**UNIX OPERATING SYSTEM**

**AIAH NADINE Q. DELOS REYES**

**ITP51**

**OPERATING SYSTEMS**

**TABLE OF CONTENTS**

TITLE PAGE I

TABLE OF CONTENTS II

CHAPTER 1: INTRODUCTION AND HISTORY 1

CHAPTER 2: PROCESS MANAGEMENT 3

CHAPTER 3: CPU SCHEDULING 12

CHAPTER 4: MEMORY MANAGEMENT 15

CHAPTER 5: STORAGE MANAGEMENT 19

CHAPTER 6: I/O SYSTEMS 22

CHAPTER 7: FILE SYSTEMS 25

REFERENCES 34

**CHAPTER 1**

**INTRODUCTION AND HISTORY**

An operating system that is originally developed in the 1970s, known by many as UNIX, is a multiuser or multitasking operating system (OS) that is intentionally designed for flexibility and adaptability. UNIX is known as one of the operating systems that is developed using the C Programming Language. After it was developed, it was said to be an OS that had an extreme contribution on the industry of computer and electronics where it offers portability and stability across a range of diverse device types.

Multiplexed Information and Computing Service, or Multics, is an interactive time-sharing system that Bell Labs, which would later become AT&T, General Electric, and the Massachusetts Institute of Technology, attempted to develop in the late 1960s with the goal of allowing multiple users to access a mainframe simultaneously.

Unfortunately, they were disappointed with the results of the system they have made which resulted to Bell Labs pulling out the project where Bell computer scientists Ken Thompson and Dennis Ritchie continued their work, which leads up to the development of the Unix Operating System. Together with other Bell Lab Researchers, they build a suite of components that provided a a foundation for the UNIX OS. It included a hierarchical file system, a command-line interface, and multiple small utility programs.

Later on, Thompson made a self-hosting operating system with an assembler, editor, and shell. UNIX, pronounced as *YEW-nihks*, is an emasculated version of their first developed system, Multics. It is smaller than what the original developers has planned for Multics. UNIX is also a single-tasking system.

In the late 1972, UNIX was written in an assembler language, which is a class of low-level programming language designed to interact directly with hardware on a computer. This was revolutionary at the time because operating systems were thought to be too wide and sophisticated to be written in C Language. But the fourth edition is said to be written in the C Language where it increased the operating system’s portability across different computing platforms.

In the late 1970s and early 1980s, UNIX gathered a strong following in academia that led to commercial startups like Solaris Technologies and Sequent in order to adopt it in a larger scale. Moreover, between 1977 and 1995, Berkeley Software Distribution (BSD), one of the earliest UNIX distribution and the foundation for several other UNIX spinoffs, was developed by the Computer Systems Research Group at the University of California.

Meanwhile, in 1991, a student of University of Helsinki, known as Linus Torvalds, created a Unix-based operating system for his personal computer that is called as Linux. This moment led many Unix-like systems to grow and be popular.

In conclusion, today, many servers, workstations, systems, and devices are driven by Unix-based operating system. This includes macOS and Android mobile devices.

**CHAPTER 2**

**PROCESS MANAGEMENT**

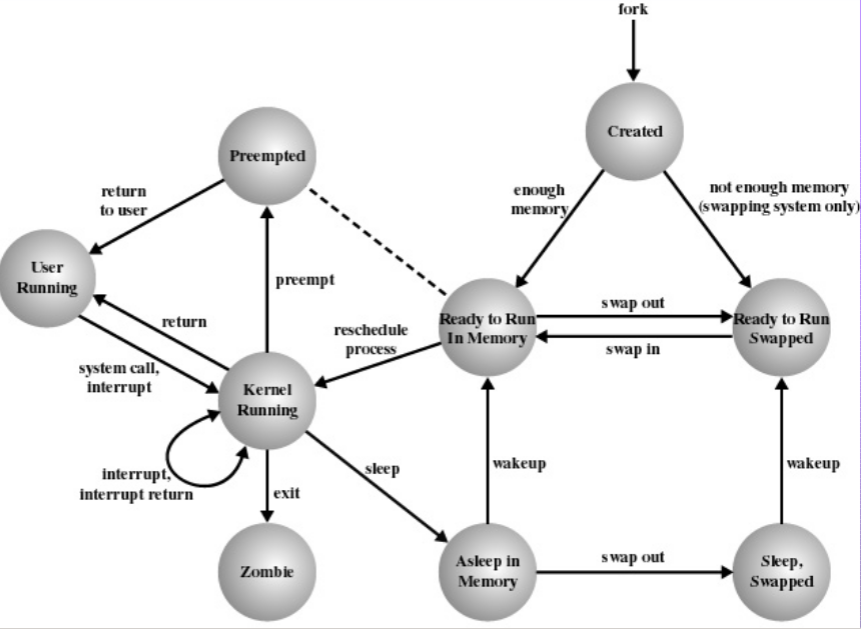
This chapter will discuss the process management of the Unix Operating System. This will cover the modes of execution, the processes states of the system, and how a program is executed in Unix OS.

In Unix process management, the operating system functions executes within user process. There are two (2) modes of execution for Unix System: *user mode* and *kernel mode.* Furthermore, there are also two (2) types of processes under Unix OS: the *system processes,* this executes the operating system code and the *user processes,* where this one executes the user program code. In addition, system call is used to transfer from user mode to system mode.

