CS 111

Contents

Part A.

Preface

This is a special part with Roman numeral numbering.

Preface: Legend

Operating Systems, Three Easy Pieces

Text in these boxes will indicate that further details can be found in the textbook (*Operating Systems: Three Easy Pieces* by Arpaci-Dusseau)

Definitions

Any definitions will be appear in a grey box like this one. There may be more than one definition per box if the topics are dependent on each other or are closely related.

Examples

Any examples will be appear in a blue box like this one. Examples will typically showcase a scenario that emphasizes the importance of a particular topic.

Abstractions

Layers of abstractions will appear in a violet box like this one. The key point of this box is to actively reinforce the concept of abstraction (and recognize how important it is in computer science!).

Preface: Introduction to Abstraction

Definition: Abstraction

Abstraction is the concept of providing a (relatively) simple interface to higher level programs, hiding unnecessary complexity.

The OS implements these abstract resources using physical resources, and thus is the source of one of the main dilemmas of OS design: What should you abstract?

Example: Network Neverland

Network cards consist of intricate technical details and specifications, but most users don't care about them. As a result, the operating system abstracts the technical aspects of a specific network card, such as the process of sending a message, for higher-level programs. Instead of manually performing each step to send a message using a particular network card, users can simply call the OS's `send()`[a] function and let the operating system handle the complex operations.

[a] The actual function name may vary.

Why Abstract?

Abstraction is utilized to simplify code development and comprehension for programmers and users. Furthermore, it naturally fosters a highly modular codebase as each abstraction introduces an additional layer of modularity. Moreover, by concealing complexity at each layer of abstraction, it encourages programmers to concentrate on the essential functionality of a component.

Due to variability of a machine's hardware and software, we can abstract the common functionality and make different types appear the same. This way, applications only need to interface with a common library.

Chapter 1

Overview

This section defines what an operating system is as well as gives motivating reasons as to why we should be studying them.

1.1 What is an Operating System?

Definition: Operating System

An **operating system (OS)** is system software that acts as an intermediary between hardware and higher level applications (e.g. higher level system software, user processes), acting as an intermediary between the two. It manages hardware and software resources and provides common services for user programs.

Abstraction: The Operating System

The operating system plays a crucial role in managing hardware resources for programs, ensuring controlled sharing, privacy, and overseeing their execution. Moreover, it provides a layer of abstraction that enhances software portability.

1.2 Why Study Them?

We study operating systems because we rely on the ***services*** they offer.

Definition: Services

In the context of operating systems, **services** are functionality that is provided for by the operating system. They can be accessed via the operating system's API in the form of system calls.

Moreover, a lot of hard problems that we run into at the application layer have (probably) already been solved in the context of operating systems !

Example: Difficult Downloads

Suppose you are developing a web browser and implementing a *download* feature. While downloading things one by one works fine, what if you need to download multiple items from different sites simultaneously? Thinking abstractly, we can see that this is a problem of coordinating concurrent activities, and fortunately, this problem has already been solved in the context of operating systems! Since you have already learned how to tackle this issue in operating systems, now you can apply the same solution to your *download* problem!

1.3 Key Topics (OS Wisdom)

When thinking of how to solve complex problems, these are some things you should take into consideration (to hopefully make your life a lot easier).

Objects and Operations

Think of a service as an object with a set of well-defined operations. Moreover, thinking of the underlying data structure(s) of an object may be useful in many situations.

Interface v. Implementation

Definition: Interface and Implementation

An **interface** defines the collection of functionalities offered by your software. It specifies the method names, signatures, and the *purpose* of each component.

An **implementation** refers to the actual code that provides the functionality described by the interface. It specifies *how* the interface's operations are executed and realized in practice.

We separate the two components to improve modularity and create robust, well-structured code. It allows for different compliant implementations (as long as they adhere to the agreed-upon interface specifications). This provides immense flexibility at the implementation level!

Example: Sort Swapping

Assume you are writing a library that contains a collection of common algorithms, one of them being `sort()`. Being the genius that you are, your implementation is as follows: Randomly reorder the elements until they're sorted. By some miracle, your library garners a lot of attention, but users are complaining that `sort()` takes too long. Not knowing what's wrong with your implementation, you take DSA[a] and learn that you've got shit for brains. You want to rewrite `sort()` but are worried that it might break the interface. However, you remember that interface $\neq$ implementation, so you rewrite `sort()` (using something like merge sort) pushing this new implementation into production, and bragging about how `sort()` now runs in $O(n \log n)$ time.

[a] DSA: Data Structures and Algorithms

Encapsulation

We want to abstract away complexity (when appropriate) into an interface for ease of use.

Policy and Mechanism

Definition: Policy and Mechanism

A **policy** is a high-level rule or guideline that governs the *behavior* of a system.

A **mechanism** is the implementation that is used to *enforce* the policy.

It is important to note that keeping policy and mechanism independent of one another is crucial. By separating policies from the underlying mechanisms, it becomes easier to change or modify policies without affecting the core functionality or technical implementation. This approach provides the ability to update policies independently from the underlying mechanisms, promoting modifiability, maintainability, and customization when designing software.

1.4 Why is the OS Special?

Definition: Standard and Privileged Instruction Set

The **standard instruction set** is the set of hardware instructions that can be executed by anybody.

The **privilaged instruction set** is the set of hardware instructions only the kernel can execute. When an application wants to execute a privilaged instruction, it must ask the kernel to execute it for them.

The OS is special for a number of reasons. Mainly, it has *complete* access to the privileged instruction set, *all* of memory and I/O, and mediates applications' access to hardware. This implies that the OS is *trusted* to always act in good faith. Thus, the OS stays up and running as long as the machine is still powered on (theoretically), and if the OS crashes, you're fucked lol.

1.5 Miscellaneous

Below are a list of miscellaneous topics.

1.5.1 Definitions

Definition: Instruction Set Architectures

An **instruction set architecture (ISA)** is the set of instructions supported by a computers. There are multiple (all incompatible) ISA's and they usually come in families.

ISA's usually come with privilaged and standard instruction sets.

Definition: Platform

A **platform** is the combination of hardware and software that provide an environment for running applications.

Common platforms include: Windows, [Mac, i]OS, Linux.

Definition: Binary Distribution Model

The **binary distribution model** is the paradigm of distributing software in compiled or machine code in the form of executables.

The binary distribution model is good for performance and security, but lacks in flexibility and is dependent on the platform you compile it for.

