CHAPTER 4

Operating System

Now that we have discussed the basics of computer hardware and software fundamentals, we will go over how they work together in this chapter. The operating system abstracts interaction to the HW and makes it efficient and convenient for software to leverage those HW resources.

When a computer turns on, the processor will execute the instructions that are presented to it; generally, the first code that runs is for the boot flow. For a computer that is used for general purposes and after it has booted up, there may be a variety of applications that need to be run on it simultaneously. Additionally, there could be a wide range of devices that could be connected to the computer (not part of the main system, for instance). All these need to be abstracted and handled efficiently and seamlessly. The user expects the system to "just work." The operating system facilitates all of this and more.

What Is an Operating System

An operating system, commonly referred to as the OS, is a program that controls the execution of other programs running on the system. It acts as a facilitator and intermediate layer between the different software components and the computer hardware as shown in Figure 4-1.

P. D. Crutcher et al., *Essential Computer Science*,
https://doi.org/10.1007/978-1-4842-7107-0_4

When any operating system is built, it focuses on three main objectives:

- Efficiency of the OS in terms of responsiveness, fluidity, and so on
- Ease of usability to the user in terms of making it convenient
- Ability to abstract and extend to new devices and software

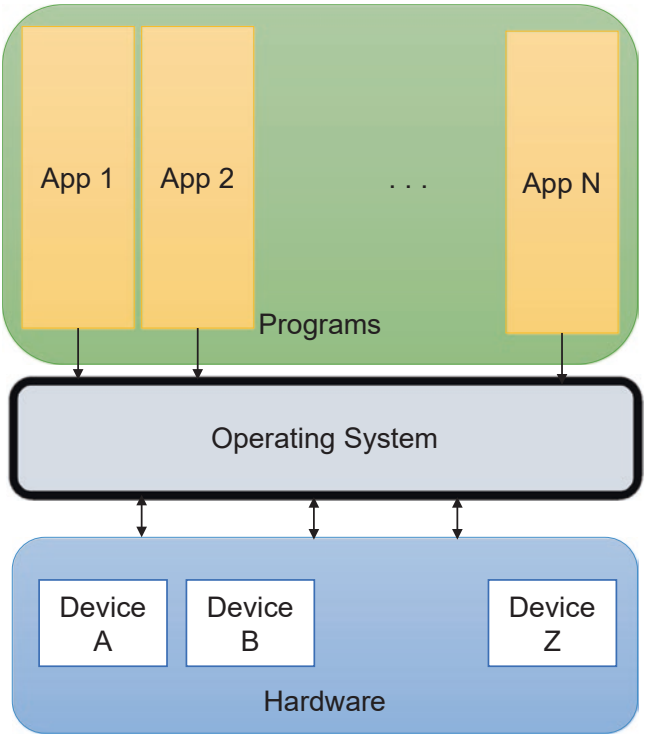

Figure 4-1. *High-Level Overview of an Operating System*

Let us take a quick look at how this is implemented. Most OSs typically have at least two main pieces:

- There is a core part that handles the complex, low-level functionalities and is typically referred to as the kernel.
- There are generally some libraries, applications, and tools that are shipped with the OS. For example, there could be browsers, custom features, frameworks, and OS-native applications that are bundled together.

Although the latter are generally referred to be a part of the OS, for the rest of our discussion, we will be focusing on the OS kernel.

Few common examples of operating systems that are prevalent are listed below. This list is not meant to be comprehensive but give the user a high-level idea of the list of operating systems that are commonly prevalent:

- Microsoft Windows
- GNU/Linux-based OS
- macOS (used for Apple's computers and client models)
- iOS (used for Apple's smartphone/tablet models)
- Android

All of these operating systems have different generations, versions, and upgrades. Some of the features supported across OS builds may also vary from time to time. However, in general, the core concepts discussed in this chapter are applicable to all of them.

OS Categories

The OSs can be categorized based on the different methods in use. The two most common methodologies are by the **usage type** and the **design/supported features** of the OS.

The first methodology is based on how the system is used. Based on this, there are five main categories:

1. Batch: For usages where a sequence of steps needs to be executed repeatedly without any human intervention. These classes are called batch OSs.
2. Time Sharing: For systems where many users access common hardware, there could be a need to time-share the limited resources. The OSs in such cases are categorized as time-sharing OSs.
3. Distributed: For hardware that is distributed physically and a single OS needs to coordinate their access, we call these systems distributed OSs.
4. Network: Another usage model, similar to the distributed scenario, is when the systems are connected over an IP (Internet Protocol) network and therefore referred to as network OSs.
5. Real Time: In some cases, we need fine-grained time precision in execution and responsiveness. We call these systems real-time OSs.

The second methodology is based on the **design and supported features** of the operating system. Based on this, there are three main categories:

1. Monolithic: In this case, the entire OS is running in a high-privilege kernel space and acts as the supervisor for all other programs to run. Common monolithic OSs include many of the UNIX flavors.

2. Modular: In some OSs, a few parts of the OS are implemented as so-called plug-and-play modules that can be updated independent of the OS kernel. Many modern OSs follow this methodology, such as Microsoft Windows, Linux flavors, and macOS.

3. Microservice based: More modern OSs are emerging and leverage the concept of microservices where many of the previously monolithic OS features may be broken down into smaller parts that run in either the kernel or user mode. The microservice approach helps in assigning the right responsibility of the components and easier error tracking and maintenance. Some versions of Red Hat OS support microservices natively.

Why We Need an OS

As we discussed before, the OS needs to be able to facilitate different applications running on the system. For example, consider an application that wants to play music from the file system and another application that needs to create a file and write to the disk. In both these cases, these applications need to access storage, must be able to render/display some content on the screen, and may need to access additional devices on the system.

Let us consider two very different approaches to enabling the preceding example. One approach could be that each of the applications will run directly on the HW with no OS abstraction; in this case, they must each implement all of the required functionality including hardware access and resource management on their own. This approach has some obvious challenges. One problem is the size of the resultant programs; they must have code for their application logic as well as all of the lower-level code

for accessing hardware. This will increase the number of defects in the code and increase the time it takes to get the application working correctly. Another problem is that the application may not be able to handle all types of hardware and devices. For example, the application would need to encode specific functions to support a given storage device, but another storage device on a slightly different system may be different enough that the application will fail there. Also, with this approach, you would not be able to run the two applications at the same time; they would need to run sequentially, since there is no mechanism to allow two programs to run at the same time in this scenario. Another, more mainstream approach would be for a common program to facilitate all the interactions with the hardware, handle complexities that happen underneath, and provide an abstraction for the applications to interact to. This allows the applications to focus on their business logic, reducing the size and complexity of the resultant application, which also gets the application written and validated much faster.

Before we can decide which is a better approach, let us take a similar analogy with a building construction company that is developing a new gated community. As part of the community, there could be many houses that need to be built. For each of these houses, there could be several common requirements such as water piping, electricity lines, drainage system, and so on that may be needed. Each of the individual houses may handle these on its own and have its own separate blueprints for water, drainage, communication, and so on. But it doesn't scale. With this example, we can see that this is inefficient and often messy in terms of provisioning the lines and piping as well as supporting and maintaining them, in the long term. The best practice here is for the engineering team to streamline these via a central pipeline and then branch off from the central line to the individual houses as per the requirements. This not only saves cost, it is easier to maintain and manage and is less error-prone. The same concept can be applied for the case of a computing device, where

the OS manages and streamlines usage of hardware resources and allows multiple applications to run in parallel with each other.

In practice, there are many common features that may be needed by your programs including, for example, security, which would have services like encryption, authentication, and authorization, to name a few. It makes sense for these kinds of capabilities to be provided by the operating system, so they can be leveraged consistently by all.

Purpose of an OS

As a precursor to this section, consider a common home appliance such as a dishwasher. The appliance supports a set of functionalities that is usually predefined (more modern systems may additionally have some programmability) in manufacturing. Such modern appliances have microprocessors with their runtime code already loaded and configured so that they "know" exactly what to do. Here, the complete programming logic is embedded into a non-volatile memory that is later executed using a microcontroller. It still has complexities in terms of reliability, error handling, and timing. However, the environment and the variabilities are quite contained within the appliance.

In the case of a general-purpose computing device, as we discussed earlier, there are varying needs in terms of the underlying hardware, the applications that need to run on the system, and the support for different users. At a high level, many of these are not deterministic in nature and could vary from one system to another. The purpose of the operating system is to ensure that it abstracts the HW and facilitates the seamless execution of our applications using the system. Now, we will take a more detailed look at the different complexities on such systems and how the OS handles them.

Complex and Multiprocessor Systems

Many modern computing architectures support microprocessors with multiple CPU cores. On higher-end systems, there could even be multiple sockets each able to host a microprocessor (with several cores). Typically, when all cores provide the same or identical capabilities, they are called as homogeneous platforms. There could also be systems that provide different capabilities on different CPU cores. These are called heterogeneous platforms. There are also additional execution engines such as Graphics Processing Units (GPUs), which accelerate graphics and 3D processing and display, for instance. An operating system supporting such a platform will need to ensure efficient scheduling of the different programs on the different execution engines (cores) available on the system. Similarly, there could be differences in the hardware devices on the platform and their capabilities such as the type of display used, peripherals connected, storage/memory used, sensors available, and so on. It may not be possible to release a new OS for every new system configuration. Hence, the OS would also be required to abstract the differences in the hardware configurations to the applications.