Moreover, there are nine (9) process states in Unix OS. These processes states are the following:

**Table 1. Unix Processes States**

|  |  |
| --- | --- |
| PROCESS STATES | DESCRIPTION |
| User Running | Executing in user mode. |
| Kernel Running | Executing in kernel mode. |
| **Ready to Run, in Memory** | Ready to run as soon as the kernel schedules it. |
| **Asleep in Memory** | Unable to execute until an event occurs; process is in main memory (a blocked state). |
| **Ready to Run, Swapped** | Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute. |
| **Sleeping, Swapped** | The process is awaiting an event and has been swapped to secondary storage (a blocked state). |
| **Preempted** | Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process. |
| **Created** | Process is newly created and not yet ready to run. |
| **Zombie** | Process no longer exists, but it leaves a record for its parent process to collect. |



**Figure 1. Unix Process State Diagram**

If the process is about to move from kernel mode to user mode, preemption can occur. If the process is running in kernel mode it may not be preempted which makes Unix not suitable for real time processing. In Unix, *Process 0* is a special process that is created when the system boots. Then *Process 1,* also called as *init process,* is the child of *Process 0.* While all other processes has *Process 1* as ancestor and all new processes are created under *Process 1.*

In Unix, elements of process image is divided into three (3): User level context, Register context, and System level context. In *user level context,* it covers the process text, process data, user stack, and the shared memory. On the other hand, *register context* has the program counter, process states registers, stack pointer, and general purpose registers. While the *system level context* has the process table entry, user area, per process region table, and the kernel stack. Take note that when a process is not running, the processor status information is stored in register context area.

Furthermore, the system context model has two (2) parts: the *static* and the *dynamic.*

Process table entry contains the process control information that is always accessible to the kernel. So in VM systems all processs table entries are maintained in the main memory. Under the Unix Process Table Entry are the process status, pointers, process size, user identifier, event descriptor, priority, signal, timers, p\_link, and the memory status.

**Table 2. Unix Process Table Entry**

|  |  |
| --- | --- |
| **Process Table Entries** | **Description** |
| **Process Status** |  |
| **Pointers** | To user area and process memory area (text, data, stack). |
| **Process size** | Enables OS to know how much space to allocate. |
| **User identifier** | * Real user ID – ID of user who is responsible for the process. * Effective user ID – used by process to gain temporary privilege while the program is being executed as a part of process. * Process identifiers – ID of the process. |
| **Event descriptor** | Valid when a process is in sleep state. When the event occurs, the process is transferred to a ready to run state. |
| **Priority** | Used for scheduling. |
| **Signal** | Enumerates signals send to process but not yet handled. |
| **Timers** | Process execution time, kernel resource utilization, and user set timer used to send alarm signal to a process. |
| **P\_link** | Pointer to the next link in the ready queue. |
| **Memory status** | Indicates the process image is in the main memory or swapped out. |

On the other hand, user are have additional process control information that is needed by the kernel when it executes in context process. This is also used for swapping process to and from the memory. The Unix User Area has the process table pointer, user identifiers, timers, signal handling array, control terminal, error field, return value, I/O parameters, file parameters, user file descriptor table, limit fields, and permission mode field.

**Table 3. Unix User Area**

|  |  |
| --- | --- |
| User Area | Description |
| **Process table pointer** | Indicates entry corresponding to the user area. |
| **User identifiers** | Real and effective user IDs, used to determine user privileges. |
| **Timers** | Record time that the process spent executing in user mode and kernel mode. |
| **Signal handling array** | For each type of signal defined in the system, indicate how the process will react to receipt of that signal. |
| **Control terminal** | Indicate login terminal for this process, if it exists. |
| **Error field** | Record errors encountered during system call. |
| **Return value** | Contain result of s/m call. |
| **I/O Parameters** | Describes amount of data transfer, the address of the source data array in user space, file offset for I/O. |
| **File parameters** | Current directory and current root describe the file system environment of the process. |
| **User file descriptor table** | Record the file the process has opened. |
| **Limit fields** | Restrict the size of the process and size of the file it can write. |
| **Permission mode field** | Mask mode settings on files the process creates. |

Also, *kernel stack* is used when a process is executing in kernel mode and contains information that must be saved and restored as procedure calls and interrupts.

Moreover, the *fork()* method in Unix gives the new process a slot in the process table and it gives the new process a distinct process ID. It also creates a duplicate of the parent’s process image, excluding shared memory. Also to indicate that a new process now also owns these files, it raises counters for any files controlled by the parent. The kid process is put in a ready-to-run state. It gives the kid process a 0 value and the parent process the child’s ID number. All of these tasks are completed in the parent process’ kernel. After completing those functions, OS will do the following operations as a part of dispatcher routine:

1. Stay in the parent process. At the parent’s fork call point, control returns to user mode.
2. Transfer control to the child process. The return from the fork call is where the child process starts running, at the same place in the code as the parent.
3. Transfer control to another process. The parent and child are left in a state of readiness to flee.

Furthermore, there are three (3) types of processes in Unix. The three mainly types are the *user process, daemon process,* and *kernel process.* The *daemon process* completes the task on a system-wide basis. Although the function might be of any auxiliary type, it is essential to managing the system’s computing environment. Examples are print spooling and network management. A daemon process that has been created can continue to exists for the duration of the operating system.

Additionally, there is also a termination process in Unix which is called as the Exit(status\_code). Where status\_code denotes the state of termination for the procedure. After getting the exit call, the kernel performs the following actions:

1. Close all the process’ open files.
2. Releases the allocated memory for it.
3. Destroy the process’ user area:

* The proc structure is not destroyed; rather, it is held in place until Pi’s parent destroys it.
* Although the process has ended, it is still active. This is why the procedure is termed *ZOMBIE.*

Also, the exit call signals Pi’s parent, who may choose to ignore it.