Definition: Portability

Portability refers to the ability to be adapted to different platforms with minimal modifications to the source code.

Portability is important if you're an OS designer because you want to maximize the number of people using your product $\implies$ your OS should run on many ISA's and make minimal assumptions about specific hardware.

Chapter 2

Resource Types

This section covers three types of OS resources: serially reusable, partitionable, and sharable.

Definition: Graceful Transition

A **graceful transitions** refers to the process of transferring control between two jobs such that there are no resource conflicts.

A graceful transition maintains system stability, data integrity and therefore cleanly releases resources. They typically ensure that users leave resources in a clean state; i.e. each subsequent user finds the resource in a "like new" condition.

2.1 Serially Reusable

Definition: Serially Reusable Resource

Serially reusable resources are resources that can be used by a single process at a time (sequentially) and are not designed to be shared in parallel.

These resources require access control mechanisms to ensure that only one process can access them at any given time. This control ensures a graceful transition between users and prevents conflicts or data corruption that may arise from concurrent access.

Example: Printing Process

Printers are a serially reusable resource: multiple job can use it but only a single job will be printed at a time.

2.2 Partitionable

Definition: Partitionable Resource

Partitionable resources can be divided up (or *partitioned*) into smaller, disjoint[a] segments.

[a] Disjoint: Independent of one another.

These resources require access control mechanisms to ensure that each segment is *contained*[1] and *private*[2]. Partitionable resources can be temporarily allocated (e.g. RAM, CPU *time slice*, etc.) or permanently allocated (e.g. Disk storage[3], database tables, etc.).

[1] Contained: Resources outside of a partition are not accessible.
[2] Private: External users cannot access the resources in your partition.
[3] Disk storage is permanently allocated until it isn't. You can use something like `fdisk` (in Linux) to modify partitions.

Example: Memory Mania and Disk Division

Memory can be partitioned, allowing multiple unique processes to access their own allocated memory space independently.

Disk storage can be partitioned into separate logical volumes! This is commonly used to dual-boot different operating systems. Recently, M1(/2*ish*) Apple products can now run Linux on bare metal (still in beta !) via Asahi Linux!

Graceful transitions are still necessary in partitionable resources! Partitionable resources that aren't permanently allocated need to clean up after themselves.

2.3 Shareable

Definition: Shareable Resource

Shareable resources are usable by multiple *concurrent* clients. They need not "wait" for access nor do they "own" a particular subset of a given shared resource.

These resources require access control mechanisms to ensure that the shareable resource is used in a controlled and secured manner.

Example: Cloud Crazy

The cloud (e.g. Google Drive, Oracle Cloud, etc.) is a powerful shared resource! It enables multiple concurrent users to access shared folders and files, facilitating simultaneous editing and collaboration.

Graceful transitions typically are not necessary since a shareable resource generally doesn't change state *or* doesn't require any clean up. In the example above, while the cloud files change state, there is no cleaning up to do, so a graceful transition is not necessary (what's clean doesn't need cleaning).

Chapter 3

Services

The OS provides services in a multitude of ways. This section will introduce and explain how services are provided throughout the software stack.

3.1 Subroutines

Definition: Subroutine

A **subroutine** in the context of operating systems is a small, self-contained and reusable portion of code that performs a specific function.

Subroutines are usually called directly, and operate like normal code (stack manipulation), and are typically seen in higher layers of the OS stack. They are fast, but usually cannot use the privileged instruction set. Furthermore, they are usually not language agnostic. The most common way to implement these subroutines are libraries!

3.2 Libraries

One way the OS provides services to users is via libraries. Standard utility functions such as `malloc` can be found in libraries (in this case, `stdlib.h`). So what exactly is a library?

Definition: Library

A **library** is a collection of code modules that encapsulate common operations, algorithms, and functionality.

Abstraction: Libraries

Most systems are equipped with a wide range of standard libraries, which are designed to be reused. These libraries encapsulate complexity and provide an additional layer of abstraction, simplifying problem-solving.

Example: DSA Doozy

In DSA, you likely had to implement different types of data structures (linear, hierarchical, graphical, etc.) and algorithms (search, divide/conquer, dynamic programming). Imagine your surprise when you find out most of these data structures and algorithms have already been written (and probably perform better than your implementation, no offense).

3.2.1 Bind Time

The choice of library bind time depends on multiple factors that include (but are not limited to): performance requirements, deployment and distribution considerations, and dependency management.

Definition: Static

Static libraries are precompiled code modules that link directly to the executable at compile time. They allow for efficient standalone executables, but they result in larger file sizes and require recompilation if the library is updated.

One easy example of a static library is `libc`, or the `C` standard library! It encapsulates a myriad of commonly used operations, algorithms, and functionality when programming in `C` like everyone's favorite `malloc()`.

Definition: Shared/Dynamic

Shared/Dynamically Linked libraries are separate files from the load modules that can be loaded and shared by multiple concurrent processes. They are loaded and linked to the executable during runtime.

One thing to note about shared libraries is that they cannot define or include global data storage. Moreover, called routines must be known at compile time since the fetching of the code is the only thing that is delayed until runtime.

3.3 System Calls

Definition: System Call

A **system call** is when users request functionalities from the operating system that are a part of the privileged instruction set.

System calls allow for the use of the privileged instruction set and interprocess communication, so why don't we just make everything a system call? System calls are *slow*[1]. Thus, we typically like to reserve system calls for operations that *require* the privileged instruction set.

Example: System Call Stress

System calls sound like a big deal, but you have already been introduced to them! Some include:

(i) File I/O: Reading from or writing to files on a disk require system calls since they require privileged instructions.

(ii) Memory management: Though functions like `malloc()` itself isn't a system call, it is implemented by calling system calls!

(iii) Process management: Only the kernel can *directly* create/destroy processes and ensure process privacy and containment.

(iv) Interprocess communication: Mainly for security reasons, communication between processes require privileged instructions.

Most of the time, everything related to a given system call is dealt with in the kernel. However, there are instances when the kernel will outsource tasks via direct calls to *untrusted*[2] code, waiting for a response, then returning to the calling process.

[1] System calls are between 100 and 1,000 times slower than standard subroutine calls!
[2] Untrusted: Not guaranteed to be secure.

3.3.1 Trusted Code

Not all trusted code must be inside the kernel! If code doesn't *need* to access kernel data structures or execute privileged instructions, they might sit outside of the kernel layer.

