-storage_computer

László Bélády, 1969, “An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine”, IBM, https://dl.acm.org/citation.cfm?doid=363011.363155

Alfred Aho, 1970, “Principles of Optimal Page Replacement”, Bell Telephone Laboratories.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.3111&rep=rep1&type=pdf

Dennis Ritchie, 1974, “The UNIX Time-Sharing System”,
https://people.eecs.berkeley.edu/~brewer/cs262/UNIX-annotated.pdf

Alan Kay, 1987, Presentation of the Xerox Alto system and the rationale that led to its development, https://www.youtube.com/watch?v=6ZdxiQoOBgs

Jochen Liedtke, 1995, "On u-Kernel Construction", ACM, ISBN: 0-89791-715-4, http://srl.cs.jhu.edu/courses/600.439/ukernel-construction.pdf

Hermann Härtig, 1997, "The Performance of u-Kernel-Based Systems", ACM, https://www.cs.cornell.edu/courses/cs614/2003sp/papers/HHL97.pdf

Michael Weisenmiller, 1999, "A Study of the Readability of On-Screen Text", Virginia Polytechnic Institute and State University. https://theses.lib.vt.edu/theses/available/etd-102999-110544/unrestricted/WeisenmillerDissertation.pdf

Books

Donald E. Knuth, 1968, "The Art of Computer Programming", Addison-Wesley, ISBN: 0201038013

John Lions, 1976, "Lions' Commentary on UNIX 6th Edition, with Source Code", Peer-to-Peer Communications, ISBN: 1573980137

Andrew Tanenbaum, 1987, "Operating System Design and Implementation", Pearson Education International, ISBN: 0131429388

Hans Peter Messmer, 1997, "The Indispensable PC Hardware Book", Addison-Wesley, ISBN: 0201403994.

William Stallings, 2005, "Operating Systems – Internals and Design Principles", Pearson Education International, ISBN: 0131278371

Peter Seibel and Ken Thompson, 2009, "Coders at work: Reflections on the Craft of Programming", Apress, ISBN: 1430219483, https://www.safaribooksonline.com/library/view/coders-at-work/9781430219484/

White Papers and Reference Materials

Dennis Ritchie, 1978, “UNIX Time-Sharing System: A Retrospective”, Bell System Technical Journal, https://archive.org/stream/bstj57-6-1947#page/n0/mode/2up

IBM, 1981, “IBM Personal Computer Technical Reference Manual”, http://www.nj7p.org/Computers/IBM%20PC/work/6025003.pdf

BYTE Magazine, 1984, Vol 09, Issue 09, “Guide to the IBM PCs”, https://archive.org/details/byte-magazine-1984-09

IBM, 1988, “IBM PS/2, Hardware Interface Technical Reference”, http://www.nj7p.org/Computers/IBM%20PC/work/PS2_HI.pdf

IBM, 1989, “IBM Personal System 2 Technical Reference Manual”, http://www.nj7p.org/Computers/IBM%20PC/work/PS2_Model_P70.pdf

IBM, 1990, “Keyboard and Auxiliary Device Controller, Technical Reference”, http://www.mcamafia.de/pdf/ibm_hitrc07.pdf

Intel, 2018, “Intel Architecture Reference Manuals”, https://software.intel.com/en-us/articles/intel-sdm

Intel, 2015, “A Tour beyond BIOS Memory Map Design in UEFI BIOS”, https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Map_in%20UEFI_BIOS.pdf

Android Authority, 2018, Mobile OS Market Shares https://www.androidauthority.com/android-posts-highest-ever-market-share-figures-711517/

VESA, 2018, Royalty Free Standards, https://www.vesa.org/vesa-standards