In addition, there is also a wait statement in Unix. Where Pi is able to wait until a child is terminated. The variable within the address space of Pi is ‘*Wait(address(xyz)); // xyz’.* The wait call saves the termination status of the termination status of the terminated child process into *xyz* and returns with the ID of the terminated child process instantly if process Pi has child processes and at least one of them has already terminated. Pi would only be able to see the termination status of any additional child processes if it were to repeat the wait call. If process Pi has children and none of them have ended, it is blocked; it will unblock when one of the child processes does. If Pi is childless, the wait call returns *a – 1.*

Interrupt handling in Unix will only ever carry out one interrupt at a time in order to prevent race conditions. A level of interrupt priority is assigned to each interrupt. The CPU also has an interrupt priority level assigned to it. An interrupt at priority level 1 is addressed only if it is greater than the CPU’s interrupt priority; otherwise, it is left unattended until the CPU’s interrupt priority level drops.

Moreover, system calls in Unix take parameters that are necessary for them to work. These arguments are present on the user stack of process that initiates the system call when a call takes place. It is anticipated that the call number is in register O. The system call handler uses this number to identify the specific system function that is being called upon. It is aware of the handler’s address for that function based on an internal table. The process making the call has system call parameters on its user stack. These parameters are copied from the user stack into a few common locations in the user area before control is transferred to the handler for a particular call. The *‘kill(Spid>, <signum>)* system call can be used to convey a to a signal process or group of processes. Where the process signal’s ID to be conveyed is represented by pid (Recipient must be in the same process tree as sender and sender must know recipient address). Pid value 0 -> indicates that all processes in the same group as the sender process should receive this signal. To access processes that are not inside the sender’s process tree, use the pid value – 1.

In addition, for signal handling in Unix, where *Oldfunction = signal(<signum>,<function>)* where signal is a C library function that calls the signal system. Function is the name of the function, and signum is an integer. The function ought to be carried out whenever signal <signum> occurs. The user can substitute function with either 0 or 1. Where 0 denotes that the kernel’s default action should be carried out and 1 denotes that a signal should be ignored when it occurs. The bit in the destination process’ proc structure that corresponds to a signal is always set to 1 when a signal is sent to it. At this point, the kernel ascertains whether the destination process is ignoring the signal. If not, a plan to send the signal to the process will be made. The signal is provided when the process is prepared to accept it; if it is ignored, it stays pending. If the desired process is in block status, a UNIX signal stays pending. When the process exits the blocked state, the signal will be sent. When signals are sent out, before a process is blocked, it returns from a system call after being unblocked.

Also, there are interesting signals in Unix. Unix OS use signals to communicate with processes. Some interesting signals include:

**Table 4. Interesting Signals in Unix**

|  |  |
| --- | --- |
| **Interesting Signals** | **Description** |
| SIGCHLD | Child process died or suspended |
| SIGFPE | Arithmetic fault |
| SIGILL | Illegal instruction |
| SIGINT | Tty interrupt (Control C) |
| SIGKILL | Kill process |
| SIGSEGV | Segmentation fault |
| SIGSYS | Invalid system call |
| SIGXCPU | Exceeds CPU limit |
| SIGXFSZ | Exceeds file size limit |

Understanding signals is crucial for managing processes and handling system events in Unix-like operating systems.

**CHAPTER 3**

**CPU SCHEDULING**

CPU Scheduling in Unix addresses the issue of selecting which of the procedures in the CPU has to be assigned to the ready queue. Multi-programmed operating systems work on the basis of CPU scheduling, which alternates between activities. The goal of multiprogramming is to maximize CPU utilization by executing some tasks continuously.

The basic idea of multiprogramming is where memory is used to store multiple processes at once. The operating system uses the CPU when a process needs to wait and shifts the CPU's focus from that task to another.

Decisions on CPU scheduling may be made under the subsequent four situations:

* When a process transitions from the state of operation to the state of waiting.
* When an executable moves from the running to the ready states condition.
* When a process enters the ready state after leaving the waiting state.
* At the conclusion of a process.

Interactive processes are intended to profit from CPU scheduling in Unix. A priority algorithm, which reduces to round-robin scheduling for CPU-bound workloads, assigns processes short CPU time slices. The UNIX system's scheduler is a member of the round-robin with multi-level feedback general class of operating system schedulers. This means that the kernel allots CPU time to a process for a brief window of time, preempts a process that goes over that window, and feeds the excess time back into one of several priority queues. Before a process is complete, it could go through the "feedback loop" several times.

Scheduling Algorithms are as follows:

* Non Preemptive vs. Preemptive Scheduling
* Shortest Job First (SJF)
* First Come First Serve (FCFS)
* Priority Scheduling
* Round Robin Scheduling (RR)
* Multi-level Feedback Queue Scheduling

Furthermore, Non Preemptive algorithms are made to ensure that a process remains in the processor after it has reached the end of its service time. While in Preemptive Scheduling, the process occupying the processor at the moment should always be the one with the highest priority. The process that is presently using the processor should be removed and put back on the ready list until it is once more the highest priority process in the system if a new process with a higher priority joins the ready list.

While in Priority Scheduling, SJF is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order. Moreover, we discuss scheduling in terms of high priority and low priority. Priorities are generally some fixed range of numbers, such as 0 to 7, or 0 to 4095.