Definition: Trust

Trusted code is guaranteed to be secure and will perform correctly. That is, it is safe for the operating system to run it.

Example: Lazy Login

A login manager/application is a great example of a program that is trusted but doesn't sit inside the kernel. When you login, the kernel will outsource the job to the login application!

3.4 Messages

Another way of service delivery is via messages that are exchanged with a server via system calls.

Advantages

Messages allow users to send and receive requests from anywhere! They are also highly scalable and can be implemeneted in user-mode code.

Disadvantages

Messages are *slow*[a] and are limited to operate on process resources.

[a] Messages are between 1,000 and 100,000 times slower than subroutine calls!

3.5 Middleware

Middleware refers to software components that are essential for a particular application or service platform but do not sit inside the OS. They bridge the gap between the application and underlying OS, providing additional functionalities and services.

Example: Middleware Madness

Some examples of middleware include database systems, web servers (like Apache and Nginx), distributed computing platforms (like Hadoop and Zookeeper), and cloud computing platforms like OpenStack.

We prefer middleware over implementing such functionalities directly in the kernel because kernel code is expensive and risky since kernel-level issues can impact the stability of the entire system! Instead, middlware is typically developed in user mode, making it easier to build, test, and debug[3]. Moreover, it is more portable, meaning it can be used across different operating systems without modification!

[3] If the middleware crashes, it can be restarted independently of the entire system, minimizing its impact on the overall OS stability.

Chapter 4

Interfaces

How do processes communicate with the OS? Interfaces! There are two main types: API and ABI.

Abstraction: Interfaces

Interfacess introduce a layer of abstraction, allowing programmers to focus on *what* they need without worrying about *how* it's implemented!

4.1 Application Programming Interface

Definition: Application Programming Interface

The **application programming interface (API)** provides a standardized set of rules and policies (at the source code level) that govern how different software components can communicate with each other.

API's are the basis for software portability, allowing a program to be compiled for a particular architecture or OS. That is, programmers can recompile for different targets without changing the source code, provided that the API is consistent. To do this, we link our program with OS-specific libraries that implement the funcitonality specified by the API.

Thus, when the program is compiled and linked using an API-compliant system, the binary executable will be compatible[1] with any other API-compliant system! Well-defined API's allow us to create interoperable applications and libraries that are platform/system independent, promoting software portability, reusability, and simplicity when building complex systems!

4.2 Application Binary Interface

Definition: Application Binary Interface

The **application binary interface (ABI)** defines the low-level binary interface that allows compiled programs to interact with the underlying system and hardware.

ABI's govern how DLL's are structured and how they interact with programs at a binary level. But how? ABI's define how data is represented in memory and passed between functions and across different modules! This includes details like register usage, linkage conventions, parameter passing conventions, and stack layout. They also connect the API to the specific hardware, translating source-level instructions to actual machine level instructions.

Once a program is compiled using an ABI-compliant system, it can run unmodified (i.e. without recompilation) on any other system with the same ABI! This ensure that a single binary can service all ABI-compliant systems. Hence, they are usually intended for end-users and software deployment.

[1] An API-compliant program will compile and run on any system that supports the same API.

4.3 Corollary: Libraries

Libraries are accessed through API's! The API provides source-level definitions for how to access the library, and is readily portable between systems. While DLL's are also accessed via an API, the loading mechanism is specified by the ABI.

4.4 Best Practices for Interoperability

Standalone programs are useless! All useful programs use system calls, library routines, operate on external files, exchange messages, etc. That is, they all utilize OS services! Thus, if the interface changes, these programs will fail. Thus, API requirements are frozen (finalized) at compile time. This means:

(i) Execution platforms must support the interfaces.

(ii) All partners/services must support the interface protocols.

(iii) The API must be backwards compatible

That is, API's need to be *stable* in order to support interoperability! API's need to be rigorously specified and standardized[2]. Thus, the developers that use the API are encouraged to be compliant with the API specifications to ensure that their program will survive updates.

Example: New Version New Problems

Suppose you are writing an application for an OS running on version 0.6.8, and being the genius programmer you are, find an exploit that goes around the OS API to make your app run faster. Everything runs fine until version 0.6.9. Confused, you find that the developers hate you in particular and decided to patch the exploit you used. Having learned your lesson, you have to rewrite that entire feature, being compliant with the API.

Interoperability requires both parties to honor their side of the contract: Standard bodies must keep the interface stable, while developers must be compliant with the API if they want their products to survive new updates.

4.5 Aside: Side Effects

Definition: Side Effect

A **side effect** is occurs when an action on one object has *non-trivial*[a] consequences.

[a]Non-trivial: Effects that are not included in the interface specifications.

Side effects are inevitable, but are not the end of the world! They usually happen due to shared state between independent modules and functions. Thus, developers should ignore side effects and continue being compliant with the interface as they *should* be patched. In general, try to avoid exploiting side effects since they are not guaranteed to be there or maintained!

[2]Standard body: Most big projects (like Linux) have standard bodies that manage the interface definitions so as to maintain stability!

Chapter 5

Abstraction

Recall that we like abstractions in Computer Science. Life is easy for high level programmers if they work with a simple abstraction. The OS is responsible for creating managing, and exporting such abstractions.

Example: Hardware Hiding

Hardware is fast, but complex and limited, so using it *correctly* can be extremely challenging. Thus, hardware is commonly seen as a building block rather than a solution. We provide abstractions to encapsulate implementation details (like error handling and performance optimization) and eliminate behavior that is irrelevant to the user. Thus, abstractions make it more convenient to work with the hardware!

The OS provides some core abstractions that our computational model relies on: memory, processor, and communication abstractions.

5.1 Memory Abstractions

Memory abstractions provide a consistent way to interact with various data storage resources, simplifying the process for users. However, there are some complicating factors that come with abstracting memory.

5.1.1 Complications

The operating system managing these abstractions doesn't have abstract devices having arbitrary properties. Instead, it handles physical devices that may have inconvenient properties. Therefore, the primary goal of OS abstraction is to create an abstract device with desirable properties derived from the physical device that lacks them.

Memory Lifetime

Definition: Persistent and Transient Memory

Persistent memory retains data even when power is turned off (long term memory). Examples of non-volatile storage include hard drives and solid state drives.

Transient memory loses its data when power is turned off (short term memory). An example of volatile storage is RAM.

Managing data in both types of memory presents different challenges and considerations when designing a complex system.

