**GNU**

**Operating System**

**ITP56**

**OPERATING SYSTEMS**

**TABLE OF CONTENTS**

CHAPTER 1 Introduction and History 04

CHAPTER 2 Process Management 06

26.1 Running a Command 07

26.2 Process Creation Concepts 08

26.3 Process Identification 09

26.4 Process Identification

26.5 Executing a File 10

26.6 Process Completion 11

26.7 Process Completion Status 12

26.8 BSD Process Wait Function

CHAPTER 3 CPU Scheduling 13

22.3.1 Absolute Priority 14

22.3.2 Realtime Priority 15

22.3.3 Basic Scheduling Functions 16

22.3.4 Traditional Scheduling 17

22.3.5 Limiting Execution to Certain CPUs

CHAPTER 4 Memory Management 19

3.2 Allocating Storage for Program Data

3.2.1 Memory Allocation in C Programs

3.2.2 The GNU Allocator 20

3.2.4 Allocation Debugging 21

3.2.5 Replacing Malloc

3.2.6 Obstrack 22

CHAPTER 5 Storage Management 23

Locating the Drive 23

Creating a Disk Label or Partition Table 24

Creating a Prtition

Naming a Partition

Create a File System

CHAPTER 6 I/O System 27

Input/Output on Streams 28

Low - Level Input/Output

File System Interface 29

Pipes and FIFOs

Sockets 30

Low - level Terminal Interface

CHAPTER 7 File System 31

14.1 Working Directory

14.2 Accessing Directories 32

14.3 Working with DIrectory Trees

14.4 Hard Links 33

14.5 Symbolic Links

04

CHAPTER 1

**INTRODUCTION**

GNU is pronounced as "g-noo" and stands for Gnu's Not Unix. It is an acronym that is recursive and stands for "Gnu's Not Unix." Richard Stallman founded the free and open-source GNU operating system in 1984. Although it has undergone significant changes over time, GNU is largely based on the Unix operating system. Changing from Unix, GNU operating system is a completely free software system. The GNU Project was announced in September 1983 by Richard Stallman while last March 1985, a longer version referred to as the GNU Manifesto was issued, and the translation into multiple other language have been made. There are three factors led to the choice for the name “GNU”. It was a real word, it was a recursive acronym for “GNU’s Not Unix” and it was entertaining to say or sing. The word “free” in “free software” pertains to freedom. There are four specific freedoms in using the GNU software; the freedom to run the program as you wish; the freedom to copy the program and give it away to your friends and co-workers; the freedom to change the program as you wish, by having the full access to source code; the freedom to distribute an improved version and thus help the community.

The goal of the GNU Project, which was founded in 1983, was to remove the barriers to cooperation posed by proprietary software owners in order to restore the cooperative spirit that once defined the computing community. Almost all software in the 1980s was proprietary, meaning its owners prohibited and prevented users from cooperating with it. The GNU Project became as a result. An operating system is essential for anyone using a computer; without one, proprietary software is the only way to even begin using a computer. Thus, it stands to reason that a free operating system would be the first item on the agenda for free software. Since the overall architecture was well-proven and

05

portable, and compatibility facilitates the transfer of unix users to GNU, they decided to make the OS compatible with UNIX. An operating system that resembles Unix comprises various components such as kernel, editors, compilers, text formatting, mail software, graphical user interfaces, libraries, games and much more. It is a very big task writing an operating system from scratch. They started in January 1984, while in October 1985, the free software foundation was established with the goal of raising money to support the development of GNU. With the exception of the kernel, we had either discovered or written all of the important parts by 1990. Then in 1991, Linus Torvalds developed Linux, a kernel that was similar to Unix, and released it as free software in 1992. The nearly finished GNU system and Linux were combined to create the GNU/Linux operating system. Tens of millions of people are thought to currently use GNU/Linux systems, usually through GNU/Linux distributions. Linux-libre, a modified free version of Linux maintained by free software activists, is a response to the presence of nonfree firmware "blobs" in the main Linux version. But the GNU Project is not just about the operating system at its core.

Their goal is to offer a wide range of software, catering to the needs of numerous users. Software applications are included in this. For a list of free software applications, visit the Free Software Directory. Additionally, they would like to offer software to non-techies. As a result, they created the GNOME graphical desktop to aid new users in utilizing the GNU system. They would also like to offer games and other leisure activities. There are already a ton of free games accessible. Except in cases where free software is restricted by laws like the patent system, there are no restrictions. The ultimate objective is to eliminate proprietary software by offering free software that can perform every task that computer users desire.

06

CHAPTER 2

**PROCESS MANAGEMENT**

The collection of actions involved in starting, stopping, and scheduling processes is referred to as process management in an operating system. One way to conceptualize a process management system is as an instance of a running program on a computer system. Any modern operating system must have process management in order for multiple programs to run simultaneously and effectively share system resources. The distribution of system resources, such as CPU time, memory, and input/output devices, to active processes is the responsibility of the process management operating system subsystem. Scheduling process execution to minimize response times and maximize system throughput is another duty of process management software. If one or more processes use up all of the system resources without proper management, the computer system may become unresponsive or even crash. Process management, therefore, is an essential part of any contemporary operating system, and the way it is designed and implemented greatly affects the system's overall functionality and performance.