Meanwhile in Round Robin Scheduling, the active jobs are stopped to allow the pending jobs to use the processor. A *time quantum*, sometimes referred to as a time slice, is defined. A time quantum typically ranges from ten to one hundred milliseconds. The CPU Scheduler rounds the ready queue, assigning the CPU for a maximum of one time quantum to each process. We maintain the ready queue as a First In, First Out (FIFO) queue of processes in order to implement RR scheduling. The ready queue's tail is expanded to accommodate new processes. Then the CPU scheduler selects the first process in the ready queue, launches the process, and sets an interrupt timer for one time quantum later.

Typically, in multi-level queue scheduling, when a process joins, it is typically permanently assigned to a queue upon system entry. Procedures don't switch between the lines. However, processes are able to switch between the queues thanks to multi-level feedback queue scheduling. The intention is to divide up processes with various CPU burst properties. A process will be relegated to a lower priority queue if it consumes excessive CPU time.

**CHAPTER 4**

**MEMORY MANAGEMENT**

In this chapter, we will delve into the intricate realm of Memory Management within the Unix operating system. Memory Management is a critical aspect of Unix, playing a pivotal role in optimizing system performance and ensuring efficient resource utilization. As we navigate through this discussion, we will explore the mechanisms employed by Unix to allocate, deallocate, and organize memory space for processes. Understanding the nuances of memory allocation, virtual memory, and the strategies employed by the Unix OS to facilitate seamless multitasking is essential for system administrators, developers, and anyone seeking a comprehensive grasp of Unix internals.

Memory management is a fundamental aspect of any operating system, playing a crucial role in the efficient utilization of a computer's resources. The primary function of memory management is to oversee the allocation and deallocation of system memory, ensuring that processes have access to the necessary space to execute and store data. In an operating system, memory is typically categorized into different regions, including the kernel space reserved for the core functions of the operating system and user space where application processes operate. Operating systems, such as Unix, use various techniques like virtual memory to extend the available physical memory and support multitasking.

In modern operating systems, memory management also includes features like paging, swapping, and caching to optimize the use of both physical and virtual memory. Overall, a well-designed memory management system is essential for the smooth functioning of an operating system, supporting the diverse requirements of applications and ensuring a seamless user experience.

Behind an operating system, there are two types of memory: *Random Access Memory (RAM)* and *Read-Only Memory (ROM).* RAM, or random access memory, is a key part of the hardware design of computers and is essential to the system's overall performance. RAM is the short-term, high-speed storage that the CPU of the computer utilizes to store and quickly retrieve data that is being processed or needed by programs that are now operating. In contrast to long-term storage systems like SSDs or hard drives, RAM is volatile, meaning that when the power is switched off, its data are lost. In RAM, "random access" refers to the ability to access data instantly and directly at any random point, as opposed to sequential access memory, which necessitates reading data in a particular order. The CPU can read and write data quickly thanks to this random access function, which speeds up program execution and effective multitasking.

Meanwhile, in computers, read-only memory, or ROM, is a kind of non-volatile memory that keeps its contents even after the power is switched off. Random Access Memory (ROM) holds data that is permanently or semi-permanently stored, in contrast to Random Access Memory (RAM), which is volatile and loses its contents upon power outage. The phrase "read-only" denotes that regular computer operations typically cannot readily alter the data kept in ROM. ROM is essential for a computer system's basic functions, particularly while bootstrapping. Usually, it holds firmware, which is a collection of low-level instructions required to initialize hardware and start an operating system.

Furthermore, operating systems depend heavily on memory management to provide effective memory resource allocation and use. To successfully manage memory, a variety of strategies are used to handle issues such physical memory limitations, multitasking demands, and process isolation and protection. These are a few typical operating system memory management strategies:

* **Contiguous Memory Allocation** - this method allots a continuous block of RAM to every process. Easy to use and effective, however it may result in internal (unused memory within allocated blocks) and external (unallocated spaces between allocated blocks) fragmentation problems.
* **Paging** - Processes are separated into corresponding blocks called page frames, while memory is partitioned into fixed-size blocks called pages. Makes it possible to use memory more effectively, lessen fragmentation, and make it simpler to load and unload programs. Can, however, result in page-level fragmentation.
* **Segmentation** – this separates a program into logically addressable sections, such as the code, data, and stack. Also provides a more adaptable method of memory management than contiguous allocation. But may experience fragmentation from the outside.
* **Virtual Memory -** thisincreases the amount of physical memory that is accessible by combining RAM and disk space. This permits the running of programs that would not fit completely in physical memory. Data is moved between RAM and disk using page replacement algorithms like Least Recently Used (LRU) and First-In-First-Out (FIFO).

Since the key component of the Unix operating system is memory management, which entails allocating, allocating, and releasing memory resources to guarantee effective use and peak performance. To meet the demands of its multitasking environment and to give processes a stable and responsive foundation, Unix uses a variety of memory management strategies. The following are important facets of Unix memory management:

* **Virtual Memory** - Operating systems that resemble Unix use virtual memory, which enables processes to access huge amounts of memory even when there isn't enough physical memory available. RAM and disk storage are used in conjunction to create virtual memory.
* **Paging** - The process of transferring virtual memory addresses to physical pages and partitioning physical memory into fixed-sized blocks known as pages is known as paging. This frees up physical memory for other processes by enabling the operating system to relocate memory pages to disk when they are not in use.
* **Swapping** - Transferring a whole process's address space from physical memory to the disk is known as swapping. When paging is unable to free up physical memory for another process, the operating system resorts to this.
* **Memory Allocation** - Operating systems resembling Unix use dynamic memory allocation to give RAM to processes as needed. The buddy system, which employs a free list of blocks of various sizes and allocates memory in the smallest block that may satisfy the request, is the most often used allocation technique.
* **Caching** - Caching is a technique that keeps frequently accessed data in physical memory in order to speed up memory access. Operating systems that resemble Unix employ a range of caching strategies, including disk and file system caching, to boost efficiency.