Multitasking and Multifunction Software

There is also an increasing need to use computers for multiple tasks in parallel. Let's build on the same example that we had before where a user may want to play music and also create a content and write a file at the same time. In general, there could be many such applications that may need to be running on the system at the same time. These could include applications that the user initiated, so-called "foreground" applications, and applications that the OS has initiated in the background for the effective functionality of the system. It is the OS that ensures the streamlined execution of these applications.

Multiuser Systems

Often, there could be more than one user of a system such as an administrator and multiple other users with different levels of access permission who may want to utilize the system. It is important to streamline execution for each of these users so that they do not find any perceived delay of their requests. At the same time, there need to be controls in place to manage privacy and security between users. The OS facilitates and manages these capabilities as well.

As we discussed earlier, in general, there are various dynamic scenarios on the platform, and it is the role of the operating system to handle these in a consistent, safe, and performant manner. Most general-purpose OSs in use today, such as Windows, Linux, macOS, and so on, provide and handle most of the preceding complexities. Figure 4-2 shows a slightly detailed view of an abstract operating system.

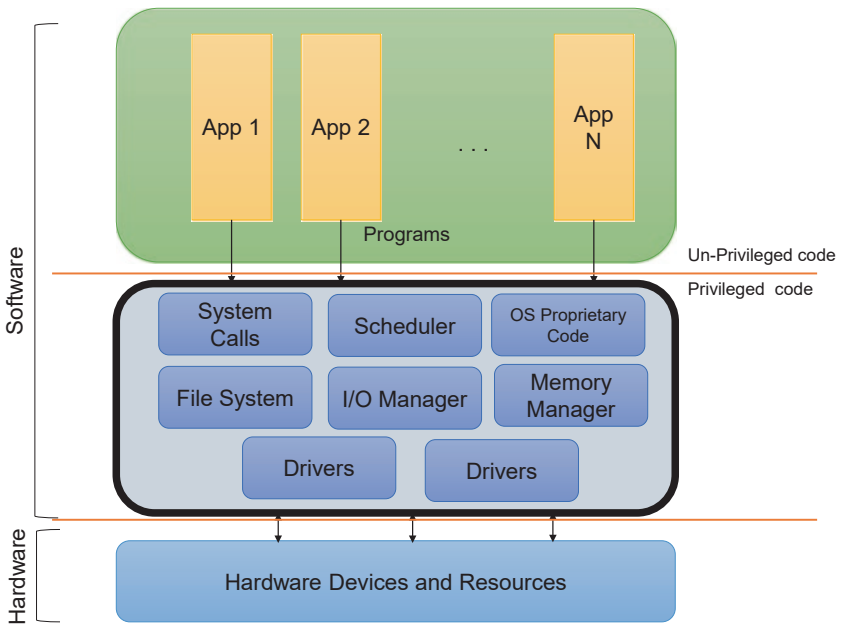

Figure 4-2. *Operating System Components*

As we can see here, it supports multiple different hardware, supports co-existence of multiple applications, and abstracts the complexities. The OS exposes different levels of abstractions for applications and drivers to work together. Typically, there are APIs (application programming interfaces) that are exposed to access system resources. These APIs are then used by programs to request for communicating to the hardware. While the communication happens, there could be requests from multiple programs and users at the same time. The OS streamlines these requests using efficient scheduling algorithms and through management of I/Os and handling conflicts.

Why Is It Important to Know About the OS?

Software developers must have a good understanding of the environment, the OS, that their code is running in, or they won't be able to achieve the things they want with their program. As you, a software developer, go through the stages of development, it is important for you to keep in mind the OS interfaces and functionality as this will impact the software being developed.

For a given application, the choice of language and needed runtime features may be OS dependent. For example, the choice of inter-process communication (IPC) protocols used for messaging between applications will depend on the OS offerings.

During development and debug, there could be usages where the developer may need to understand and interact with the OS. For example, debugging a slowly performing or nonresponsive application may require some understanding of how the OS performs input/output operations. Here are some questions that may come up during the debug:

- Are you accessing the file system too often and writing repeatedly to the disk?
- Is there a garbage collector in place by the software framework/SDK?

- Is the application holding physical memory information for too long?
- Is the application frequently creating and swapping pages in memory? What it the average commit size and page swap rate?
- Is there any other system event such as power event, upgrades, or virus scanning that could have affected performance?
- Is there an impact on the application based on the scheduling policy, application priority, and utilization levels?

If the application needs to interface with a custom device, it will most likely need to interface some low-level functionality provided by the OS. For example, if there was a custom device that is connected to the system, the application would need to use the OS-provided API for communication. As a software developer, it may be required to understand these APIs and leverage the OS capabilities. There could also be a need to follow certain standard protocols provided by the OS for authenticating a given user of your application to grant permissions and access.

The list can grow based the variety of applications and their intended usages. As we discussed before, the design considerations for the OS must leverage appropriate abstraction and separation of concerns between different hardware and users. Also, most OSs are tuned and optimized for some common use cases, based on expected use. From a software developer point of view, it is important to be aware of some of these and leverage the configuration knobs and built-in tools provided by the OS.

Responsibilities of an OS

As we have seen in the previous sections, the OS needs to be able to abstract the complexities of the underlying hardware, support multiple users, and facilitate execution of multiple applications at the same time. In Table 4-1, we articulate some of these requirements and discuss how an OS can achieve them.

Table 4-1. *Requirements and Solutions*

Requirement	Solution
Applications require time on the **CPU** to execute their instructions.	The OS shall implement and abstract this using suitable **scheduling** algorithms.
Applications require access to system **memory** for variable storage and to perform calculations based on values in memory.	The OS shall implement **memory management** and provide APIs for applications to utilize this memory.
Each software may need to access different **devices** on the platform.	The OS may provide APIs for **device and I/O management** and interfaces through which these devices can be communicated.
There may be a need for the user or applications to save and read back contents from the **storage**.	Most OSs have a **directory and file system** that handles the storage and retrieval of contents on the disk.
It is important to perform all of the core operations listed in the preceding **securely** and efficiently.	Most OSs have a **security subsystem** that meets specific security requirements, virtualizations, and controls and balances.
Ease of access and usability of the system.	The OS may also have an additional GUI (graphical user interface) in place to make it easy to use, access, and work with the system.

To summarize, the OS performs different functions and handles multiple responsibilities for software to co-exist, streamlining access to resources, and enabling users to perform actions. They are broadly classified into the following functional areas:

- **Scheduling**
- **Memory management**
- **I/O and resource management**
- **Access and protection**
- **File systems**
- **User interface/shell**

The remainder of this part of this chapter will look at the preceding areas one by one.

Scheduling

One of the primary functionalities of the OS would be to provide the ability to run multiple, concurrent applications on the system and efficiently manage their access to system resources. As many programs try to run in parallel, there may be competing and conflicting requests to access hardware resources such as CPU, memory, and other devices. The operating system streamlines these requests and orchestrates the execution at runtime by scheduling the execution and subsequent requests to avoid conflicts.

Before we go into the details of scheduling responsibilities and algorithms, it is important to know some background about the basic concepts of program execution, specifically processes and threads.

Program and Process Basics

When a software developer builds a solution, the set of capabilities it provides is usually static and embedded in the form of processed code that is built for the OS. This is typically referred to as the program. When the program gets triggered to run, the OS assigns a process ID and other metrics for tracking. At the highest level, an executing program is tracked as a process in the OS. Note that in the context of different operating systems, jobs and processes may be used interchangeably. However, they refer to a program in execution.

Process States

When a program gets triggered for execution, typically say using a double click of the EXE (or using a CreateProcess() API in Windows), a new process is created. A process typically supports multiple states of readiness in its lifecycle. The following diagram captures some generic process execution states.

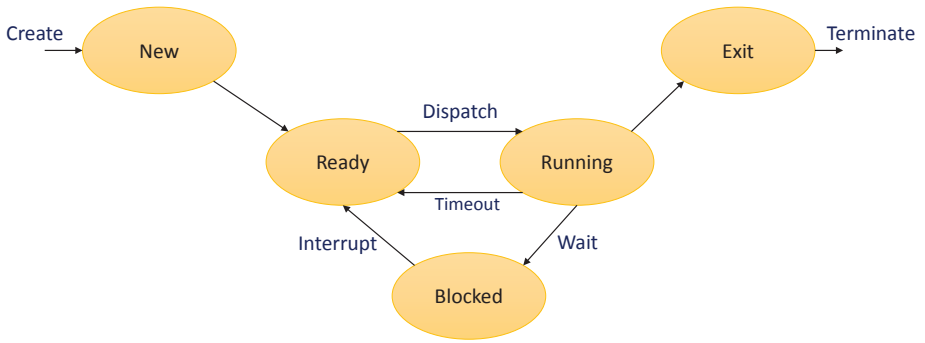

Figure 4-3. *Process States and Transitions*

As we can see in Figure 4-3, the process begins "life" in the **New** state just after it is created. From there it may move to other states, with the next state typically being the **Ready** state, where it is waiting for the

OS to assign a CPU to run on. The OS has a scheduler that takes care of selecting a process from a list of processes to be executed. Once selected, the dispatcher comes in that ensures the process selected gets time on the CPU. At this point, the process moves to the **Running** state. There could be a case when a process is running on the CPU, but may not have completed its job. The OS would also have to ensure other processes on the system get a fair share of time on the CPU. So the OS continues to execute the process on the CPU till a "timeout" is reached. After which, the process could be moved back to the Ready state waiting to be dispatched. This sequence of steps can continue to happen. At a later point, if the process is waiting on a device I/O, say a disk, it could be moved to the **Blocked** state if the device is busy. The same process continues till the process gets terminated and moves to the **Exit** state.