The fundamental components for distributing system resources are called processes. Every process has a single thread of control and an address space of its own. A program is executed by a process; multiple processes can execute the same program, but each process runs the program independently of the others and has its own copy of the program within its own address space. The organization of processes is hierarchical. Every process has a parent process that was created specifically for it. A parent process's child processes are those that it created. Many of a child's characteristics are

07

inherited from the parent process. The creation, termination, and control of child processes by a program are covered in this chapter. In reality, there are three separate operations at perform: starting a new child process, executing a program in the new process, and synchronizing the child process's completion with the original program. The system function performs all three of these tasks automatically and offers a straightforward, portable mechanism for launching another program. If you want more control over the specifics of how this is done, you can perform each step separately using the primitive functions.

### 26.1 Running a Command

By using the system function, you can run another program more easily. All of the work involved in running a subprogram is handled by this function, but you don't have much control over the specifics—you have to wait until the subprogram ends before continuing. *Function: int system (const char \*command)* This function uses a shell command to carry out the command. The command in the GNU C Library is always executed using the default shell, sh. Specifically, it looks through the PATH directories to locate executable programs. If the shell process could not be created, the return value is -1; otherwise, it indicates the shell process's status. A return value of zero means that there isn't a command processor available if the command argument is a null pointer. A multi-threaded program's cancellation point is this function. If the thread allocates resources (such as memory, file descriptors, semaphores, or anything else) when the system is called, this is a problem. These resources are allocated until the program terminates even if the thread is canceled. Cancellation handlers should be used to protect system calls in order to prevent this. *The system function is declared in the header file stdlib.h.*

08

### 26.2 Process Creation Concepts

An overview of processes and the procedures needed to create a process and have it run a different program are provided in this section. When one of the functions posix\_spawn, fork, \_Fork, or vfork is called, a new process is created. The process of starting a new process is sometimes referred to as "forking" a process because of the name of the fork function (the system and popen also create new processes internally). A unique process ID, separate from the parent process's process ID, is assigned to each new process (also known as a child process or subprocess).

Both the parent and child processes carry on with their regular operations after forking a child process. You must specifically call wait or waitpid after the fork operation if you want your program to wait for a child process to complete its execution before proceeding. These functions provide you with limited details about the child's termination reason (such as the code for its exit). When the fork or \_Fork call returns, the newly forked child process keeps running the same program as its parent process. The program's return value from fork or \_Fork can be used to determine whether it is executing in the parent process or the child process. The usefulness of having multiple processes running the same program is rare. However, the child can use one of the exec functions to run a different program (see Executing a File). Its process image is the program that the process is running. When a new program is launched, the process loses all memory of its previous process image and exits along with the new program, rather than going back to the previous image.

### 09

### 26.3 Process Identification

A process ID number, which is a value of type pid\_t, is assigned to each process. Every newly created process is assigned a process ID. Over time, process IDs are repurposed. When a process terminates and its parent process waits on the process ID, the process lifetime of the parent process comes to an end. Only during a process's lifetime can a process ID uniquely identify itself (the parent process may implicitly make arrangements for such waiting to occur). This generally implies that the process needs to be active. Process IDs can also denote process groups and sessions. A thread ID is also assigned to Linux threads created with pthread\_create. The process ID for the entire process is the same as the thread ID of the first (main) thread. Subsequently created threads have unique thread IDs. The same numbering space as process IDs is used to assign them. Task IDs are another term that is occasionally used to refer to both process and thread IDs together. Since threads are never explicitly waited for, as opposed to processes, a thread ID can be reused as soon as the thread ends or is canceled. Not only does this apply to detached threads, but also to joinable threads. A thread group is given to each thread. The thread group ID is the process ID in the Linux version of the GNU C Library. Calling getpid will yield the process ID for that particular process. The process ID of the current process' parent—also referred to as the parent process ID—is returned by the function getppid. To use these functions, your program needs to include the header files sys/types.h and unistd.h.

### 26.4 Creating a Process

The fork function is the primitive for creating a process. It is declared in the header file unistd.h. *Function: pid\_t fork (void)* the fork function creates a new process. If the operation is successful, the parent and child processes will both see the fork return, but they will see different values: the child

10

process will see a value of 0, and the parent process will see the child's process ID. The parent process receives a value of -1 from the fork if the process creation attempt fails. Fork has defined the following errno error conditions:

**EAGAIN**

Either the user already has too many open processes, or there aren't enough system resources to start another one. This is going over the RLIMIT\_NPROC resource limit, which is typically able to be raised.

**ENOMEM**

The system cannot accommodate the amount of space needed by the process.

### 26.5 Executing a File

### The exec family of functions, which allows you to run a file as a process image, is covered in this section. These functions can be used to force a forked child process to run a new program. The only way the functions in this family vary is in how the arguments are specified, but other than that, they all perform the same task. The header file unistd.h contains their declarations. *Function: int execv (const char \*filename, char \*const argv[])* The file specified by filename is executed as a new process image by the execv function. The value of the argv argument is supplied to the main function of the program to be executed through an array of null-terminated strings called the argv argument. This array's final component needs to be a null pointer. Conventionally, the program's file name without directory names appears as the first element in this array.