To sum up, Unix OS Memory Management is an essential part that helps to guarantee the effective distribution and use of memory resources. Virtual memory, paging, and other strategies let Unix efficiently manage heavy workloads, many tasks, and multitasking. The system's stability and security are enhanced by the use of shared memory, dynamic heap management, and memory protection. Overall, Unix is a strong option for a variety of workloads and applications since its memory management techniques are made to offer a dependable and responsive computing environment.

**CHAPTER 5**

**STORAGE MANAGEMENT**

In this chapter, we will delve into the intricacies of how Unix effectively organizes, allocates, and manages storage resources. The methodical and effective management of digital data storage resources in a computer environment is known as storage management. It entails a number of procedures, guidelines, and technological advancements meant to maximize the distribution, arrangement, and application of storage capacity in order to satisfy a system's or organization's data storage requirements. Ensuring data integrity, availability, dependability, and effective use of storage resources are the main objectives of storage management.

A crucial component of Unix system administration is storage management, which includes a variety of methods and instruments to maximize disk usage, improve efficiency, and guarantee data integrity. We will go through the basic ideas and cutting-edge tactics that Unix uses to manage storage effectively, from file systems and disk partitioning to volume management and RAID settings. This chapter aims to give you important insights into the ever-changing world of storage management under the Unix operating system, whether you are an experienced system administrator looking to hone your craft or a novice keen to learn the fundamentals of Unix storage. Accompany us on this investigation as we dissect the intricacies and optimal methodologies that regulate storage within the Unix framework.

In Unix operating systems, storage management refers to a collection of procedures and systems for effectively allocating, managing, and organizing storage resources. Important elements and ideas pertaining to Unix storage management comprise:

* **File Systems -** File systems are used by Unix to arrange and store data on storage devices. File systems that are often used are ext4, XFS, and ZFS. The file system gives users and apps access to and organization of data through a hierarchical structure of directories and files.
* **Disk Partitioning -** A physical disk can be partitioned into different portions. A file system can be used to format each partition and use it as a data storage device. Partitioning in Unix often involves the use of fdisk and parted.
* **Mounting and Unmounting -** To attach a file system to a particular directory in the file hierarchy, Unix systems utilize the mount command. Detaching a file system when it is no longer required is known as unmounting.
* **RAID Configurations -** Redundant Array of Independent drives (RAID) configurations combine several physical drives into a single logical unit, improving performance, availability, and data security. Multiple RAID levels, including RAID 0, RAID 1, and RAID 5, are supported by Unix.
* **File System Maintenance -** Using programs like fsck, routine file system maintenance chores involve looking for faults and irregularities. For best results, file system defragmentation and resizing can also be required.

In Unix, efficient storage management is essential to preserving data integrity, system performance, and dependability. System administrators are responsible for meticulously organizing, supervising, and modifying storage configurations to satisfy the changing demands of users and applications while maintaining data security and accessibility.

To conclude, the storage management of Unix operating systems consists of a comprehensive collection of procedures and instruments intended to effectively manage the arrangement, distribution, and upkeep of storage assets. Unix offers an extensive feature set for efficient storage management, ranging from file systems and disk partitioning to logical volume management, RAID settings, and backup plans. System administrators can customize storage configurations to meet the unique requirements of users and applications thanks to Unix's flexibility, which guarantees optimum performance, data integrity, and dependability.

Moreover, precise control over storage-related tasks is made possible by Unix's command-line tools and utilities, which make monitoring, performance tuning, and routine maintenance easier. Unix is a stable and flexible platform for storage management, which is further reinforced by the addition of virtual memory, swap space management, and effective backup and restoration systems.

Unix's storage management features are still essential for preserving a responsive and steady computing environment even as data volumes increase. Whether web applications, databases, or file servers are being used, Unix offers a solid framework for handling a range of storage needs. In summary, Unix's storage management features make a substantial contribution to the general efficacy and efficiency of Unix-based systems.

**CHAPTER 6**

**I/O SYSTEMS**

For this chapter, we will discuss about the management of communication between the computer's central processing unit (CPU) and external devices, including storage devices, input devices (like keyboards and mice), output devices (like monitors and printers), and communication devices (including network interfaces). All of the mentioned above is the responsibility of the input/output (I/O) system, also referred to as an IO system.

Facilitating data flow between the computer and external devices is the main goal of an I/O system. It offers a standardized interface via which application programs can conduct input and output activities, abstracting and streamlining the intricate complexities of communicating with different hardware devices. This abstraction makes software more portable across multiple computer platforms and protects application developers from the nuances of particular hardware implementations. Key components of an I/O system includes the following:

* **Device Drivers -** These are software modules that act as bridges between particular hardware devices and the operating system. Device drivers offer a common interface over which various hardware components can be communicated with by the operating system and applications.
* **I/O Controllers -** These physical elements are in charge of overseeing the real communication that takes place between the central processing unit and other devices. I/O controllers frequently include intelligence built in to manage activities and protocols unique to a given device.
* **Buffers and Caching -** During input/output (I/O) activities, buffers are utilized as temporary storage spaces to hold data, enabling effective data flow between devices and memory. To reduce the need for repetitive I/O operations, data that is often accessed can be stored using caching methods.

The whole functionality and performance of computer systems depend on efficient I/O systems. They are essential in controlling the data transfer between the computer and its add-ons, guaranteeing correct and timely communication. An I/O system's design affects things like throughput, user experience, and system responsiveness.