Note that there could be more than one CPU core on the system and hence the OS could schedule on any of the available cores. In order to avoid switching of context between CPU cores every time, the OS tries to limit such frequent transitions. The OS monitors and manages the transition of these states seamlessly and maintains the states of all such processes running on the system.

Process Control Block (PCB)

The OS has a well-defined data structure through which it manages different processes and their states. It is called as the Process Control Block (PCB). As we can see in Figure 4-4, the PCB includes all information that is required to manage and monitor the process. It includes details such as the unique identifier of the process, current state, and other details pertaining to accounting and scheduling. It may also store the processor register details, program counter (which contains the address of the next instruction to be executed), and memory information. All these are required to execute the process and also save the context of the process when it is moved from one state to the other as we discussed previously.

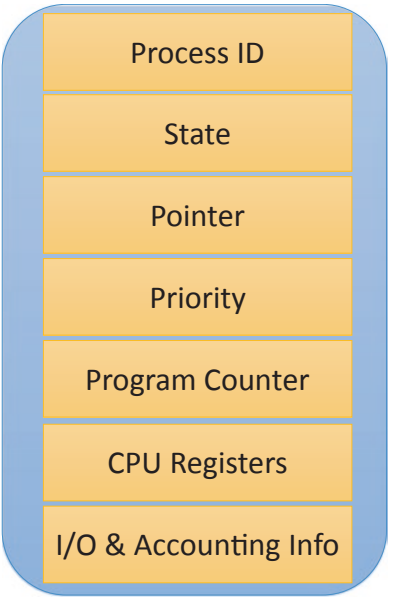

Figure 4-4. *Process Control Block (PCB) Representation*

- The **process ID** is a unique identifier for the instance of the process that is to be created or currently running.
- The **process state** determines the current state of the process, described in the preceding section.
- The **pointer** could refer to the hierarchy of processes (e.g., if there was a parent process that triggered this process).
- The **priority** refers to the priority level (e.g., high, medium, low, critical, real time, etc.) that the OS may need to use to determine the scheduling.
- **Affinity and CPU register details** include if there is a need to run a process on a specific core. It may also hold other register and memory details that are needed to execute the process.

- The **program counter** usually refers to the next instruction that needs to be run.
- **I/O and accounting** information such as paging requirements, devices assigned, limits, and so on that is used to monitor each process is also included in the structure.

There could be some modifications to how the PCB looks on different OSs. However, most of the preceding are commonly represented in the PCB.

Now that we have looked at how a process is represented in the OS and how the OS maintains the context of different processes, we will look at how the OS supports multitasking and how these processes are scheduled.

Context Switching

The operating system may need to swap the currently executing process with another process to allow other applications to run, if the current process is running for too long (preventing other processes/applications from running). It does so with the help of context switching.

When a process is executing on the CPU, the process context is determined by the program counter (instruction currently run), the processor status, register states, and various other metrics. When the OS needs to swap a currently executing process with another process, it must do the following steps:

1. Pause the currently executing process and save the context.
2. Switch to the new process.
3. When starting a new process, the OS must set the context appropriately for that process.

This ensures that the process executes exactly from where it was swapped. With CPUs running at GHz frequencies, this is typically not perceivable to the user. There are other hardware interfaces and support

to optimize these. For example, the time taken to save and restore context could be automatically supported in certain hardware, which could improve the performance further.

Scheduling

The most frequent process states are the Ready, Waiting, and Running states. The operating system will receive requests to run multiple processes at the same time and may need to streamline the execution. It uses process scheduling queues to perform this:

1. Ready Queue: When a new process is created, it transitions from New to the Ready state. It enters this queue indicating that it is ready to be scheduled.
2. Waiting Queue: When a process gets blocked by a dependent I/O or device or needs to be suspended temporarily, it moves to the Blocked state since it is waiting for a resource. At this point, the OS pushes such process to the Waiting queue.
3. In addition, there could be a **Job queue** that maintains all the processes in the system at any point in time. This is usually needed for bookkeeping purposes.

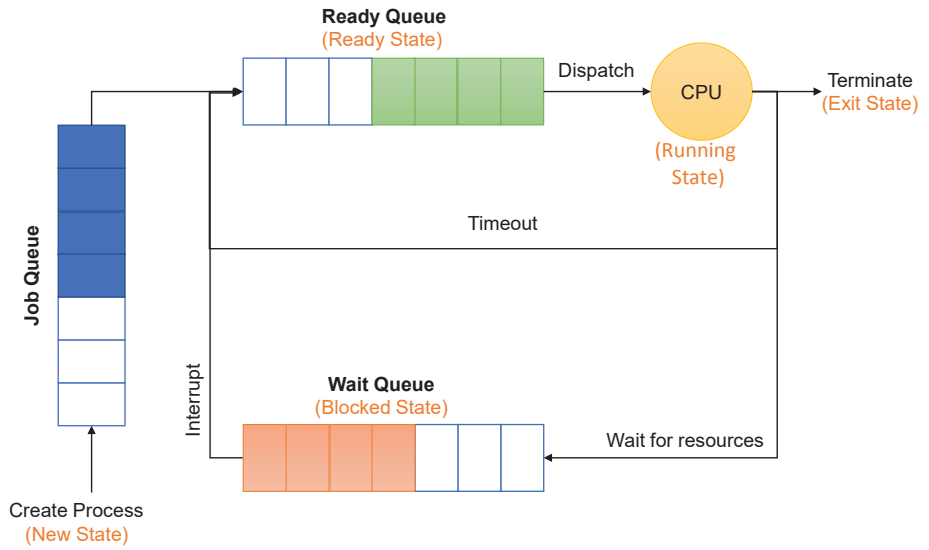

Figure 4-5. *Scheduling Flow in a Typical OS with Different Process States*

As we can see in Figure 4-5, all processes go through the Job queue and are waiting to be dispatched for execution. Once they are assigned CPU time, they get scheduled to run on the CPU for a specific time period. This is called as the quanta of time for which the process gets to run on the CPU. Once that time period is elapsed, the process is moved back to the Ready queue, where it waits to be scheduled again, until it has completed its task. If the process is running and gets blocked waiting on some I/O or an external event, the OS moves the process to the Waiting queue so that it is not wasting time on the CPU. This process of Ready -> Schedule -> Wait continues till the process completes its task, at which time it moves to the Exit state and gets released.

Typically, any process can be compute or I/O intensive depending on what kind of problem it is trying to solve. As a software developer, it is important for you to balance these requirements and optimize the code, perhaps utilizing threads, locks, and critical sections appropriately for best behaviors.

Scheduling Criteria

Most operating systems have predefined criteria that determine the scheduling priorities. Some of them have a criterion to provide maximum throughput and utilization of the CPU effectively, while others may have a higher preference to minimize the turnaround time for any request that comes to the scheduler. Often, most general-purpose operating systems provide a balance between the two and are usually tuned to the general workload needs. There may be additional power and performance settings that can be tuned to modify these behaviors.

Some of the typical metrics that the OS may use to determine scheduling priorities are listed in the following:

- CPU Utilization and Execution Runtime: The total amount of time the process is making use of the CPU excluding NOP (no-operation) idle cycles.
- Volume/Execution Throughput: Some OSs may need to support certain execution rates for a given duration.
- Responsiveness: The time taken for completion of a process and the average time spent in different queues.
- Resource Waiting Time: The average time taken on external I/Os on the system.

Based on these criteria and the strategic needs for the OS, the scheduling behavior of the system is defined.

Note Most OSs try to ensure there is fairness and liveness in scheduling. There are various scheduling algorithms like First Come, First Serve (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Round-Robin, Static/Dynamic Priority, and so on that the OS uses for scheduling of processes.

Thread Concepts

Now that we have looked at how the process works and how the OS manages the scheduling of a process, we will look at an interesting concept called threads. A thread is nothing more than a lightweight process. When a process gets executed, it could create one or more threads internally that can be executed on the processor. These threads have their own program counter, context, and register information, similar to how the process is managed.

Threads help in performing parallelism within the same process. For example, if we have a simple form application that is executed, it typically starts with a main thread on which the user interface is running. Let's assume we need to read some content that may take a while to load. This could cause the main thread to be blocked preventing the user from interacting with the application. However, if the call is made asynchronously, on another thread, the main thread can continue to run while the content read is happening. This not only improves performance, it also enhances the user experience. Note that all of this happens within the context of the same process.