### 11

### 26.6 Process Completion

The functions outlined in this section are used to ascertain the status of a child process and wait for it to stop or terminate. The header file sys/wait.h contains declarations for these functions. *Function: pid\_t waitpid (pid\_t pid, int \*status-ptr, int options)* To get status information from a child process whose process ID is pid, use the waitpid function. Usually, the calling process waits for the child process to terminate in order to provide status information. There are distinct interpretations for other values of the pid argument. A negative value, such as pgid, requests information for any child process whose process group ID is pgid. A value of 0 or WAIT\_MYPGRP requests information for any child process in the same process group as the calling process. Finally, a value of -1 or WAIT\_ANY requests status information for any child process. When a child process's status information is instantly available, this function returns without any delay. When status information is available for multiple eligible child processes, one of them is randomly selected and its status is promptly restored. You must call waitpid once more to obtain the status from the other eligible child processes. A bit mask is used as the options argument. The bitwise OR (also known as the "|") of zero or more of the WNOHANG and WUNTRACED flags should be its value. The WUNTRACED flag can be used to get status information from both stopped and terminated processes, while the WNOHANG flag can be used to signal that the parent process shouldn't wait. Unless status-ptr is a null pointer, the status information from the child process is kept in the object that it points to. A multi-threaded program's cancellation point is this function. If the thread allots resources (such as memory, file descriptors, semaphores, or anything else) at the same time that waitpid is called, then there is a problem. These resources are allocated until the program terminates even if the thread is canceled. Cancellation handlers should be used to protect waitpid calls in order to prevent this. The process ID of the child process whose status is reported is typically the return value.

12

Waitpid will block until a child process is detected if there are any but none of them are waiting to be noticed. However, waitpid will return zero rather than block if the WNOHANG option was specified. Waitpid will disregard any additional kids if it receives a specific PID to wait for. Consequently, waitpid will block or return zero as previously mentioned if there are children waiting to be noticed but the child for whom a PID was provided is not one of them.

### 26.7 Process Completion Status

Waitpid or wait will report the same status value of zero if the child process's exit status value is zero. The following macros can be used to check for additional types of information encoded in the returned status value. The header file sys/wait.h defines these macros.

### 26.8 BSD Process Wait Function

For BSD compatibility, the wait3 function is also provided by the GNU C Library. In sys/wait.h, this function is declared. It is the more adaptable wait4's predecessor. wait3 has been superseded. *Function: pid\_t wait3 (int \*status-ptr, int options, struct rusage \*usage)*

13

CHAPTER 3

**CPU SCHEDULING**

CPU scheduling is the process that makes full use of the CPU by allowing one process to use it while another's execution stops because of the unavailability of any resource, such as I/O. CPU scheduling aims to improve system speed, fairness, and efficiency. It also describes the transitioning between running processes. It is the cornerstone of systems with multiple programs. By maximizing CPU utilization, this switching makes the computer more productive. Preemptive and non-preemptive CPU scheduling are the two primary varieties.

The scheduling policy of the system and the CPU priorities of the processes determine which ones get CPU time when multiple processes need it at the same time. This section explains the decision-making process and the GNU C Library functions that govern it. People often simplify things by calling CPU scheduling just "scheduling" and referring to the CPU priority of a process as simply the "process," with the implication that it involves the CPU resource. However, it's essential to remember that processes don't exclusively rely on or compete for resources—CPU time is just a single component. In certain situations, CPU time may not even be highly significant. The effect of assigning a high "priority" to a process on its execution time in comparison to other processes can be minimal. Keep in mind that the priorities discussed in this section specifically address CPU time. Managing CPU scheduling is a multifaceted challenge, and diverse systems employ vastly different approaches to tackle it. Continuous innovation introduces new ideas that intricately weave into the fabric of various system scheduling algorithms. Within this section, we delve into overarching concepts, delve into particulars of systems frequently utilizing the GNU C Library, and explore established standards. To simplify discussions, we

14

often refer to CPU contention as if the system has only one CPU. However, the same principles hold true when a processor incorporates multiple CPUs. Understanding that the maximum number of processes running concurrently equals the number of CPUs allows for straightforward extrapolation of this information.

The functions outlined in this section are established by the POSIX.1 and POSIX.1b standards. Nevertheless, POSIX doesn't establish specific meanings for the values manipulated by these functions. This chapter adopts semantics derived from the Linux kernel's interpretation of the POSIX standard. Notably, the Linux implementation diverges significantly from the original intent of the POSIX syntax creators, as you'll observe.