A key element that enables communication between the computer's hardware and software and ensures effective data transmission across devices is the Unix operating system's input/output (I/O) system. Renowned for its design ethos of simplicity and modularity, Unix boasts a resilient I/O system distinguished by its adaptability, consistency, and dedication to the Unix tenets.

The idea of treating devices as files is at the core of the Unix I/O system. All hardware, including hard disks, printers, and network connections, is abstracted into files under Unix. Programmers may utilize a standard set of system calls and commands for I/O activities, whether they are reading from a keyboard, writing to a file, or interacting over a network, thanks to this uniformity, which makes it easier for apps and devices to connect.

The use of file descriptors is one of the main components of the Unix I/O system. Small, non-negative integers called file descriptors are used to describe open files or I/O streams. File descriptors represent standard input/output (I/O) channels like stdin, stdout, and stderr, giving programs a consistent way to communicate with input and output. Application functionality can be neatly and modularly separated from particular I/O devices thanks to this abstraction.

I/O operations, both blocking and non-blocking, are supported by the Unix I/O system. Processes have the ability to send read or write requests and decide whether to block an operation until it finishes or to carry on with other activities in the non-blocking mode. This adaptability is essential for developing responsive and effective programs, particularly in contexts where processes may have different input/output requirements due to multitasking.

Additionally, Unix has a strong and adaptable feature called I/O redirection. Users can reroute input or output streams from or to files using shell commands and operators, allowing data to move across processes seamlessly. With the help of this capability, users can more easily design complicated data processing pipelines by improving the composability of Unix commands.

Furthermore, asynchronous I/O operations are supported by Unix via the use of system calls and signals. Processes can start I/O operations and carry out other tasks without having to wait for the I/O request to finish thanks to asynchronous I/O. This is especially useful for improving the responsiveness of programs that use several I/O operations running at once.

Device drivers are supported by a uniform interface thanks to the use of device files in Unix. This also makes adding new devices easier. Applications can access physical devices in a standardized manner thanks to device files, which are stored in the /dev directory. This method perfectly reflects the design concept of Unix, which offers a clear and uncomplicated interface for a wide range of operations.

To conclude this chapter, the Unix operating system's I/O system is a fundamental component of its design philosophy. Unix offers a versatile and consistent framework for managing input and output by treating devices as files, using file descriptors, providing I/O redirection, and enabling both synchronous and asynchronous I/O operations. Unix's I/O system is elegant and efficient because of its simplicity, as well as its support for device files and composition capabilities. The success of Unix as a trailblazing and durable computing platform is still attested to by the timeless principles of its I/O design, which continue to have an impact on contemporary operating systems.

**CHAPTER 7**

**FILE SYSTEMS**

In this chapter, we will embark on a journey into the heart of data organization and storage—the file system. An essential part of operating systems is the file system, which offers an organized, hierarchical method for managing, storing, and retrieving data. The file system is crucial in determining how our digital interactions are shaped, from the straightforward process of producing a document to the intricate management and protection of enormous volumes of data.

We will examine the ideas behind file systems as we move deeper into this chapter, learning about their construction, features, and guiding principles. We'll look at how file systems help to make computing environments more efficient and well-organized, guard against data loss, and make data access easier.

One of the main components of Unix operating systems, the file system adds to its well-known efficiency, adaptability, and reliability. The structure and organization of Unix file systems enable a smooth and hierarchical method of managing, storing, and retrieving data. This essay examines the essential traits, guiding ideas, and features that make up the Unix file system and add to its long-term viability.

Operating systems based on Unix employ the hierarchical Unix file system to store and arrange files and directories. The structure resembles a tree and begins with a single directory known as the root directory, indicated by the forward slash (/) character.

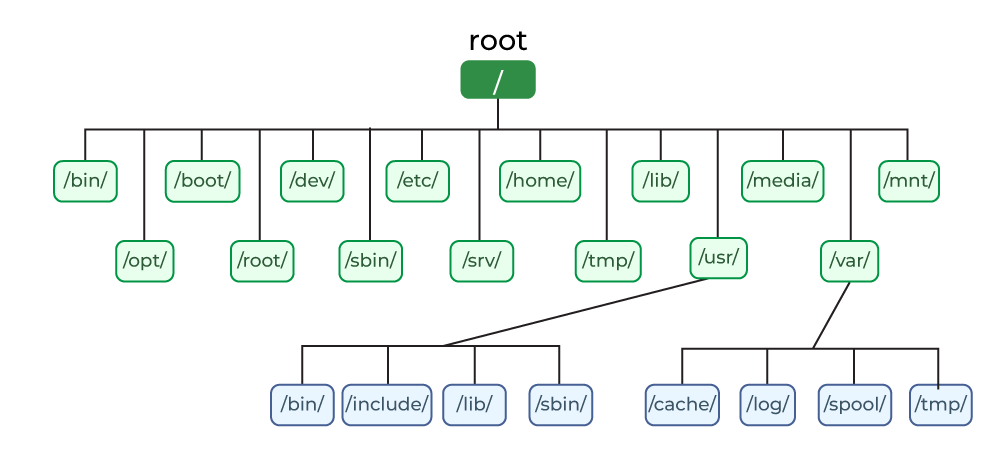
The directory hierarchy used by the Unix file system makes file organization and navigation simple. Each file and directory has a unique name, and directories can include both files and other directories.

A set of permissions is another tool used by the Unix file system to manage access to files and directories. Permissions may be adjusted to provide or deny access to the owner and group that are connected to each file and directory.

The usage of inodes (index nodes) to represent files is one of the distinguishing characteristics of Unix file systems. An inode, which holds metadata including ownership, timestamps, permissions, and pointers to the actual data blocks on the disk, is linked to each file and directory. The Unix file system is more efficient and modular because of this division of metadata and data.