Let us consider an example of a process that contains a single thread vs. the same process with multiple threads. As we can see in Figure 4-6, the parallel execution across threads happens within the context of the same process. Even if one thread in a process may be blocked, the other thread could continue its execution. Overall, this helps in completing the job faster. Since threads run within the context of a process, they relatively consume lesser system resources than processes as well.

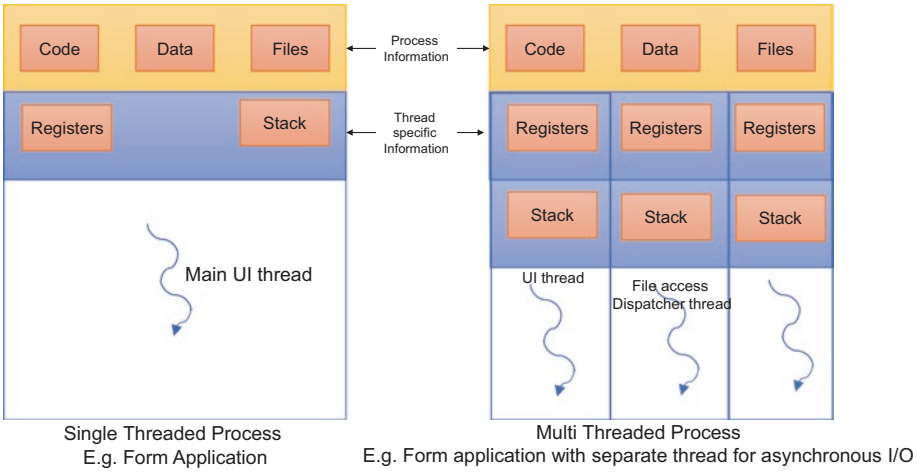

Figure 4-6. *Single- vs. Multi-threaded Process for a Simple Form Application*

The OS may employ different types of threads, depending on whether they are run from an application. For instance, an application may leverage **user-mode threads**, and a kernel driver may leverage **kernel-mode threads**. The OS also handles switching from user-mode threads to kernel-mode threads as required by a process.

Memory Management

In systems with multiple programs running in parallel, there could be many processes in memory at the same time, and each process may have specific memory needs. Processes may need memory for various reasons. First, the executable itself may need to be loaded into memory for execution. This is usually the instructions or the code that needs to be run. The second item would be the data part of the executable. These could be hardcoded strings, text, and variables that are referenced by the process. The third type of memory requirement could arise from runtime requests for memory. These could be needed from the stack/heap for the program to perform its execution.

Further, the operating system may also have its memory requirements. The OS and the kernel components may also need to be loaded in memory. Additionally, there may be a specific portion of memory needed for specific devices. For example, memory-mapped (discussed later) data for a specific device may need to be carved out and handled separately.

Like many other resources, the OS also needs to ensure efficient usage of memory. This is usually handled by the memory management subsystem. It manages various functions including allocation of new memory requests, translation of physical to virtual memories, swapping data pages, protection of specific memory pages, and so on. It may also need to manage and abstract the underlying hardware differences including memory controller intricacies and memory layout specifics. We will cover some of these topics in this section. Before we can get into the details, let's cover some basic concepts.

Address Binding

Consider a short line of pseudo-code (A = B + 2) that adds 2 to variable "B" and assigns this to variable "A". When this line gets compiled, it gets translated into a few steps. The first step would be to read the value of B from memory. The next step would be a simple mathematical calculation to add value 2 to B and perhaps store this in the accumulator. The final step would be to copy back this value and write this back to the memory location referenced by A. As we can see here, there are multiple references to read from memory and write back to memory, also, not shown here, involving the CPU registers. If these A and B are fixed memory locations like in the case of a traditional embedded system, these locations may not change. However, in the case of a general-purpose operating system, it becomes difficult to assign a location in memory that is static from run to run or even for the duration of one run.

To solve this problem, the common solution is to map the program's compiled addresses to the actual address in physical memory. In the simplest case, each program would get its own physical memory. This ensures that multiple programs can co-exist at the same time. This address binding can be done in multiple ways:

1. The address locations could be fixed at compile time. That is, the base address or the starting address of a program can be fixed while compiling, and the rest of the locations are referenced from that. This is not advisable since the fixed base address may not be available if another program is using it or may call for unexpected security violations.

2. The relative address of the program could be calculated at the time the program is loaded. A typical usage model would be to calculate this at runtime using a translation layer, which maps the program address to the real physical address. This is typically handled by the memory controller and is usually the most flexible option. Most operating systems and compilers also default to this mode for security reasons to change the base address at every launch.

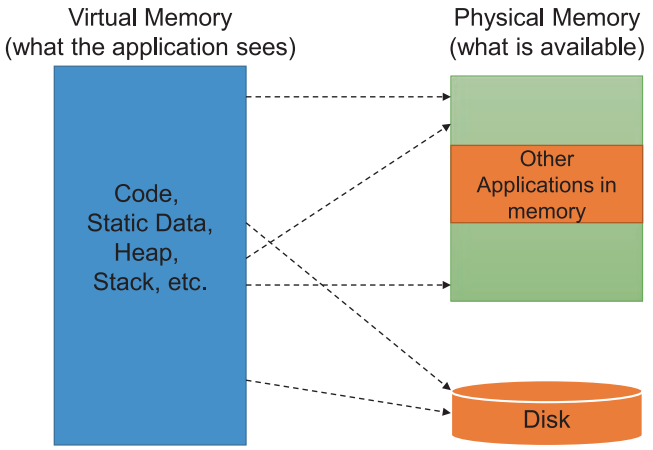

Figure 4-7. *Virtual Memory to Physical Memory Overview*

The address that the program has access to is usually referred to as the virtual address, and the actual location in memory is the physical address on the system. This could refer to a physical location on the RAM. As we can see in Figure 4-7, the application sees its code, static data, the variables, the stack, and so on. However, internally, the memory controller and the OS translate these to a location in physical memory. Not everything that the application sees may be residing in physical memory all the time. Also, at times, certain parts of the data could also be retrieved from storage such as disks. In the next section, we will look at how a simple translation happens between virtual memory and physical memory.

Logical vs. Physical Address

A program will have variables, instructions, and references that are included as part of the source code. The references to these are usually referred to as the symbolic addresses. When the same program gets compiled, the compiler translates these addresses into relative addresses.

This is important for the OS to then load the program in memory with a given base address and then use the relative address from that base to refer to different parts of the program. At this time, the OS can make use of the physical address mapping to refer to specific locations in memory. This is depicted in Figure 4-8 where the relative address is calculated using the base address and the offset.

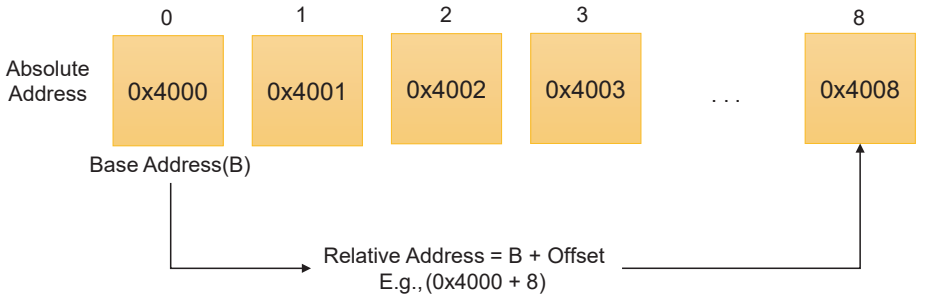

Figure 4-8. *Absolute, Base, and Relative Address Concepts*

In general, there is not enough physical memory to host all programs at the same time. This leads to the concept of virtual memory that can be mapped to physical memory. The memory management unit is responsible for translating virtual addresses to physical addresses. Typically, most OSs have a page table, which is like a lookup table, that is used to translate virtual addresses to a physical address at runtime. When the contents that need to be referred are outside the page, the memory content is then swapped to the new page at runtime. As shown in Figure 4-9, an unwanted page is usually identified and moved out to the secondary disk. Then, the required page is moved into memory to continue with the execution.

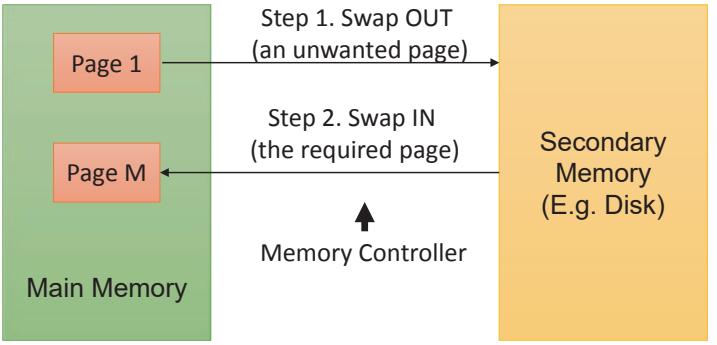

Figure 4-9. *Page Swapping Example*

Inter-process Communication

It is often desirable to have processes communicate with each other to coordinate work, for instance. In such cases, the OS provides one or more mechanisms to enable such process-to-process communication. These mechanisms are broadly classified as inter-process communication (IPC). There are many ways IPCs can be implemented. The two common ways are explained in the following, which involve shared memory and message passing.