#### 22.3.1 Absolute Priority

Each process is assigned a numerical value representing its absolute priority. A higher number signifies a higher absolute priority. Historically and in the majority of contemporary systems, all processes share an absolute priority of 0, making this section non-applicable. Absolute priorities were introduced to cater to real-time systems where certain processes must promptly respond to external events occurring in real time. In such cases, waiting for less critical processes to execute could be impractical. In situations where two processes vie for CPU usage simultaneously, the one with the higher absolute priority consistently gains access. This holds true even if the process with the lower priority is currently utilizing the CPU. It's important to note that we are specifically referring to processes that are actively running or in a "ready to run" state, indicating their immediate readiness to execute instructions. When a process enters a blocked state, such as waiting for I/O, its absolute priority becomes irrelevant in this context.

#### 15

#### 22.3.2 Realtime Scheduling

When two processes with identical absolute priorities are prepared to run, the kernel faces a dilemma, as only one can run at any given moment. If the processes share an absolute priority of 0, the kernel follows the decision-making process outlined in Traditional Scheduling. Otherwise, the decision adheres to the guidelines presented in this section. When two processes, each with distinct absolute priorities, are prepared for execution, the decision-making process is more straightforward and is elucidated in Absolute Priority. Every process is assigned a scheduling policy. For processes with absolute priorities other than zero, there are two options: **First Come First Served, Round Robin.** The most logical scenario occurs when all processes with a specific absolute priority adhere to the same scheduling policy, which we'll address first. In the **Round Robin policy**, processes take turns using the CPU, each running for a brief quantum of time before yielding to the next in a cyclic manner. Naturally, only processes that are both ready to run and share the same absolute priority participate in this rotation. On the other hand, in the **First Come First Served policy,** the CPU is allocated to the process that has been waiting the longest to run. It retains CPU access until it willingly releases it, completes its tasks, or is preempted by a higher-priority process. When a process absolutely must run at maximum CPU speed or not run at all, First Come First Served, coupled with the highest absolute priority and meticulous control of interrupts and page faults, is the ideal choice. Strategic use of sched\_yield function calls by processes employing the First Come First Served scheduling policy provides a balanced compromise between Round Robin and First Come First Served. To comprehend the intricacies of scheduling when processes with different policies coexist at the same absolute priority, a grasp of the detailed processes for entering and exiting the ready-to-run list is necessary. In both instances, the ready-to-run list functions as a true queue, with a process being added to the tail upon readiness and

16

removed from the head when the scheduler deems it ready for execution. It's crucial to note the exclusive nature of the "ready to run" and "running" states. Once a scheduler initiates a process, it is no longer ready to run or listed in the ready-to-run queue. When the process concludes its execution, it may revert to a ready-to-run state. The sole distinction between a process assigned the Round Robin scheduling policy and one assigned First Come First Serve is that, in the former, the process is automatically removed from the CPU after a specific time interval. Upon this occurrence, the process returns to the ready-to-run status, rejoining the queue at the tail. Notably, the time quantum in Round Robin scheduling is significantly brief—far shorter than traditional timesharing, exemplified by the Linux kernel's time slice being a thousand times shorter than its typical slice for conventional scheduling. Processes inherit their scheduling policy from their parent process initially, and this can be altered using functions outlined in Basic Scheduling Functions. It's essential to highlight that only a privileged process holds the authority to set the scheduling policy for a process with an absolute priority higher than 0.

#### 22.3.3 Basic Scheduling Functions

The GNU C Library functions for determining a process's absolute priority and scheduling policy are covered in this section. More functions to fine-tune the scheduling are available in Traditional Scheduling in the event that it is the scheduling policy. The naming and organization of these functions should not be taken too seriously. The functions are as defined by POSIX.1b, so they don't correspond with the ideas in this manual; however, the implementation on systems that make use of the GNU C Library is the opposite of what the POSIX structure expects. The POSIX scheme makes the assumption that the scheduling policy is the main scheduling parameter and that the priority value, if any, is a scheduling policy parameter. However, in practice, the scheduling policy merely serves to fine-tune the impact of the priority value, which is essential.

#### 17

#### 22.3.4 Traditional Scheduling

The scheduling of processes with an absolute priority of 0 is covered in this section. The scheduling described herein decides which of the great unwashed processes gets to use the leftover scraps of CPU time when the processes with higher absolute priority have used them all up.

#### 22.3.5 Limiting execution to certain CPUs

When a system has multiple processors, the operating system typically divides up the various processes that can be executed on each CPU in a way that maximizes system performance. With the scheduling functionality covered in the previous sections, you can control which processes and threads run to some extent. However, it is not covered which CPU ultimately runs which thread or process.

A program might want to have authority over this part of the system as well for a variety of reasons:

* A singular thread or process bears the responsibility for executing absolutely critical tasks, and it is imperative that under no circumstances should it be disrupted or impeded in its progress by other processes or threads utilizing CPU resources. In such a scenario, this specialized process is confined to a dedicated CPU, inaccessible to any other process or thread.
* The expenses associated with accessing specific resources, such as RAM and I/O ports, vary across different CPUs, particularly in NUMA (Non-Uniform Memory Architecture) machines. Ideally, memory access should be local, although this preference is often not apparent to the scheduler. Consequently, directing a process or thread to CPUs with local access to frequently used memory significantly enhances performance.
* In managed runtimes, tasks like resource allocation and bookkeeping (e.g., garbage collection) are optimized for performance in proximity to processors. This optimization proves beneficial in

18

minimizing locking costs, especially when resources do not need protection from concurrent accesses by multiple processors.