Example: File Finding

When you run a program, all of the variables local to the program are stored in transient memory, or RAM. However, suppose you write to a file `foo` in your program. `foo` is stored in persistent memory! This means that weeks later, if another program wants to access foo (e.g. `cat foo`), the contents you wrote into `foo` will still be there.

Size

There can be a discrepancy between the size of memory operations that the user wants to work with and the size that the physical memory can handle. Thus, being able to manage data manipulation efficiently, especially when data sizes differ, is very important!

Example: Caught in 4k

Hardware usually processes data at the word level[a]. However, when writing a block of data to flash memory, we typically move data in 4k chunks. Therefore, when reading from or writing to memory, we must ensure that data is moved in the appropriate word size.

[a] Word sizes are typically 32 or 64 bits depending on the architecture

Latency

Definition: Persistent and Transient Memory

Latency (in the context of memory) refers to the time it takes for a process to read from memory.

Note that the latency reading from RAM and from disk are two very different times which lead to varying performance gains depending on which one you optimize for.

Implementation Variety

The same memory abstraction might be implemented using various physical devices. This leads to varying performance depending on which device imlements the abstraction.

Example: Caught in 4k

Storage devices like hard disks, solid-state drives, and optical drives can all be used to implement file storage, but the performance differences vary between the three. (SSD > HD > optical).

5.2 Interpreters

Definition: Persistent and Transient Memory

An **interpreter** is the module (abstract or physical) that executes commands and "gets things done".

Abstraction: Interpreters

At the physical level, we have the CPU. Directly working with the CPU is not easy, so the OS provides a higher level intepreter!

An interpreter has several basic components:

(i) Instruction Reference: Tells the interpreter which instruction to execute next.

(ii) Repertoire: The set of features that the interpreter supports.

(iii) Environment Reference: Describes the current state on which the next instruction should be executed.

(iv) Interrupts: Situations in which the instruction reference pointer is overridden.

Example: Processing Processes

The process that we interface with is an example of an interpreter. The OS maintains the *instruction reference* (a program counter for a given process). Its source code specifies its *repertoire*, and its stack, heap, and register contents are its *environment*. The OS manages all three of these components. Another thing to note is that no other interpreters should be able to access the process' resources; i.e. the interpreter should be private.

Aside: Implementation

Implementing the process abstraction in the OS is relatively straightforward when dealing with only one process, but in reality, that is seldom the case. When dealing with multiple processes, we have to consider that:

(i) The OS has limited physical memory to hold environment information.

(ii) There are usually only one set of registers (or one per core).

(iii) The process shares the CPU (or core) with other processes!

To address these issues, we need:

(i) A *scheduler* to share the CPU among multiple processes.

(ii) Better *memory management* hardware and software to create the illusion that each process has full access to RAM (when in reality, they don't).

(iii) Access control mechanisms for other memory abstractions to keep our machine secure.

5.3 Communications

Definition: Communication Link

A **communication link** allows interpreters to talk to each other (on the same or different machines).

Abstraction: Communication Links

A communication link at the physical level consists of memory and cables. However, at more abstract levels, we have networks and interprocess communication mechanisms.

Communication links are distinct from memory abstractions for a couple of reasons:

(i) Factors such as network congestion, distance, and hardware limits contribute to variations in speed, latency, and bandwidth. On the other hand, memory access offers more predictability and consistent performance.

(ii) Communication links are often asynchronous[1], introducing additional complexity compared to synchronous memory access.

[1] Asynchronous: Data can be sent and received independently of each other. Timing of communication is not guaranteed to be coordinated.

(iii) The receiver in communication links may be reactive, meaning they only perform an operation because the sender initiated it. In contrast, data is usually immediately available when requested via memory access.

Aside: Implementation

If both ends of the communication link are on the same machine, it's trivial: use memory for transferring data! Copy the message from the sender's memory into the receiver's memory *or* transfer *control*[2] of the memory containing the message from the sender to the receiver. To implement communication links across machines, we have to consider the following:

(i) We need to optimize the cost of copying data.

(ii) Memory management can become very tricky (especially when manipulating ownership!).

(iii) We need to include complex network protocols into the OS itself. This raises new security concerns that the OS might need to address.

(iv) We need to be able to deal with message loss, retransmission, etc.

5.4 Generalizing Abstractions: Introduction to Federation Frameworks

Rather than applications dealing with varied resources, we can make many different things *appear*[3] the same by using a unifying model! Usually, these unifying models involve a federation framework.

Example: Computer Communism

A Portable Document Format, or PDF, is the unifying model for printed output. If we want to print something to a printer, as long as the document is in the PDF format, it will know how to print it!

SCSI, SATA, and SAS are standard ways to interface with hard disks and other storage devices (CD, SSD, etc.).

Definition: Federation Framework

A **federation framework** is a structural design that enables similar (but different) entities to be treated uniformly by creating a single *interface*[a] that all entities must adhere to.

[a]The *implementation* that supports the interface is specified by the particular entities.

Note that a unifying model need not be the model with the "lowest common denominator"[4]. Rather, the model can include "optional features" which are implemented in a standard way. Why? Some devices may have features that others of the same class do not. Thus, these "optional features" allow us to create a highly modular federation framework while maintaining a common unifying model.

Example: Pretty Printing

Suppose you are building a federation framework for printers. Some printers can only print single-sided while others can print double-sided. So, a possible federation framework could require that all devices that classify as printers *must* be able to print *at least* single-sided, with the *optional feature* of double-sided printing. Extending this idea, we can add more optional features like color printing, DPI settings, etc.

[2]Transferring control: Change who owns the memory segment!
[3]We want to *abstract* away the implementation details!
[4]Lowest common denominator: The set of features *all* entities have in common.

Unfortunately, there may be instances where a particular device may have features that cannot be exploited through a common model. This is the tradeoff we make for a uniform model. There have been arguments both for and against being able to handle such features in a federation framework.

5.5 Layering Abstractions

It is very common practice to create increasingly complex services by *layering* abstractions.

Example: Abstract Abstract Abstract!

A generic file system is an abstraction layer over a particular file system (1). A particular file system layers on top of an abstract disk (2). This abstract disk layers on a real disk (3). Here, a generic file system is implemented with 3 layers of abstraction! This hierarchical structure simplifies development and enhances system scalability.

Layering allows for modularity, easy development of multiple services on multiple layers, and flexibility in supporting various underlying services. Abstractions hide complex implementation details, promoting structured and independent design.