Shared Memory Method

When two or more processes need to communicate with each other, they may create a shared memory area that is accessible by both processes. Then, one of the processes may act as the producer of data, while the other could act as the consumer of data. The memory acts as the communication buffer between these two processes. This is a very common mechanism to communicate between processes. This is depicted in Figure 4-10.

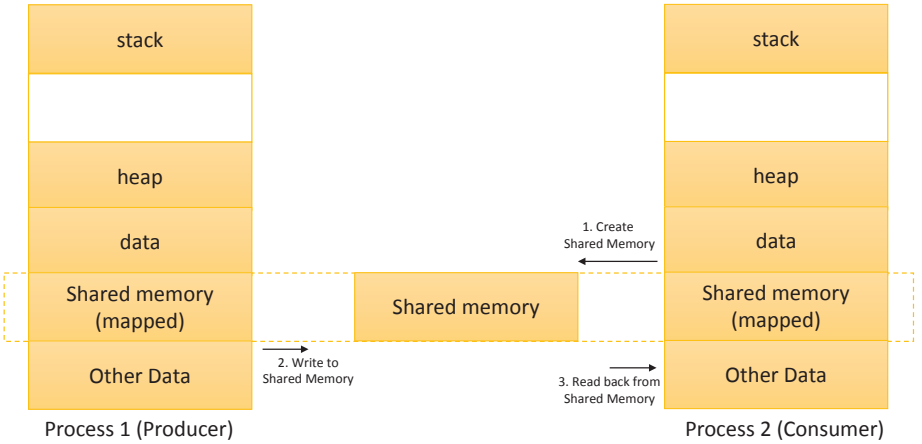

Figure 4-10. *Simple Shared Memory–Based Inter-process Communication*

There are additional details on the timing, creation of memory itself, permissions, and so on. However, we will not cover the details in this book.

Message Passing Method

The other method is called message passing where the two processes have a predefined communication link that could be a file system, socket, named pipe, and so on and a protocol-based messaging mechanism that they use to communicate.

Typically, the first step would be to establish the communication channel itself. For example, in the case of a TCP/IP communication, one of the processes could act as the server waiting on a specific port. The other process could register as a client and connect to that port. The next step could involve sharing of messages between the client and server using predefined protocols leveraging Send and Receive commands. The processes must agree on the communication parameters and flow for this to be successful. Given this, they can communicate until the IPC is terminated by either of the process. This is a common communication mechanism that is used by networking applications as well.

Further Reading

The memory management unit forms a critical part of the operating system. Additionally, some OSs use Translation Lookaside Buffers (TLBs), which contain page entries that have been recently used, multilevel page tables, and page replacement algorithms to perform optimal memory management depending on the needs. The performance, thrashing of memory, and segmentation needs vary from one OS to another. Some of these concepts are covered by the references shared later in this chapter.

I/O Management

As part of the system, there could be multiple devices that are connected and perform different input-output functions. These I/O devices could be used for human interaction such as display panel, touch panels, keyboard, mouse, and track pads, to name a few. Another form of I/O devices could be to connect the system to storage devices, sensors, and so on. There could also be I/O devices for networking needs that implement certain parts of the networking stack. These could be Wi-Fi, Ethernet, and Bluetooth devices and so on.

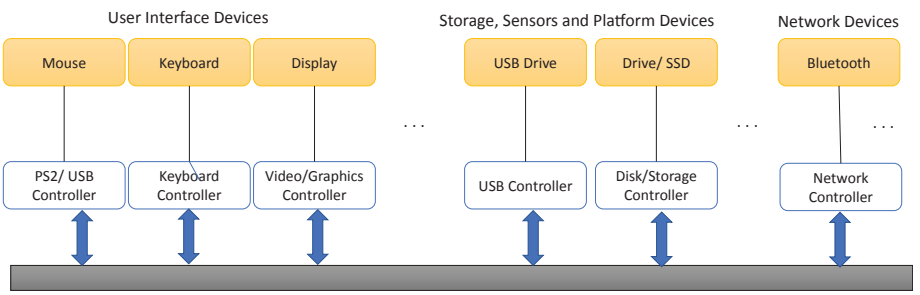

Figure 4-11. *Example I/O Controllers on a System*

As we can see in Figure 4-11, there are varied sets of I/O devices, and each of them has a specific purpose and programming interface. They vary from one to another in the form of protocols they use to communicate such as the data format, speed at which they operate, error reporting mechanisms, and many more. However, from an abstraction point of view, the OS presents a unified I/O system that abstracts the complexity from applications. The OS handles this by establishing protocols and interfaces with each I/O controller. However, the I/O subsystem usually forms the complex part of the operating system due to the dynamics and the wide variety of I/Os involved.

I/O Subsystem

Input/output devices that are connected to the computer are called peripheral devices. There could be additional lines that are used to connect to these devices for communication purposes. These are called buses that are a combination of "data lines" to transfer data, "control lines" to control a device, and "address lines" that may be used to specify address locations. There could be different buses or device protocols that an operating system may support. The most common protocols include Peripheral Component Interconnect Express (PCIe) protocol, Inter-Integrated Circuit (I2C), Advanced Configuration and Power Interface (ACPI), and so on. A device can be connected over one or more of these interfaces.

Consider the need to send a request to read the temperature of a specific device that is connected via ACPI. In this case, the operating system sends a request to the ACPI subsystem, targeting the device that handles the request and returns the data. This is then passed back to the application. In another example, we want to change the display brightness of the display device. In this case, a request is made from the application to the OS, which in turn detects the display device from the I/O subsystem

and requests the appropriate display brightness control setting. The display subsystem then makes the necessary action and returns the result, for example, success or failure, back to the OS. All of these happen in a seamless fashion so that the user is not aware of the intricacies involved. Typically, there is a software component in kernel mode called as the "device driver" that handles all interfaces with a device. It helps with communicating between the device and the OS and abstracts the device specifics. Similarly, there could be a driver at the bus level usually referred to as the bus driver. Most OSs include an inbox driver that implements the bus driver. As we saw in Figure 4-11, there is usually a driver for each controller and each device.

The I/O devices can be broadly divided into two categories called block and character devices. Usually, most devices would have a command and data location and a protocol that the device firmware and the driver understand. The driver would fill the required data and issue a command. The device firmware would respond back to the command and return an error code that is utilized by the driver. The protocol, size, and format could differ from one device to another.

Block Devices

These are devices with which the I/O device controller communicates by sending blocks of data. A block is referred to as a group of bytes that are referred together for Read/Write purposes. For example, when a request is made to write a file to the storage disk or if we need to transfer a file to a connected USB drive or if we need to read an image from a connected camera, the transfers are made as block reads. These could be defined by the device, for example, in multiple blocks of 512 or 1024 bytes. The device driver would access by specifying the size of Read/Writes.

Character Devices

Another class of devices are character devices that typically have a protocol defined using which the driver can communicate with the device. The subtle difference is that the communication happens by sending and receiving single characters, which is usually a byte or an octet. Many serial port devices like keyboards, some sensor devices, and microcontrollers follow this mechanism.

The protocols used by the different devices (block devices or character devices) could vary from one to another. There are three main categories of I/O protocols that are used.

Special Instruction I/O

There could be specific CPU instructions that are custom developed for communicating with and controlling the I/O devices. For example, there could be a CPU-specific protocol to communicate with the embedded controller. This may be needed for faster and efficient communication. However, such type of I/Os are special and smaller in number.

Memory-Mapped I/O

The most common form of I/O protocol is memory-mapped I/O (MMIO). As we discussed in the "Memory Management" section, the device and OS agree on a common address range carved out by the OS, and the I/O device makes reads and writes from/to this space to communicate to the OS.

OS components such as drivers will communicate using this interface to talk to the device. MMIO is also an effective mechanism for data transfer that can be implemented without using up precious CPU cycles. Hence, it is used to enable high-speed communication for network and graphics devices that require high data transfer rates due to the volume of data being passed.

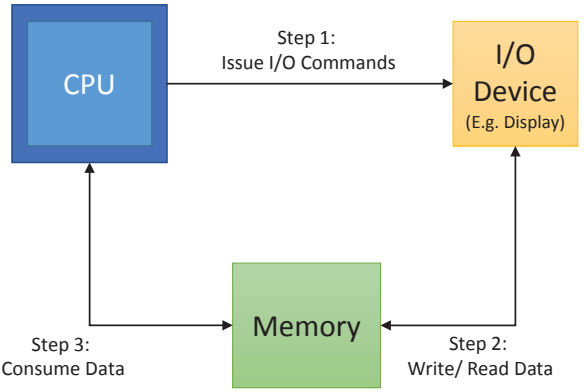

Figure 4-12. *Memory-Mapped I/O Flow in a Graphics Device Example*

Figure 4-12 depicts the case where the graphics driver acts as the I/O device and a memory-mapped location is used to share and communicate to the graphics device.