Currently, the POSIX standard does not offer much assistance in resolving this issue. A set of interfaces provided by the Linux kernel enable the specification of affinity sets for a process. The thread or process will be scheduled by the scheduler on the CPUs indicated by the affinity masks. The Linux kernel interface is partially followed by the interfaces defined by the GNU C Library.

19

CHAPTER 4

**MEMORY MANAGEMENT**

An operating system's ability to handle or manage primary memory and switch between main memory and the disk during the execution of a process is known as memory management. Every memory location is monitored by memory management, regardless of whether it is free or assigned to a specific process. It determines the amount of memory that each process should have. It determines when and which process will use memory. It keeps track of when memory is released or unallocated and updates the status accordingly.

**3.2 Allocating Storage For Program Data**

This section discusses how regular programs handle data storage, including the well-known malloc function and a few more upscale features exclusive to the GNU Compiler and C Library.

**3.2.1 Memory Allocation in C Programs**

C programs leverage variables to facilitate two types of memory allocation:

**Static Allocation:**

Occurs when declaring a static or global variable.

Involves defining a fixed-size block of space for each static or global variable.

Allocated once during program initiation as part of the exec operation.

The allocated space persists throughout the program's execution and is never released.

**Automatic Allocation:**

Takes place when declaring an automatic variable, such as a function argument or a local variable.

20

The space for an automatic variable is allocated when entering the compound statement containing the declaration.

The allocated space is released when exiting the compound statement, ensuring efficient memory usage during the variable's scope.

The automatic storage size in GNU C can be expressed as a variable. It needs to be a constant in other C implementations. Dynamic allocation, a third significant type of memory allocation, is accessible through GNU C Library functions but is not supported by C variables.

**3.2.2 The GNU Allocator**

The GNU C Library's implementation of malloc is based on ptmalloc, or pthreads malloc, which was derived from dlmalloc, or Doug Lea malloc. Two different methods of memory allocation are possible with this malloc function, depending on their size and user-specified parameters. The most common technique is to allocate memory chunks, or portions of memory, from a large, continuous memory region. These pieces are thoughtfully divided up to maximize productivity and reduce waste from leftover pieces. The system heap has historically stood for a single, sizable memory region. Nevertheless, the malloc implementation in the GNU C Library deviates from convention by preserving several of these regions, which are internally referred to as arenas. This design ensures effective memory resource allocation and management by optimizing memory utilization, especially in multi-threaded applications. Unlike other versions, the malloc function in the GNU C Library avoids rounding up chunk sizes to powers of two, whether for large or small allocations. Regardless of their size, neighboring chunks can be coalesced upon being freed. This unique approach makes the implementation versatile for various allocation patterns, minimizing memory waste typically associated with fragmentation. The inclusion of multiple arenas is a notable feature, enabling concurrent memory

21

allocation by multiple threads in separate arenas. This design enhancement enhances overall performance by facilitating simultaneous memory operations and resource utilization.

When allocating memory for very large blocks—much larger than a page—another technique uses mmap (either anonymously or through /dev/zero; see Memory-mapped I/O). One major benefit of this approach is that these large chunks are quickly reinserted into the system after being released. This avoids situations in which a sizable portion is "locked" among smaller ones, thereby preventing possible memory waste even after calling free. Depending on the allocation patterns of the program, the size threshold for choosing mmap dynamically changes. M\_MMAP\_THRESHOLD can be used to statically modify the threshold with mallopt for more precise control. Furthermore, if necessary, the option M\_MMAP\_MAX can be used to completely disable the use of mmap.

**3.2.4 Allocation Debugging**

It can be difficult to find memory leaks in programming languages that do not use garbage-collected dynamic memory allocation. Ensuring that dynamically allocated objects are appropriately released at the end of their lifecycle is vital for programs that run for a long time. If this isn't done, the system may eventually run out of memory. The malloc implementation in the GNU C Library offers straightforward methods to identify and locate memory leaks. To activate this feature, the application needs to be launched in a special debugging mode, which is triggered by setting an environment variable. Importantly, there are no speed penalties incurred by the program when the debugging mode is not enabled.

**3.2.5 Replacing malloc**

The GNU C Library offers the flexibility to substitute the default malloc implementation with an alternative allocator sharing the same interface. In dynamically linked programs, this substitution occurs through ELF symbol interposition, achieved via shared object dependencies or LD\_PRELOAD. In the case

22

of static linking, the replacement malloc library must be linked before linking against libc.a to ensure seamless integration.

**3.2.6 Obstacks**

An obstack serves as a memory pool that houses a stack of objects. Users can create multiple separate obstacks and allocate objects within designated obstacks. While the order of freeing within each obstack must adhere to the last-in, first-out principle, different obstacks operate independently of one another. Beyond the constraint of freeing order, obstacks are highly versatile: each obstack can accommodate any number of objects of varying sizes. Implemented with macros, obstacks typically facilitate rapid allocation, especially for smaller objects. The sole space overhead per object is the padding required to align each object appropriately.