Everything should be treated as a file, according to the philosophy of Unix file systems. This conceptual consistency makes it easier for applications to communicate with a variety of hardware components by extending to devices, directories, and even inter-process communication. Users may browse the Unix file system with well-known commands like cd, ls, and cp because files are arranged into a hierarchical structure of directories.

Both hard and symbolic links are supported by Unix file systems, giving users the ability to create alternate file references. Hard links save disk space and streamline file administration by enabling several directory entries to refer to the same inode, whereas symbolic links serve as pointers to other files, enabling flexible file referencing.



**Figure 2. Unix File System**

**Table 5. Directories or Files and their Description**

|  |  |
| --- | --- |
| **NAME** | **DESCRIPTION** |
| / | The slash / character alone denotes the root of the file system tree. |
| /bin | Stands for “binaries” and contains certain fundamental utilities, such as ls or cp, which are generally needed by all users. |
| /boot | Contains all the files that are required for successful booting process. |
| /dev | Stands for “devices”. Contains file representations of peripheral devices and pseudo-devices. |
| /etc | Contains system-wide configuration files and system databases. Originally also contained “dangerous maintenance utilities” such as init,but these have typically been moved to /sbin or elsewhere. |
| /home | Contains the home directories for the users. |
| /lib | Contains system libraries, and some critical files such as kernel modules or device drivers. |
| /media | Default mount point for removable devices, such as USB sticks, media players, etc. |
| /mnt | Stands for “mount”. Contains filesystem mount points. These are used, for example, if the system uses multiple hard disks or hard disk partitions. It is also often used for remote (network) filesystems, CD-ROM/DVD drives, and so on. |
| /proc | procfs virtual filesystem showing information about processes as files. |
| /root | The home directory for the superuser “root” – that is, the system administrator. This account’s home directory is usually on the initial filesystem, and hence not in /home (which may be a mount point for another filesystem) in case specific maintenance needs to be performed, during which other filesystems are not available. Such a case could occur, for example, if a hard disk drive suffers physical failures and cannot be properly mounted. |
| /tmp | A place for temporary files. Many systems clear this directory upon startup; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a startup script at boot time. |
| /usr | Originally the directory holding user home directories,its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of /usr, such as the default as in Minix. (on modern systems, these user accounts are often related to server or system use, and not directly used by a person). |
| /usr/bin | This directory stores all binary programs distributed with the operating system not residing in /bin, /sbin or (rarely) /etc. |
| /usr/include | Stores the development headers used throughout the system. Header files are mostly used by the #include directive in C/C++ programming language. |
| /usr/lib | Stores the required libraries and data files for programs stored within /usr or elsewhere. |
| /var | A short for “variable.” A place for files that may change often – especially in size, for example e-mail sent to users on the system, or process-ID lock files. |
| /var/log | Contains system log files. |
| /var/mail | The place where all the incoming mails are stored. Users (other than root) can access their own mail only. Often, this directory is a symbolic link to /var/spool/mail. |
| /var/spool | Spool directory. Contains print jobs, mail spools and other queued tasks. |
| /var/tmp | A place for temporary files which should be preserved between system reboots. |

In addition, the Unix files system contains several different types of files. The classification of Unix file system are the: *Ordinary Files, Directories, Special Files, Pipes, Sockets,* and the *Symbolic Links.*

Any file on the system that has text, data, or computer instructions are regarded as an ordinary file. It is used to keep track of your data, including written words and drawn pictures. This kind of file is what you typically work with.

Found always in or beneath a directory file. This do not include any additional files. The "-" symbol in the long-format output of ls -l designates this type of file.

Both special and regular files are kept in directories. UNIX directories are similar to folders for users who are accustomed to using Windows or Mac OS. Every file and subdirectory housed in a directory file has an entry in it. There will be 10 entries in a directory if it contains 10 files. Every entry consists of two parts. (1) The filename; (2) An inode number, which is a special identifying identifier for the file or directory. In long-format output of ls –l , this type of file is specified by the “d” symbol.

Meanwhile, special files is used to simulate an actual physical device for input/output (I/O) tasks, such as a printer, tape drive, or terminal. On UNIX and Linux systems, devices or special files are used for device input/output (I/O). They show up in a file system in the same manner as regular files or directories. Character special files and block special files are the two types of special files available on UNIX systems for every device:

* One character at a time is conveyed while using a character special file for device input/output (I/O). We refer to this kind of access as raw device access.
* For device input/output (I/O), data is sent in big fixed-size blocks when a block special file is used. Block device access is the term for this kind of access.

It's one character at a time for terminal devices. However, when it comes to disk devices, raw access refers to reading or writing entire blocks—data segments that are built into the disk itself.

* The letter "c" designates character special files in the ls -l long-format output.
* The "b" symbol designates block special files in the ls -l long-format output.

Moreover, With UNIX, you may use pipes to connect commands together. A Unix pipe allows data to travel one way; the output or result of one command sequence is used as the input for the next command sequence. The pipe functions as a temporary file that is only meant to keep data from one command until it is read by another. In the command line, create a pipe by placing a vertical bar (|) between two commands, such as this: who | wc -l. Named pipes are indicated by the symbol "p" in the long-format output of ls -l.

A unique file that enables sophisticated inter-process communication is called a Unix socket, sometimes known as an inter-process communication socket. In a client-server application framework, a Unix socket is utilized. To put it simply, it's a data stream that resembles network streams and sockets, but with all transactions occurring locally within the filesystem. The “s” sign indicates a Unix socket in the long-format output of ls -l.