Direct Memory Access (DMA)

As we discussed earlier, there could be devices that run at a slower speed than supported by the CPU or the bus it is connected on. In this case, the device can leverage DMA. Here, the OS grants authority to another controller, usually referred to as the direct memory access controller, to interrupt the CPU after a specific data transfer is complete. The devices running at a smaller rate can communicate back to the DMA controller after completing its operation.

Most OSs also handle additional specific device classes, blocking and nonblocking I/Os, and other I/O controls. As a programmer, you could be interacting with devices that may perform caching (an intermediate layer that acts as a buffer to report data faster) and have different error reporting mechanisms, protocols, and so on.

Next, let's consider the difference between a polled and an interrupt-driven I/O.

Polled vs. Interrupt I/Os

Consider our temperature device discussed previously. If the device supports a polled I/O mechanism, the typical flow would involve requesting the device for temperature by issuing the command and filling the data field. At this point, the host system could wait for the operation to complete. In this case, it could be a blocked I/O call and a synchronous operation. However, it may not be efficient to block the execution. So, alternatively, the host system may issue a call and check the response at a later point in time if the operation has been completed. These could be implemented as a polled and an interrupt-driven I/O as shown in Figure 4-13.

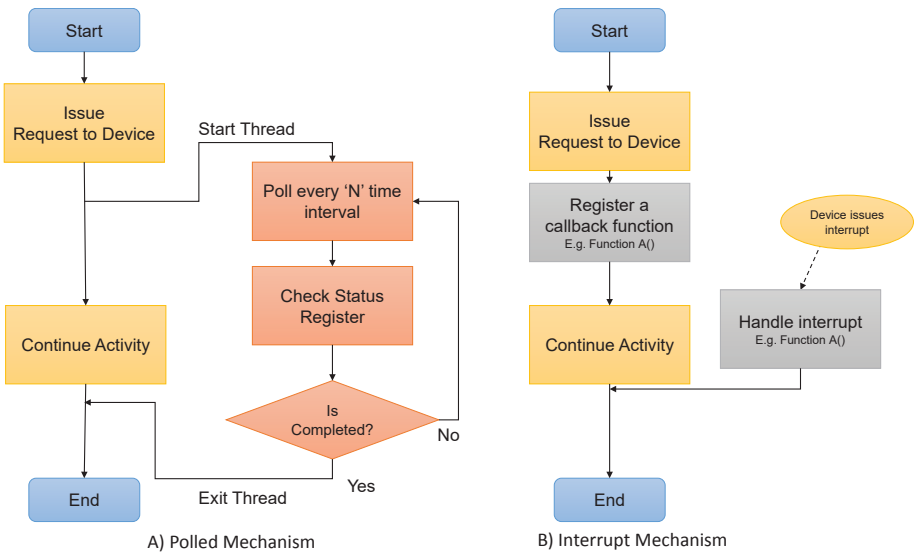

Figure 4-13. *Example Polled vs. Interrupt-Driven I/O Flow*

One mechanism would be for the host to poll the device and check the status of the operation. There is usually a status register that determines if the device has completed the operation. This is a common I/O flow for some devices as shown in Figure 4-13 (A).

Another mechanism would be to use the interrupt-driven mechanism. In this case, the request for operation is issued to the device. A callback function is also defined that needs to get called when the operation is completed. The device would continue and complete the operation and raise an interrupt once done as shown in Figure 4-13 (B). The callback function would be called appropriately to handle the interrupt. The callback function is also called as the ISR (Interrupt Service Routine), and as the name suggests, it services the interrupt. As a programmer, it is important to keep in mind that these ISRs are short-lived and lightweight and need to service the interrupt raised as quickly as possible.

I/O and Performance

The I/O subsystem plays a major factor in the overall performance of the system. As a software programmer, some of the operations done by your program could inadvertently impact the performance of the system. For example, a program could have multiple context switches arising due to the delays, responsiveness, and performance of the devices on the system. This may lead to an overall impact on the performance of your application. An application performing frequent writes to the disk or making many requests for continuous memory allocation can lead to excessive page swapping. A program could request for memory and may inadvertently not free up the memory requested after usage. These can cause memory leaks that may result in lower available memory and eventually impact the system performance. Also, requests for large blocks of contiguous memory may also have an impact since the memory subsystem may have to swap memory to accommodate the same.

A programmer would need to be cognizant of the I/O subsystem and its limitations in terms of performance expectations, limits/boundaries, and potential impacts. This is required since it may not only affect their application but could also affect the overall platform eventually.

Synchronization Concepts

Given there are devices and apps that must run together, access to hardware needs to be properly synchronized. There could be situations where more than one application may want to communicate to the same hardware device and the hardware device may not support concurrent access. It is important to know a few basics about how the OS uses synchronization to avoid potential conflicts. For this, let's start with the concepts of atomicity, critical sections, and locks.

Consider a multi-threaded application where a function is incrementing a global static variable:

```
count++; // count is a location in RAM
```

The preceding statement can be decomposed into three operations, which include fetching the value of count, incrementing the value of count in a local register, and then storing the updated value back to memory. However, as we saw earlier in this chapter, the thread that was executing this instruction could have been swapped in the middle of this operation. At the same time, there could be another thread that could be swapped in and may try to increment count. This is depicted in Figure 4-14 where Thread A was in the middle of incrementing while another thread tried to read the value of count. Ideally, Thread B should be able to access the count variable only after the operation in Thread A was completed.

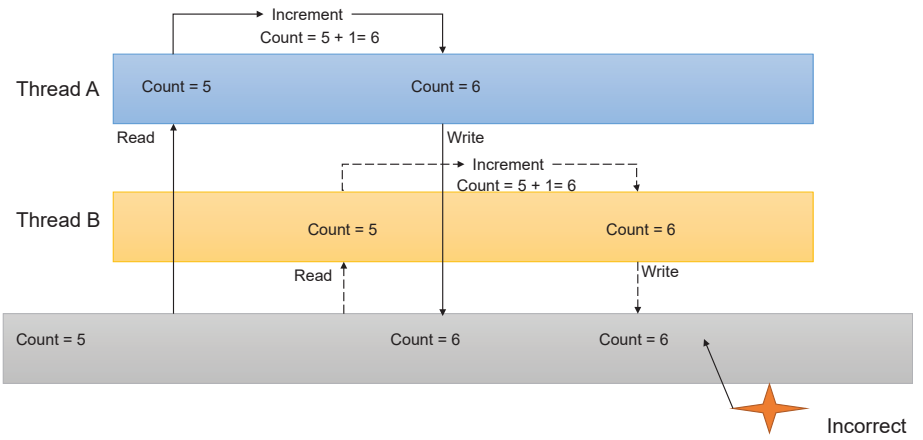

Figure 4-14. *Example of Increment Operation (count++) Across Threads*

If more than one thread tries to increment count at the same time, we may get unexpected results at any of the three steps we've described in the preceding. Such bugs are quite difficult to recreate and locate. This is an example where we need atomicity in instruction execution. Atomicity, as the name suggests, is a group of instructions that may need to be executed together as if they were a single instruction. The OS attempts to protect us from interrupting individual instructions while they are being executed.

Critical Sections

In multi-threaded applications, if one thread tries to change the value of shared data at the same time as another thread tries to read the value, there could be a race condition across threads. In this case, the result can be unpredictable. The access to such shared variables via shared memory, files, ports, and other I/O resources needs to be synchronized to protect it from being corrupted. In order to support this, the operating system provides mutexes and semaphores to coordinate access to these shared resources.

Mutex

A mutex is used for implementing mutual exclusion: either of the participating processes or threads can have the key (mutex) and proceed with their work. The other one would have to wait until the one holding the mutex finishes. As we can see in Figure 4-15, both Threads A and B would like to access a shared resource such as a file and write to it. Thread A initiates a request to acquire a lock before it can access the file. Once the lock is acquired, it finishes its operations on the file and then releases the lock. During this time, Thread B will not be able to access the file. Once completed, Thread B can follow the same procedure to access the shared resource.

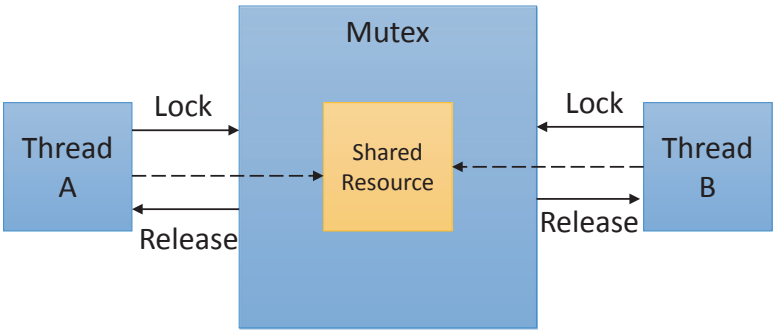

Figure 4-15. *Example Mutex*

A sample pseudo-code of the same implementation is shown in the following. As we can see, both threads try to acquire the lock before accessing the shared resource, that is, count in this case:

```
incrementCount()
{
    mutex_lock(&COUNT_MUTEX);
    count = count + 1;
    mutex_unlock(&COUNT_MUTEX);
}
```

Semaphore

A semaphore is a generalized mutex. A binary semaphore can assume a value of 0/1 and can be used to perform locks to certain critical sections. It is usually helpful to batch lock resource requests for better performance. As we can see in Figure 4-16, each of the threads A, B, C, and D requires access to the critical shared resource. When each of the threads requests to acquire the lock, the semaphore increments a counter and also maintains a waiting list of threads on the shared resource. Typically semaphores also expose two functions wait() and signal() that may be used to send notifications to threads appropriately.