23

CHAPTER 5

**STORAGE MANAGEMENT**

The methods and instruments that businesses use to keep up with a data storage system are collectively referred to as storage management. To assist businesses in striking a balance between expenses, performance, and storage capacity is the goal of storage management. Furthermore, the goal of storage management is to safeguard critical information while also making it more accessible across the entire enterprise.

According to Seth Kenlon, (April 1, 2021) In the 21st century, people often overlook the significance of data storage. With abundant and affordable options, including various types of storage and ample free cloud space, it's easy to take it for granted. Yet, when it comes to truly vital data, there's a distinct value in having a tangible hard drive. However, most hard drives don't come pre-configured off the shelf. Whether you're acquiring a new drive or adjusting the setup of an existing system, understanding how to partition a drive on Linux is a crucial skill. While a hard drive technically doesn't need extensive software to function as a storage device, operating it without modern conventions such as a partition table and filesystem can be challenging, impractical, and poses risks to your data's safety. These are the three important concepts that people need to know about hard drives:

* A partition table, also known as a disk label, is a metadata that is placed at the start of a drive and tells the computer what kind of storage is available and where it is located on the drive.

24

* A partition serves as a boundary that indicates where a filesystem is located. For instance, you can allocate a partition on a 512GB drive that takes up the full capacity, or you can make two partitions, each taking up 256GB, or you can make multiple partitions of different sizes.
* An agreed-upon storage structure between a computer and a hard drive is represented by a filesystem. In order to read, organize, and write data back to the filesystem in order to maintain data integrity, the computer must comprehend the filesystem.

The first two ideas—disk labels and partitions—are managed by the GNU Parted program. Although Parted has a basic understanding of filesystems, other tools, like mkfs, handle the complexities of filesystem implementation.

**Locating the drive**

Ensure that you know exactly where your drive is located on your system before using GNU Parted. Start by inserting the hard drive that you want to format into your computer. Then, use the parted command to look at the devices that are connected to your computer. The device most recently connected is assigned a name further down the alphabet than devices with longer attachment history. In this scenario, /dev/sdc is likely the recently attached drive, which can be verified by its size. Knowing that the USB thumb drive just connected is only 2GB (or close, say 1940MB), as opposed to the terabyte-sized main drives of the workstation, helps confirm the identity. If there's uncertainty, more details about the presumed drive for partitioning can be obtained. Certain drives offer varying degrees of metadata. This particular drive not only labels itself as a Yoyodyne drive, matching the physical branding, but also includes a discreet partition at the drive's beginning featuring some pre-installed software, followed by a FAT32 partition compatible with Windows. It is unequivocally the drive I plan to reformat.

25

**Creating a disk label or partition table**

A drive needs a disk label, also called a partition table, in order to create a partition on it. Parted accepts both terms. Use either the mklabel or mktable subcommand to start a disk label. Running this command removes any existing labels by creating a GPT label at the beginning of the drive designated as /dev/sdX. The replacement of partition-related metadata is what makes this process quick.

**Creating a partition**

Use the mkpart subcommand to create a partition on a drive. You can optionally specify the partition's name and its start and end points. If your drive only needs one partition, then the sizing is simple: start at 1 and end at 100%. Add the --align opt option to allow Parted to optimize partition boundary placement for better performance.

**Naming a partition**

Apart from indicating the filesystem associated with a partition, you have the option to assign a name to each partition. Certain file managers and utilities can interpret these partition names, facilitating drive identification. For instance, on my media workstation, where multiple drives are often connected for various projects, I assign names to both the partition and the filesystem. This ensures that, regardless of how I navigate my system, areas containing vital data are easily distinguishable.

**Create a filesystem**

To make your drive functional, it's essential to establish a filesystem within the new partition. GNU Parted doesn't handle this task as it exclusively functions as a partition manager. The Linux command for crafting a filesystem on a drive is mkfs, and there are convenient utilities aliased to help

26

create specific filesystem types. For example, mkfs.ext4 generates an EXT4 filesystem, while mkfs.xfs creates an XFS filesystem, and so forth. Since your partition resides "within" the drive, the process involves creating the filesystem in locations like /dev/sdX1 for the initial partition, /dev/sdX2 for the second partition, and so forth.

27

CHAPTER 6

**I/O SYSTEM**

I/O is pronounced "eye-oh," input/output, or I/O, is an a acronym that refers to all operations, programs, and hardware used in the transfer of data to and from a computer. Keyboards, mice, printers, and hard drives are examples of common I/O devices. Programming for computers relies heavily on input/output devices because they facilitate communication between computing devices connected to a network. I/O in computer architecture refers to a range of operations necessary for efficiently transferring data between various machines. Proper I/O operations are necessary for all data movement within a system, including text and video streams, software instruction sets, and audio files. The central processing unit (CPU) of the computer receives instructions from input/output signals, which it must follow in order to start data transfer. Human interaction, software, or hardware could be the source of the input. Data is transferred to a storage device via I/O input signals from a CPU, storage controller, or memory. I/O output signals are transferred from a computer to an output device in the meantime. Some I/O devices only operate as input-only devices—that is, they send data without receiving it. In contrast, output-only devices do not have the ability to transmit data to other machines; they can only receive input. Certain I/O devices can also receive input, process the data, and produce output. For almost all programs to carry out useful tasks, input (reading data) or output (writing data), frequently both, are needed. With so many input and output functions available in the GNU C Library, choosing the right function for a task can often be difficult.