Unfortunately, layers in a system often introduce performance penalties due to the additional indirection they bring. Moving between layers can be costly as it typically involves changing data structures and representations, and may require extra instructions. Moreover, layers may not be entirely independent of one another; for example, lower layers can impose limitations on what upper layers can achieve.

Example: Packages Play Hide n Seek

An abstract network link may hide causes of packet loss, since the lower layer that this abstraction is built off of may hide certain implementation details that are relevant to these issues.

The OS offers numerous abstractions, catering to diverse needs and use cases. Selecting the most suitable abstractions is crucial for achieving optimal results. It involves understanding the trade-offs between higher-level and lower-level abstractions to ensure efficient utilization of system resources and effective application development.

Part I

Virtualization

Chapter 6

The Process

Definition: Process and State

A **process** is a type of interpreter that executes an instance of a *program*[a].

[a] A **program** is a set of instructions that defines a particular application.

A **state** is a mode or condition of being, representable by a set of bits.

When you begin executing a program, it becomes a process. There may be multiple instances of the same program running simultaneously on the same computer. Typically in these cases, each running instance is a separate process.

Abstraction: Processes

Processes are a type of interpreter, an abstraction we covered in the previous section. We can think of it as a virtual private computer.

A process is an *object*[1], characterized by its state and its operations. All persistent objects have state, distinguishing them from other objects and characterizing the object's current condition. OS's objects' state is mostly managed by the OS itself and not by the user code. Thus, we must ask the OS to access or alter the state of an OS object.

Example: Priority Process

The OS maintains information about the current priority of each process in the system. This subset of *state* determines its position in the scheduling queue.

6.1 Process Address Space

Definition: Process Address Space

The **process address space** is the set of addresses visible to the process. This address space is *private*[a] to the process.

[a] Private: Inaccessible by outside processes.

The process' address space consists of all memory locations accessible by the process. Invalid addresses are those outside its address space, and as such, the process cannot request access to them. Modern operating systems give the illusion that *every* process' address space can (but often don't) include *all* of memory.

[1] Object: NOT the OOP object.

6.1.1 Layout

The process address space typically consists of different segments:

(i) Shared Code: Contains the executable code of the program. We do not do self-modifying code in modern computer systems. Thus, this shared code is static while the process is running, meaning they are read/executable only. So, subsequent instances of a program will be accessing the one stored in RAM since it's a *shared resource*.

(ii) Shared Libraries: Like shared code, shared libraries are a *shared resource* that are stored somewhere in RAM. Thus, multiple processes can access these shared libraries concurrently.

(iii) Private Data: Stores global and static variables used by the program. Since private data is read/write, they are *not* a shared resource, and thus is private to a particular instance of a process.

(iv) Private Stack: Like private data, the private stack is private to a particular instance of a process.

All of these must sit somewhere in RAM, but different type of memory elements have different requirements (e.g. *shared* coded is read/execute, *private* stack is read/write).

Example: Linux Layout

Each operating system puts these process memory segments in different places, but here's how Linux does it! Code segments are statically sized and are put at the beginning (e.g. 0x00000000). The data segment (and heap) is placed after the code segment and grows upward. The stack is placed at the very end (e.g. 0xFFFFFFFF) and grows downward. It is crucial that the data segment and stack are **not** allowed to meet!

6.1.2 Code Segment

Definition: Load Module

The **load module** is the output of a linkage editor, where all external references have been resolved and object modules have been combined into a single executable.

The process starts by creating a load module. To make instructions executable, we need to load the code into RAM, as we can't directly run instructions from the disk. Next, we read the code from the load module and copy it into a specific *code segment*[2] within the process's address space. This allows the CPU to fetch and execute instructions from the memory while the program runs. Since the code is static (read/execute only), we can share it among multiple processes, which helps reduce unnecessary duplication of code.

6.1.3 Data Segment

Data also needs to be initialized within the address space of a process. This requires the creation and mapping of a process data segment into the process' address space. The initial contents of the data segment are copied from the load module. In particular, the BSS[3] segments are initialized to all zeroes. Data segments are read/write (and thus private to the instance of the process). The program can grow or shrink this segment (via the `sbrk` syscall).

6.1.4 Stack Segment

Definition: Stack Frame

A **stack frame** serves as storage for procedure local variables, invocation parameters, and save/restore registers.

[2]Code segment: A segment we establish in the process' address space to accommodate the code.
[3]BSS: Block Started by Symbol

Modern programming languages are stack-based. Thus, each procedure call allocates a new stack frame, and once it is completed, the corresponding stack frame is popped off of the stack, freeing any memory that was allocated for it. Modern CPU's have built-in stack support. Thus, the stack must be preserved as part of the process state to ensure proper execution and continuity during a process' lifetime.

The size of the stack in a program depends on its activities, such as the amount of local storage used by each routine. It grows larger as calls nest more deeply (since each procedure call allocates a new stack frame!), and once these calls return, their stack frames can be recycled for future use.

Corollary: Who Manages The Stack?

The operating system is responsible for managing the process' stack segment! It is created alongside the data segment when the program is loaded into memory.

Different operating systems implement stack management differently Some allocate a fixed-size stack at the program's load time, while others dynamically extend the stack as the program needs more space.

Across all operating systems, stack segments are usually only read/write for security reasons. This prevents any unintended execution of code in the stack (e.g. buffer overflows) and ensures that it is used exclusively for storing data and variables.

Stack segments are process private, meaning each process has its own unique stack. This isolation ensures that processes cannot interfere with each other's stacks, which is crucial for maintaining system stability and security.

6.1.5 Libraries

Static libraries are added to the load module, so each module includes a copy of the required library code. This means that if a program is linked with a static library, the compiled code of that library is integrated into the load module. So, if there are multiple processes using the same static library, we have code duplication. Another downside is that programs must be re-linked to get newer versions of static libraries.

Shared libraries are loaded into memory once they are required, and are separate from load modules. Thus, once they are in memory, other processes need not reload the library to access it, reducing memory consumption. Since shared libraries are separate files from load modules, the operating system handles loading the required libraries into memory when the program is executed. In contrast to static libraries, shared libraries are easier to upgrade.

6.2 Process Data Structures

Definition: Register

A **register** is a small, fast, storage location within the CPU used to store data temporarily during the execution of a program. They are *much* faster to access than memory since registers are on the CPU.

General registers are used for temporary data storage and arithmetic tasks, while the program counter keeps track of the memory address of the next instruction to be executed. The processor status register contains important CPU status information, and the stack/frame pointers are essential for managing the function calls and local variables.