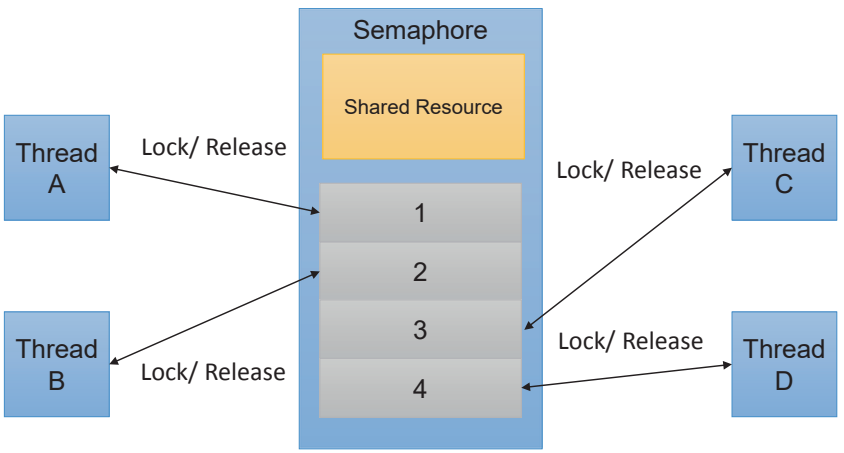

Figure 4-16. *Example Semaphore*

Now that we have seen how mutexes and semaphores work, we will go over another concept called deadlocks that may happen when the OS attempts to synchronize the operations on the system.

Deadlocks

In general, when we access a resource, we don't always know all the ways other parts of the system may also access that resource. The OS manages this resource access, but there could be certain situations where a set of processes become blocked because each process is holding a resource and waiting for another resource acquired by some other process. This is called as a deadlock. As we can see in Figure 4-17, Process A holds Resource 1 and requires Resource 2. However, Process B already is holding Resource 2, but requires Resource 1. Unless either of them releases their resource, neither of the processes may be able to move forward with the execution.

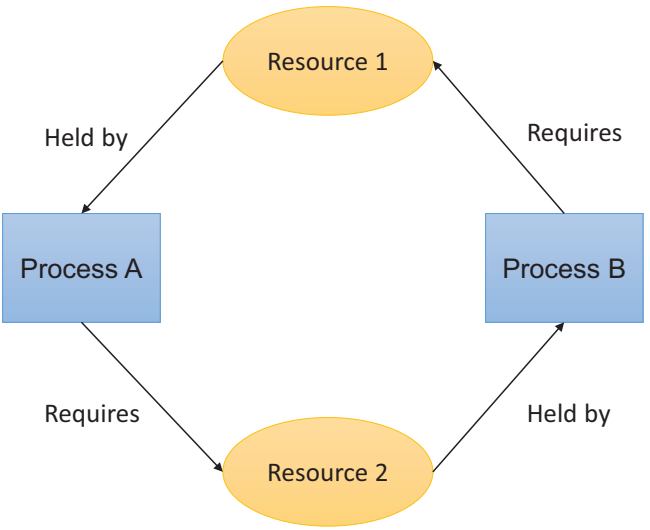

Figure 4-17. *Example of a Deadlock*

To elaborate from Figure 4-17, a deadlock can arise if the following four conditions hold:

- Mutual Exclusion: There is at least one resource on the system that is not shareable. This means that only one process can access this at any point in time. In the preceding example, Resources 1 and 2 can be accessed by only one process at any time.
- Hold and Wait: A process is holding at least one resource and is waiting for other resources to proceed with its action. In the preceding example, both Processes A and B are holding at least one resource.
- No Preemption: A resource cannot be forcefully taken from a process unless released automatically.
- Circular Wait: A set of processes are waiting for each other in circular form. As we can see in Figure 4-17, the arrows form a circular loop.

There are various mechanisms available to handle deadlocks using mutexes and semaphores that we discussed earlier along with additional algorithms to detect, avoid, and prevent deadlocks on the system. As a programmer, you would want to use these synchronization mechanisms.

To summarize, the I/O subsystem plays a critical role in the overall performance of the system. Memory management, interrupt responses, handling of I/O serializations, synchronizations, contentions, and so on play an important role in the overall performance of the system. Defining them, tuning and optimizing these are a major challenge for any operating system. There are various adaptive methodologies and runtime optimizations that various OS vendors invest in and try to adopt. These will continue to evolve for the better usage of our hardware.

File Systems

Applications often need to read and write files to achieve their goals. We leverage the OS to create, read, and write such files on the system. We depend on the OS to maintain and manage files on the system. OS file systems have two main components to facilitate file management:

1. Directory Service: There is a need to uniquely manage files in a structured manner, manage access, and provide Read-Write-Edit controls on the file system. This is taken care by a layer called as the **directory service**.
2. Storage Service: There is a need to communicate to the underlying hardware such as the disk. This is managed by a **storage service** that abstracts different types of storage devices on the system.

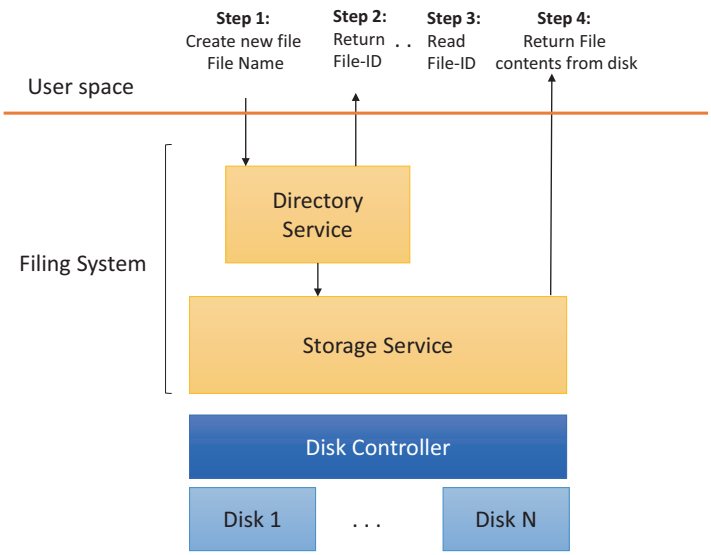

Figure 4-18. *File System Overview with File Access Process*

As shown in Figure 4-18, when a new file is created, the file name and path are passed to the directory service, which creates a unique file ID. This reference is used later to read contents back from the file using the storage service.

We will start with file concepts and then proceed to the functionality details.

File Concepts

From the perspective of the user, a file is a collection of related data that is stored together and can be accessed using a unique file ID usually referred as the file name. These files can be represented internally by different methods. For example, there could be .bin files in Windows, which only represent a sequence of bytes. There could be other structured contents with headers and specific sections in the file. For example, an EXE is also a file format in Windows with specific headers, a body, and controls in place. There are also many application-specific files, with their own formats. It is up to the programmer to define and identify if they require a custom file format for their application or if they can leverage a standard or common file format such as the JavaScript Object Notation (JSON) or the Extensible Markup Language (XML).

As a programmer, it may be important to know the attributes of the file before accessing it. The common attributes of any file include the location of the file, file extension, size, access controls, and some history of operations done on the file, to name a few. Some of these are part of the so-called file control block, which a user has access to via the OS. Most OSs expose APIs using which the programmer can access the details in the file control block. For the user, these are exposed on the graphical user interface via built-in tools shipped with the OS.

Directory Namespace

The operating system defines a logical ordering of different files on the system based on the usage and underlying storage services. One of the criteria most OSs adopt is to structure their directory service to locate files efficiently.

As shown in Figure 4-19, most OSs organize their files in a hierarchical form with files organized inside folders. Each folder in this case is a directory. This structure is called as the directory namespace. The directory service and namespace have additional capabilities such as searches by size, type, access levels, and so on. The directory namespaces can be multileveled and adaptive in modern OSs as we can see in the following folder structure with folders created inside another folder.