This are the chapters relating to the GNU I/O facilities:

28

**Input/Output on Streams**

Functions for generating streams and performing input and output operations on them are covered in detail in this chapter. A stream is a high-level, relatively abstract concept that represents a communication channel to a file, device, or process, as was discussed in the Input/Output Overview section.

* **STREAM** - Because many library functions work with objects of type FILE \*, the C data structure representing a stream is named FILE rather than explicitly "stream" due to historical conventions. This has caused confusion in several C-related books. Occasionally, the term "file pointer" is employed interchangeably to refer to a "stream." But in this handbook, we strictly adhere to the technical differences, using the terms "file" and "stream" only in their exact contexts.
* **STANDARD STREAM** - When your program's main function is called, three predefined streams are open and available for use. The "standard" input and output channels set up for the process are represented by these streams.
* **OPENING STREAMS -** Utilizing the fopen function to open a file initiates the creation of a new stream and establishes a connection between the stream and a file. This process may include the creation of a new file. All elements discussed in this section are declared in the header file `stdio.h'.

**Low-Level Input/Output**

This chapter outlines functions designed for carrying out low-level input/output operations on file descriptors. These functions encompass the foundational components for the higher-level I/O functions detailed in Input/Output on Streams, along with functions tailored for low-level control operations lacking equivalents on streams.

While stream-level I/O is typically more flexible and convenient, descriptor-level functions come into play when necessary. Common scenarios include:

29

* Reading binary files in substantial chunks.
* Reading an entire file into memory before parsing.
* Performing operations, other than data transfer, achievable only with a descriptor (obtainable using fileno to convert a stream).
* Passing descriptors to a child process. (The child can establish its own stream based on an inherited descriptor but cannot directly inherit a stream.)

**File System interface**

This chapter delves into the functions within the GNU C Library that handle file manipulation. In contrast to the input and output functions (refer to Input/Output on Streams; see Low-Level Input/Output), these functions focus on operations related to the files themselves rather than their contents. Included in this chapter are functions for inspecting or altering directories, handling file renaming and deletion, and managing file attributes such as access permissions and modification times.

**Pipes and FIFOs**

Interprocess communication is facilitated by ***pipes***, which let data written by one process be read by another. First-in, first-out (FIFO) order applies to the data. Interestingly, the pipe has no name, is created for one use only, and requires inheritance from the single process that created it on both ends. A pipe and a FIFO special file are similar, but a ***FIFO*** special file is not a transient, anonymous connection. Like any other conventional file, a FIFO has one or more names. Processes open the FIFO using its name in order to communicate with one another. Both ends of a pipe or FIFO have to be open at the same time. The read operation returns end-of-file if you try to read from a pipe or FIFO file when no processes are writing to it (because of file closure or exit). On the other hand, writing to a pipe or FIFO without a

30

reading process is regarded as incorrect and results in a SIGPIPE signal. The operation fails with error code EPIPE if the signal is handled or blocked. Importantly, file positioning is not supported by pipes or FIFO special files. Sequential reading and writing operations take place, with reading commencing at the file's beginning and writing occurring at its conclusion.

**Sockets**

This chapter explores the GNU capabilities for interprocess communication through sockets. As a file descriptor, a socket is a flexible interprocess communication channel that functions similarly to a pipe. Sockets, as opposed to pipes, allow unrelated processes to communicate with one another and with processes on different machines via a network. The most common way to communicate between machines is via sockets, which are used by popular network applications like talk, ftp, rlogin, telnet, and rlogin. Sockets are not supported by every operating system, but consistency is guaranteed by the GNU C Library. Any operating system will contain the header file sys/socket.h, and socket functions are always present. However, these features will always fail if the system does not actually support sockets.

**Low-Level Terminal Interface**

Functions designed for terminal devices are described in this chapter. These functions make it possible to do things like turn off input echoing, set up serial line properties like flow control and line speed, and change characters used for signal transmission, end-of-file, command-line editing, and other control functions. This chapter covers most of the functions related to file descriptors. See the Low-Level Input/Output section for a more thorough explanation of file descriptors and how to open one for a terminal device.

31

CHAPTER 7

**FILE SYSTEM**

A File System is a crucial component of an operating system. A file system is a type of data structure that makes information and data easily retrievable from storage devices (floppy discs, hard drives, etc.). Although they use different file systems, all operating systems share some features. The technique for indexing files on the hard drive is the most crucial component of the file system. The OS can locate a specific file on the hard drive at any given time thanks to this index. The most common basis for this index is the file names. Different file systems and, consequently, different indexing techniques, exist depending on the operating system.