Each process has its own set of OS resources (e.g. open files, CWD[4], locks for synchronization). Additionally, the OS maintains specific state information for each process, including the process ID (PID), priority, and execution state.

[4]CWD: Current Working Directory

6.2.1 Process Descriptors

Definition: Process Descriptor

A **process descriptor** stores all information relevant to the process for process management, scheduling, and resource allocation.

Process descriptors typically include information such as the state to restore to when a process is dispatched, references to allocated resources, and information to support process operations. They are managed by the OS and are also used for security decisions and allocation issues.

Process Control Block

Definition: Process Control Block

The **Process Control Block (PCB)** is the data structure that Unix systems use and is a type of process descriptor. It is used to represent and maintain information about an individual process. It contains various details and state information needed for process management, scheduling, and resource allocation.

A PCB keeps track of a process' unique PID, state, address space information, program counter, priority, and more.

Example: What am I Holding?

Let's look at what the PCB keeps track of in more detail.

(i) The unique PID distinguishes processes from each other.

(ii) The process state indicates whether a process is running, ready to run, waiting for I/O, or stopped.

(iii) Address Space information is stored in the PCB, keeping track of a process' memory layout (virtual memory address space, code/data/stack, and heap segments).

(iv) The PCB stores the CPU context when the process is interrupted or preempted, including the values of CPU registers, program counter, and other relevant CPU state.

(v) The PCB keeps track of the Parent PID (PPID), so you always know where you came from.

(vi) The File Descriptor Table (FD Table) contains the table that maps file descriptors opened by the process, indicating which files the process has access to.

(vii) Signal handlers are used for interprocess communication.

6.3 Other Process State

Not all process state is stored directly in the process descriptor! Other process state is stored in several other places:

(i) Application Execution State: The actual execution state of the application, including variables, function signature, and local data, is stored in the process' stack and CPU registers. As the program executes, data is pushed/popped from the stack, and registers are used to hold intermediate results during computation.

(ii) Supervisor-Mode Stack: Linux processes have a separate supervisor-mode stack, used to retain the state of in-progress system calls and to save the state of a process that gets interrupted by

an interrupt or preemption. This ensures that critical syscall information is preserved across privilege level changes or when the process is preempted by higher-priority tasks.

(iii) Other Memory Areas: Additional process state like the heap, shared memory segments, and memory-mapped files, may be stored in various other memory locations outside of the process descriptor.

Chapter 7

Process Handling

We will cover creating, destroying, and running processes in this section.

7.1 Creating a Process

Process can be created in two main ways: by the operating system and by the request of other processes.

The operating system is responsible for creating processes during system boot or when a user initiates the execution of a specific program. When a user starts an application or runs a command, the OS *initializes*[1] the state of a new process for that program.

Processes can also be created by other running processes! When a process wants to create a new process, we use syscalls like `fork` (in Linux) or `CreateProcess` (in Windows). The parent process is the process that initiates the creation of another process by requesting the OS to create a child process with a specific program and initial state. As a result, the child process inherits some characteristics from its parent, like file descriptors and environment variables.

Other than that, child processes are independent of the parent process and execute concurrently with their parent. They typically start executing from the same point as the parent process but can have distinct memory spaces and execution paths.

Example: Pipe Up!

In our pipe lab, we utilized `fork` and `execlp` to mimic the pipe operator (|)! This consisted of spawning child processes from parent processes to execute different commands. In this instance, we created child processes from another process (the parent) using `fork`, and ran a completely distinct program using `execlp`!

7.1.1 The Process Table

Definition: Process Table

A **process table** is a data structure used by the OS to organize and keep track of all currently active processes in the system. Each entry corresponds to a process and contains a pointer to its PCB.

When a new process is created, the operating system generates a new PCB, which serves as the basic per-process data structure. But how do we keep track of these PCB's? Once a PCB is created, the OS will typically place it into a process table. This table allows the OS to efficiently manage and access information about all running processes (tracking state, scheduling, allocating system resources, etc.).

[1] Initialize: The OS will allocate memory, setting up the program's initial execution context, and creating a PCB to manage the process.

7.1.2 Creating the Address Space

In addition to a PCB, we also need an address space to hold all the segments it requires for execution! The OS is responsible for creating and managing the address space for each process. Once the address space is created, what does a new process need in its address space?

The OS needs to allocate memory within the address space for various segments (see **6.1 Process Address Space**) and load the program code and data into these segments. After, the OS sets up the initial values of essential registers for the process:

(i) Program Counter will point tot he first instruction to be executed.

(ii) Processor Status will configure CPU flags.

(iii) Stack Pointer will point to the stop of the initial stack frame.

Once these steps are completed, the new process is ready for execution!

7.1.3 Choices for Process Creation

When we create new processes, there are two approaches commonly used in operating systems: starting from a "blank" process and starting from a template[2].

Blank Process

In this approach, a new process is created with no specific state or resources. It is a blank slate with minimal to no predefined attributes. The OS then provides a mechanism to fill in the essential details required for the process to run successfully (like the code, program counter, etc.). This approach is usually used in Windows-based systems (via the `CreateProcess` syscall).

Template Process

In this approach, a new process is created by duplicating the parent/calling process. This new process inherits most of its attributes and resources from the parent (like the code, PC value, open file descriptors, etc.). The child process will then start execution from the same point in the code as the parent. This approach is usually used in Unix-based systems (via the `fork` syscall).

Example: Forking Forking

When forking a process, we need to figure out which process is which! Luckily, the guys designing this were smart enough to write code that does this for us! When we call `fork()`, the return value will tell us if the current process is the child or parent (or if the fork failed). The process is a child, parent, or failure if the return value of `fork()` $== 0$, > 0, or < 0 respectively.

7.1.4 Corollary: Fork/Exec

The forked child shares the parent's code but not its stack. While the stack is *initialized* with the contents of the parent's stack, it is a completely separate stack! Moreover, forked processes do not share their data segments (mostly...).

[2]The "blank" approach gives more control to the user in terms of the process' initial state, whereas the template approach is simpler and faster with minimal overhead.

Copy on Write

Definition: Copy-On-Write

Copy-On-Write (COW) allows both the parent and child to share the same physical memory pages for the data segment *until it is modified*. When one of the processes attempts to modify the shared data segment (e.g. writing to a variable), the OS will copy the affected memory pages, create a separate data segment for the modifying process, and allows it to write to the new, copied segment. The other process continues to use the original shared data segment, and no copying is required for unaffected pages.