Symbolic links, often called soft links, are used to reference other files on the file system. The path to the file it refers to is contained in text form. A symbolic link will appear to have its own name to the end user, but it will refer to the file it points to for these activities when you try to read or write data to it. The data file would remain present even if the soft link was deleted. The symbolic file would not work correctly if the source file was removed or moved. Symbolic links are shown by the "l" symbol (a lowercase L) in the long-format output of ls -l.

To add up, file system of Unix OS still have its own advantages and disadvantages. The advantages and disadvantages of the Unix File System are the following:

**Table 6. Advantages and Disadvantages of Unix File System**

|  |  |
| --- | --- |
| **ADVANTAGES** | **DISADVANTAGES** |
| **Hierarchical organization** - the Unix file system's hierarchical layout facilitates simple file and directory organization and navigation. | **Complexity -** understanding and managing the Unix file system can be challenging, particularly for users who are unfamiliar with the command line interface. |
| **Robustness -** the stability and dependability of the Unix file system are well recognized. It can manage big data loads without crashing or becoming unstable. | **Steep Learning Curve -** the Unix file system may be challenging for users to learn if they are unfamiliar with Unix-based computers. |
| **Security -** through the use of permissions, administrators can manage who has access to specific files and directories on the Unix file system. | **Lack of User-friendly Interface -** the command line interface, which is mostly used to administer the Unix file system, might not be as intuitive as a graphical user interface. |
| **Compatibility -** file transfers between various Unix-based computers are simple because of the widespread use and support of the Unix file system. | **Limited Support for Certain File Systems -** Although many file systems are compatible with the Unix file system, certain file systems are not supported completely. |

In summary, the architectural ideas that have created Unix a robust and long-lasting operating system are demonstrated by the Unix file system. A sophisticated permission scheme, inode-based organization, hierarchical structure, and a variety of file types all combine to make the file system both user-friendly and powerful. The Unix file system is a mainstay of Unix-based computer environments, impacting the design of file systems in succeeding operating systems with its versatility in supporting a wide range of storage devices, security measures, and advanced functions. The Unix file system continues to be a fundamental component of the complex web of Unix operating systems, whether it is used for controlling permissions, browsing directories, or establishing symbolic connections.

In conclusion, the Unix operating system is a titan of the computer industry, recognized for its timeless ideas, sturdy architecture, and significant influence on the development of operating systems. Unix, which was developed at Bell Laboratories in the late 1960s, has withstood the test of time, changing and impacting the very structure of computing environments.

A core set of ideas, including as modularity, simplicity, and the idea that everything is a file, are the foundation of Unix's success. These ideas not only influenced Unix's design but also influenced other operating systems, setting the stage for many of the modern conveniences we take for granted.

Because of the modular approach encouraged by Unix's architecture, strong tools and utilities that can be used to complete challenging tasks have been developed. With its clear and expressive syntax, the CLI allows both administrators and users to interact with the system effectively. The pipeline concept allows commands to be composed in an elegant way to accomplish complex operations.

The fundamental ideas of the Unix operating system have remained constant despite numerous distributions and variants over time. Unix remains a major power, whether in embedded systems, traditional server contexts, or the contemporary comeback of Unix-like operating systems in the shape of Linux distributions.

Essentially, Unix is more than just an operating system; it's a paradigm, a way of thinking, and a representation of the group effort to achieve computer excellence. Its influence can be seen in the concepts and ideas that have influenced the whole computing industry, in addition to the systems that immediately bear its name. The influence of Unix continues to be a guiding light for us as we traverse the rapidly evolving technological environment, affecting our methods for designing, implementing, and utilizing operating systems.

**REFERENCES**

Fernando, J. (2023b, July 27). *Assembly language*. Investopedia. https://www.investopedia.com/terms/a/assembly-language.asp#:~:text=An%20assembly%20language%20is%20a,to%20be%20readable%20by%20humans

Gaikwad, M. (2023b, February 10). Memory Management in UNIX. *Mithilesh’s Blog*. https://esymith.hashnode.dev/memory-management-in-unix

GeeksforGeeks. (2023b, November 6). *Unix File System*. https://www.geeksforgeeks.org/unix-file-system/

*Processes in unix*. (2013b, March 7). [Slide show]. PPT. https://www.slideshare.net/miau\_max/processes-in-unix

Sheldon, R., & Mixon, E. (2022b, February 9). *Unix*. Data Center. https://www.techtarget.com/searchdatacenter/definition/Unix?fbclid=IwAR0u8Kq8KYEm0vJOujg-aKa5ob2owo-Fg1L\_CjOexM7J\_QKvZEGbkjzVDRI

*Unix CPU Scheduling. https://ggn.dronacharya.info/ITDept/Downloads/QuestionBank/Odd/V%20sem/Lecture-8\_Section-B\_SPSA.pdf*

*Unix I/O System. https://www.cse.chalmers.se/edu/year/2011/course/EDA092/SLIDES/11-IO-systems.pdf?fbclid=IwAR33E5JRNQrb7xMMHRCW8iMZIz1pc\_A\_OaOzpIlJYdlwptfmMzZlBvuIfGs*

*Unix process states.png - Wikimedia Commons*. (2012b). https://commons.wikimedia.org/wiki/File:Unix\_process\_states.png

*Unix Storage Management*. (n.d.-b). SpringerLink. https://link.springer.com/book/9781590590294?fbclid=IwAR3tzVDxVPyOr84iy9AORrfvKYPLeKHIU3tpTWdmDhyH28eI2icaItXhKqI