Everything is represented as a file in GNU/Linux. A file can be a stream of characters, or a collection of data (that is, a byte stream). A standard file may include code or text data. Users can read text files, and computers can read binary files containing code data. The functions for manipulating files in the GNU C Library are covered in this chapter. These functions deal with manipulating the files themselves instead of their contents, in contrast to the input and output functions. Functions for examining or changing directories, renaming and deleting files, and examining and setting file attributes like access permissions and modification times are just a few of the facilities covered in this chapter.

**14.1 WORKING DIRECTORY**

The current working directory, sometimes known as the working directory, is a directory that is linked to each process and is used to resolve relative file names. Shell commands such as cd can be used by users to modify the working directory. These commands, as well as other programs, use the primitives outlined in this section to examine and modify the working directory.

32

**14.2 ACCESSING DIRECTORIES**

You can view the contents of a directory file using the tools outlined in this section. If you want your program to list every file in a directory, maybe as a menu item, this is helpful. A directory stream with directory entries as its constituents is opened by the opendir function. Alternatively, if the program needs more control over how the directory is opened for reading, it can use fdopendir, which has advantages. For example, this enables the O\_NOATIME flag to be passed to open.

**14.3 WORKING WITH DIRECTORY TREES**

The functions for handling files in a directory that have been covered thus far have given you the option to process all of the files at once (see scandir) or to retrieve the information bit by bit. Processing entire directories and the files they contain can be helpful at times. To do this, two functions are defined in the X/Open specification. This function is available on SVID-derived systems because the simpler form is derived from an early definition in System V systems. The prototypes and necessary definitions are located in the header file ftw.h.

**14.4 HARD LINKS**

One file may have multiple names at once in POSIX systems. There is no preference among the names; they are all equally real. The link function can be used to give a file a name. Adding a new name to an existing file does not change its contents; rather, it merely gives the file a new name in addition to its current name or names.

33

**14.5 SYMBOLIC LINKS**

Soft links and symbolic links are supported by GNU systems. This type of "file" is really just a reference to another file name. Unlike hard links, symbolic links are unrestricted and can be created to directories or across file systems. A symbolic link to a name that isn't the name of a file can also be created. Similarly, even though the name no longer names any files, a symbolic link that was previously pointing to an existing file and later deleted will still point to the same file name. Symbolic links function as they do because of unique events that occur upon clicking on the link. When the open function detects that you have given a link's name, it reads the file name that is contained in the link and opens it instead. Similarly, rather than working on the link itself, the stat function refers to the file that the symbolic link points to. On the other hand, actions like renaming or deleting the file affect the link itself. Because their goal is to obtain information about the link, the functions readlink and lstat likewise avoid following symbolic links. link, the process that creates a solid link, does as well. It creates a solid connection to the symbolic connection, which is rarely desired.

34

**REFERENCES**

CHAPTER 1

[https://www.gnu.org/gnu/gnu-history.en.html](GNU HISTORY)

[https://www.techtarget.com/searchdatacenter/definition/GNU-General-Public-License-GNU- GPL-or-simply- GPL?fbclid=IwAR1cPhl0P2p6gH7nP0VZtFEc0sV0F1mpHaZdxl0J6TcVphuxXJc0PsZn oUM](GNU VERSION)

CHA[PTER 2

[https://datatrained.com/post/process-management- os/#:~:text=Process%20management%20refers%20to%20the,%2C%20and%20I%2FO%20devices.](Chapter 3)

[https://www.gnu.org/software/libc/manual/html\_node/Processes.html?fbclid=IwAR3aDumiHfh r5L6uPKmmK1nofUw-OMnpIQAJIU0ykbwo5hchQZLNG0iIlbU](Chapter 3)

CHAPTER 3

[http://gecnilokheri.ac.in/GPContent/operating%20system\_%20unit%20II.pdf](Chapter 3)

[https://www.gnu.org/software/libc/manual/html\_node/CPU-Affinity.html](Chapter 3)

[https://www.turing.com/kb/different-types-of-non-preemptive-cpu-scheduling-algorithms](Chapter 3)

CHAPTER 4

[https://www.tutorialspoint.com/operating\_system/os\_memory\_management.htm](chapter 4)

[https://www.gnu.org/software/libc/manual/html\_node/Memory-Allocation.html](chapter 4)

CHAPTER 5

[https://opensource.com/article/21/4/linux-parted-cheat-sheet](CHAPTER 5)

[https://www.indeed.com/career-advice/career-development/what-is-storage-management](CHAPTER 5)

CHAPTER 6

[https://www.techtarget.com/whatis/definition/input-output-I-O](CHAPTER 6)

[https://www.gnu.org/software/libc/manual/html\_node/I\_002fO-Overview.html](CHAPTER 6)

[https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html\_chapter/libc\_12.html#:~:text=This%20chapter%20describes%20the%20functions,file%2C%20device%2C%20or%20process.](CHAPTER 6)

CHAPTER 7

[https://e115.engr.ncsu.edu/file-systems/os-file-systems/#:~:text=An%20integral%20part%20of%20an,but%20all%20have%20similar%20features.](CHAPTER 7)

[https://www.gnu.org/software/libc/manual/html\_node/Symbolic-Links.html](CHAPTER 7)