If the parent has a big data segment, creating a separate copy for the child is expensive! So, we initially have the child process *share* the same data segment as the parent and set it up as a copy-on-write.

Exec

Usually, when we fork we want to run a separate process. `exec` is a Unix syscall that replaces the current code and data segment with new program, "remaking" the process into an entirely different one. When a process calls `exec`, it loads a new program into its address space, overwriting its existing code and data segments with the new program's code and data segments. `exec` will also close all file descriptors associated with the old process (except stdin/out/err) to prevent data corruption and/or resource leaks.

7.2 Destroying a Process

Processes terminate for a multitude of reasons: the machine loses power, the program reaches the end of execution, or the OS or another process kills it. In any case, when a process terminates, the OS needs to clean it up its resources in a way that allows for simple reclamation.

7.2.1 Resource Reclamation

The OS must reclaim any resources held by the terminating process. This includes releasing memory allocated to the process, releasing any locks, and terminating access to hardware devices. This ensures that there are no resources wasted or left in an inconsistent state.

7.2.2 Informing Other Processes

The OS will inform relevant processes that the process has terminated. This includes processes waiting for interprocess communications with the terminated process[3], parent, and child[4] processes.

7.2.3 Update the Process Table

The OS needs to remove the PCB of the terminated process from the process table. Doing this frees up the memory occupied by the process descriptor, making it available for new processes.

Before final cleanup, the OS will collect the exit status of the terminated process.

Example: How'd You Exit?

Assume we create a child process to perform some data retrieval. Once the child process terminates, we get the exit code! Depending on the exit code (success or failure), we can react accordingly! If the retrieval failed, we might run it again or terminate ourself! If the retrieval succeeded, we might continue execution.

[3]The OS will clean up any shared resources or communication channels to prevent potential resource leaks or data inconsistencies.

[4]A child process that outlives its parent is known as an **orphaned** process. In such cases, the OS will adopt these orphaned process under the "init" process (PID 1) to ensure proper process management.

7.3 Running a Process

Processes must execute code to do their job, meaning they need to run on a core! However, the set of processes ready to run usually outnumber the number of cores we have. Thus, the processes need to share the core(s). Sooner or later, a process not running on a core needs to be put onto one. How do we do this?

7.3.1 Loading a Process

Before we run a process on a core, the core's hardware needs to be initialized to either an initial state or to the state the process was in the last time it ran on that core. To run a process on a core, we need to:

(i) Load the core's registers with the appropriate values to start/resume process execution.

(ii) Initialize the stack and set the stack pointer to the appropriate memory location.

(iii) Set up any necessary memory control structures (like page tables for virtual memory).

(iv) Set the program counter to the memory location of the next instruction to be executed.

Now that we have done these steps, we can run a process on a core! But how exactly does it run?

7.3.2 Traps and Exceptions

Definition: Trap and Exception

An **exception** is an abnormal/unexpected event that occurs during the execution of a program, disrupting the normal flow of execution. Some exceptions like EOF[a] are routine while others like segmentation faults are unpredictable (also known as *asynchronous exceptions*).

[a] EOF: End of File

A **trap** is a mechanism that allows the CPU to transfer control from the current process to the operating system or kernel in response to a specific event/condition. Traps typically handle exceptional situations like system calls initiated by user-level processes or other exceptions (like division by zero, page faults).

Asynchronous exceptions are exceptions that are unpredictable and cannot be explicitly checked for by the program itself. Unlike synchronous exceptions which are predictable and are results of specific operations or instructions within a program, asynchronous exceptions can be triggered by external factors or hardware events.

Since asynchronous exceptions are inherently unpredictable, we cannot use typical control structures to handle them. Instead, many programming languages support a `try/catch` block mechanism which forces the developer to handle these exceptions gracefully. Asynchronous exceptions are usually intercepted by the hardware or operating system via a trap. The OS will then handle the exception appropriately.

In addition to handling asynchronous exceptions, traps are used to handle system calls[5]. System calls will cause an exception that will trap into the operating system through a "trap gate". Once inside the OS, the OS will recognize and perform the requested operation based on the syscall number. The OS may use the condition code to indicate the success or failure of the system call. After completing the instruction(s), the OS returns to the instruction immediately after the syscall, allowing the program to continue executing.

[5] System calls are defined at the processor level.

Corollary: Trap Handling

Trap handling is a combination of hardware and software mechanisms to respond to exceptions and traps.

When a trap occurs, the hardware uses the trap cause as an index to access a trap vector table, containing the addresses of trap handlers for different types of exceptions/events. The hardware then loads a new processor status word and switches the CPU to supervisor mode. The current process' program counter and processor status are pushed onto the stack to save the state of the program that caused the trap. Then the program counter is loaded with the address of the first level handler (the starting point of the software trap handling).

The first level handler will then push all other registers onto the stack to preserve the full state of the program at the time of the trap. Then, it reviews the cause of the trap (trap type, error codes, etc.). Based on the cause of the trap, the first level handler will choose the appropriate second level handler to handle the specific exception. The second level handler is a specialized routine that deals with the specified exception.

The second level handler will perform the necessary actions to handle the event. Once done, the second level handler will return control to the first level handler, which, in turn, may run more OS code before eventually returning control to the interrupted program (or terminating it, depending on the situation)

The Kernel Stack

The OS typically uses a separate stack, known as the kernel stack, for running in privileged mode. When a trap occurs, it automatically switches to the kernel stack. This isolates the trap handler's execution, ensuring that it doesn't interfere with the user's stack or leave any sensitive data behind.

7.3.3 Asynchronous Events

Definition: Asynchronous Event

An **asynchronous event** is when a program initiates an operation but does not wait for its completion immediately. Instead, the program continues its execution, and when the operation is finished, the OS will notify the program through an *event completion callback mechanism.*

Example: Why Wait?

Some operations like `read()` are worth waiting for, since we need the data from it to do anything useful! Other times, there might be multiple (independent) outstanding operations that can be done while we wait for the current one we're waiting on.

Event completion callbacks are a common programming paradigm, and are implemented via interrupts, which are similar to traps. They are usually associated with things like I/O devices and timers. When an asynchronous event completes, the associated hardware raises an interrupt to the CPU, signaling that the event has occurred. The CPU then interrupts the currently executing program to handle the event. The OS will then execute the corresponding event completion callback, which allows the program to respond to the event promptly.