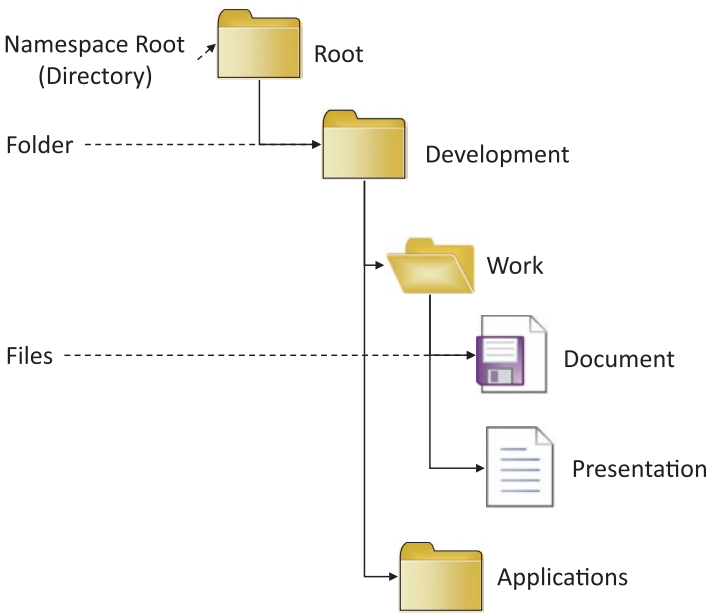

Figure 4-19. *Sample Directory Structure*

As a programmer, you should be aware of a few additional basic concepts from the file system point of view. We will discuss them in this section.

Access Control

There are different access levels that can be applied at file and directory levels. For example, we may not want a user-mode application running with a normal user credential to be able to make changes to some OS files/services. The OS provides different access control IDs and permissions to different users on the system. Also, each file may also have different levels of permissions to Read, Write, Modify, and so on. For example, there may be specific files that we may want anyone to be able to access and Read but not Write and Modify. The file system provides and manages the controls to all files when accessed at runtime. These may also be helpful when more than one user is using the same system.

Concurrency and Cleanup Control

There are many cases when the OS needs to ensure that a file is not moved or deleted when it is in use. For example, if a user is making changes to a file, the OS needs to ensure that the same file cannot be moved or deleted by another application or process. In this case, the OS would cause the attempt to move or delete the file to fail with an appropriate error code. As a programmer, it is appropriate to access a file with the required access level and mode (Read/Write). This also helps to be in line with the concurrency needs of the OS and guards against inconsistent updates.

The OS also needs to be able to periodically clear temporarily created files that may no longer be required for the functioning of the system. This is typically done using a garbage collector on the system. Many OSs mark unused files over a period of time and have additional settings that are exposed, which the user can set to clean up files from specified locations automatically.

Overall, the file system provides access, access controls, and protection mechanisms to files in the directory namespace. The programmer needs to be aware of the protections and have the right access controls (privileges) to interact with the file system successfully.

Access and Protection

If we have a system that is used by only one user without any access, networked or otherwise, to other systems, there may still not be assurance that the contents in the system are protected. There is still a need to protect the program resources from other applications. Also, there may be a need to protect critical devices on the system.

In practice, there is always a need to connect and share resources and data between systems. Hence, it is important to protect these resources accordingly. The OS provides APIs that help with access control and protection. Let's start with some of the concepts.

Rings: User Mode and Kernel Mode

We briefly covered user-mode and kernel-mode processes in the "Scheduling" section. One of the reasons the separation between user mode and kernel mode is implemented by most OSs is that it ensures different privilege levels are granted to programs, based on which mode they run in.

As shown in Figure 4-20, an abstract OS divides the program execution privileges into different rings. Internally, programs running in specific rings are associated with specific access levels and privileges. For example, applications and user-mode services running in Ring 3 would not be able to access the hardware directly. The drivers running on the Ring 0 level would have the highest privileges and access to the hardware on the system. In practice, most OSs only leverage two rings, which are Ring 0 and Ring 3.

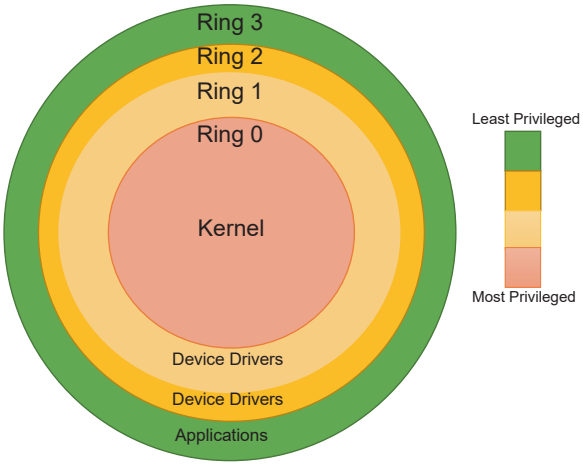

Figure 4-20. *Applications, Drivers, and Rings in an Operating System*

Virtualization

Consider the scenario where it may be required to have multiple closed environments that assume to have dedicated access to the resources on the platform. Operating systems and modern hardware provide a feature called virtualization that, you guessed it, virtualizes the hardware such that each calling environment believes it has the dedicated access it needs to function.

Virtualization is delivered via so-called virtual machines (VMs). A VM has its own **guest** OS, which may be the same as or different from the underlying **host** OS. A user can launch a VM, much like running any other program, and log into the guest OS. The host OS provides a hypervisor, which manages the access to the hardware. The guest OS is usually unaware of the internals and passes any resource/hardware requests to the host OS. The user can completely customize their VM and perform all their actions on this VM without affecting the host OS or any other VM on the system. At a high level, VMs help effectively utilize the hardware resources and are used heavily in server and cloud deployments.

Protection

There could be different security threats that may arise during the usage of a computer. These could attempt to access different critical resources on the platform such as data, compute, memory, and so on. The operating system needs to be able to detect any such attempts and potentially mitigate them. A threat could be any local or remote program that may be attempting to compromise the integrity of the resources in the system. To mitigate this, modern OSs usually implement checks to detect and protect against such incursions.

The most common protection would be to authorize the requester and apply authentication to any new request to the system. For example, when a request is made to a critical resource, the operating system would verify the user request (which is called as authentication) and their approved access levels (which is called authorization) and controls before providing access to a critical resource on the system. The OS may also have Access Control Lists (ACLs) that contain mapping of system resources to different permission levels. This is used internally before the OS grants permissions to any resource. Additionally, the OS may also provide services to encrypt and verify certificates that help with enhancing the security and protection of the system itself.

To summarize, the programmer needs to be aware of the various access controls and protection mechanisms in place and use the right protocols and OS services to successfully access resources on the system.

User Interface and Shell

Although the user interface (UI) is not part of the OS kernel itself, this is typically considered to be an integral part of the OS. That said, many OSs support different UIs, many of which are provided by third parties, for instance.

There can be multiple user interfaces for the OS all being implemented either as a text-based interface (e.g., MS-DOS) or a graphical-based interface (e.g., Microsoft Windows 10, macOS, etc.). The graphical user interface is the rich set of graphical front-end interfaces and functionalities provided by the OS for the user to interact with the computer. There could be an alternate simpler interface through a command line shell interface that most OSs also provide for communication. This is a text-based interface. It is common for programmers to use the shell interface instead of the GUI for quickly traversing through the file system and interacting with the OS. It requires the user to be aware of the commands and have the knowledge of the underlying OS implementations to be able to use it efficiently.

It is important for the software developer to be aware that the user interface and the shell interface may have an impact on their choice of programing language, handling of command line arguments, handling of the standard input-output pipes and interfacing with OS policies, and so on. Please note that the user interface and the features can be quite varied and different from each OS to another and are beyond the scope of this book.

Some OS Specifics

All OSs have features that may be unique to them. For example, UNIX has its own level of file abstraction and a hierarchical namespace. It handles heavyweight processes uniquely and supports pipes and signals for IPCs. Some of the recent UNIX enhancements provide additional capabilities and fixes across many of the IPC mechanisms.

Similarly, Windows NT has a layered architecture with Win32 APIs and a contained Windows Driver Framework (WDF) for driver development. Windows also has its unique way of handling plug and play (PnP) of devices on the system, power management, and I/O subsystem. Some of these may vary from one Windows version to the other as well.

From a programmer point of view, most of the basic concepts remain similar across these OSs. However, there could be few modifications and enhancements that you need to be aware of for your code to work across OSs. For example, the paths used to access files on the system or APIs referenced may be dependent on the OS/shell; and if you don't code for these situations, your code may not work as expected across OSs. You may want to keep these in mind at development. Further details are beyond the scope of this book.

Summary

In this chapter, we have described how the operating system forms an integral part of the system providing numerous capabilities including interaction with hardware and users and managing programs. OSs employ many design considerations and strategies based on which the OS abstracts and ensures seamless usage of the system.

As a software developer, you could be part of a larger ecosystem that could delve into device management, networking, web development, data management, and many other domains. The interfaces between the different domains and the way the operating system streamlines the operations between them are important for a software developer to comprehend and make meaningful decisions. Understanding these fundamentals helps in applying them at the various stages of software development ranging from architecture, design, deployment, and debug by taking the right choices.

References and Further Reading

- Arpaci-Dusseau, R. H.-D (2018). *Three Easy Pieces.* Arpaci-Dusseau Books. CITATION Rem18\l 1033. Arpaci-Dusseau, 2018
- Avi Silberschatz, P. B. (2012). *Operating System Concepts (Ninth Edition)*. John Wiley & Sons, Inc. CITATION Avi12\l 1033. Avi Silberschatz, 2012