Asynchronous event handling is a common programming paradigm to efficiently manage multiple operations and provide responsive behavior to users (like in websites!).

7.3.4 Limited Direct Execution

Definition: Limited Direct Execution (LDE)

Limited Direct Execution (LDE) is an execution model used by operating systems, and consist of two parts. General instructions of a process are directly run on the core without OS intervention. Privileged instructions will cause traps to the operating systems, which then handle the instruction(s) in supervisor mode.

The CPU will directly execute most application code, punctuated by occasional traps for syscalls and occasional timer interrupts for time sharing. The main goal is to maximize direct execution and minimize the time spent in the OS.

System calls initiated by user-level processes will cause traps into the OS, which will then execute the instruction(s) in the OS. Timer interrupts are used by the OS to manage time sharing among processes, ensuring a fair time allocation for efficient multitasking.

7.3.5 Signal Handling in User-Mode

Definition: Signal

A **signal** is an event or notification generated by the OS or other processes to inform a process about exceptional conditions, operator actions, or communication events.

The OS defines various types of signals, each with a unique purpose (e.g. SIGSEGV is a segmentation fault signal). When a process receives a signal, we can:

(i) Ignore the signal, pretending it never happened.

(ii) Designate a handler function that executes when the specified signal occurs. This lets processes respond to different signals accordingly.

(iii) Let the OS generate a default action, which is typically to apply the default action associated with the signal that was sent. Typically, the default action is to terminate the process or generate a *core dump*[6].

User-mode signal handling is analogous to hardware traps/interrupts, but instead of being raised by hardware events, signals are implemented and delivered by the operating system to user-mode processes. This enables processes to respond appropriately to exceptional situations.

7.3.6 Managing Process State

Managing process state is a shared responsibility between the process and OS. Each has its role in ensuring proper process resource management.

The process itself will take care of its own *stack and the data stored within it*[7]. The stack is used to manage function calls and local variables during program execution, and thus the process must properly de/allocate memory for its stack and manage the data stored within it.

The OS will keep track of resources that have been *allocated* to the process. This includes managing memory segments allocated to the process, keeping track of any open files and devices the process has access to, and maintaining a kernel stack (see **7.3.2 The Kernel Stack**).

By sharing responsibility, processes can focus on their own execution and data, while the operating ensures fair resource allocation, handles syscalls and traps, and maintains overall system stability and security.

[6]Core dump: A snapshot of the process' memory used for debugging.

[7]This implies that they can fuck it up if they aren't careful!

7.4 Blocking and Unblocking Processes

Processes can be blocked when they are waiting for certain conditions to be met, and cannot continue execution until they are met (for various reasons). Blocking a process is a way for the scheduler to temporarily suspend execution until the necessary resources are available or a specific event occurs. This blocking state is crucial for efficient resource management and proper synchronization between processes.

Example: Unblock Me!

There are several reasons a process can be blocked. Here are three:

(i) We are waiting for I/O: Reading from or writing to an external source can take a while, so we should let other processes hop on while we wait!

(ii) Resource Requests: If a resource is currently unavailable, there's no reason for us to sit on the CPU doing nothing! So, we block the process until the requested resource becomes available.

(iii) Synchronization: In multi-threaded applications, processes may need to synchronize their operations to avoid conflicts. We might block a process to wait for a specific event or signal from another process before continuing execution.

Any part of the OS can block or unblock a process! But how do we know when to block or unblock a process? A process can request to block itself through a system call[8]. Un/blocking usually happens in a resource manager, a component in the OS responsible for managing various resources and coordinating their allocation among processes.

Example: Resource Management

When a process request an unavailable resource, the resource manager does the following:

(i) Block the process to prevent the process from being scheduled.

(ii) Yield the CPU to the next process in the scheduler's queue.

When the resource becomes available, the resource manager will:

(i) Unblock the process to allow it to be scheduled.

(ii) Notify the scheduler that the process can now be run.

Effectively managing the un/blocking of processes through the resource manager is essential to ensuring fair resource allocation among processes.

[8] You better be sure that someone will unblock you.

Chapter 8

Scheduling

Definition: Scheduling/Scheduler

Scheduling is the process of determining which processes should be executed next.

A **scheduler** is in charge of scheduling as well as allocating system resources (like CPU time) in a fair and efficient manner.

A scheduler often makes decisions about the order and duration of execution for various tasks competing for resources (e.g. there are almost always more processes than cores). The primary goal of a scheduler is to maximize resource utilization and system efficiency while providing responsive and fair service to all processes.

Example: Who Goes Next?

What job should we run next on an idle core, and for how long should we let it run for? What order should we handle a set of blocked requests for a flash drive? In what order should messages be sent if they are all sent at the same time? Questions like these are all answered (for the most part) with scheduling!

Aside: Optimization Metrics

Definition: Metric

A **metric** is a quantifiable measure or characteristic used to asses the performance of a system.

Different scheduling algorithms optimize for different metrics! Do we want to optimize for throughput? Minimize wait time? Optimize for fairness? How do we decide? Luckily, there's a scheduling algorithm for each of these metrics! No single algorithm will optimize for all of these!

Example: Metric Mania

Different systems have different scheduling goals! Here are a couple examples:

(i) Time sharing is when each user gets an equal share of CPU time. This is good when optimizing for response time (e.g. interactive programs).

(ii) Batch execution optimizes to maximize total throughput. Typically, these types of systems will overlook individual process delays in favor of throughput.

(iii) Real-time systems most likely have a priority system. This way, critical operations that *must*[a] happen on time, while non-critical operations might not even happen!

(iv) Service Level Agreements (SLA) prioritize fulfilling agreements with multiple customers regarding resource allocation and performance. So, we need to find a way to ensure that all SLA's are met.

[a]The degree of "must" can vary.

The choice of scheduling strategy heavily depends on the system's intended use as well as specific requirements of its users or applications. Generally, we want to optimize for something we can control. Thus, choosing an appropriate scheduling algorithm is important!

Scheduling Metrics

Operating Systems, Three Easy Pieces: Scheduling Metrics

The book introduces two metrics to measure the performance of the scheduling algorithms discussed in this section: turnaround and response time.

Definition: Turnaround Time and Response Time

Turnaround time is defined to be the time it takes for a job to complete once it arrives on the process queue. Formally, we define $T_{turnaroundtime} := T_{completion} - T_{arrival}$.

Response time is defined to be the time it takes for a job to get scheduled for the first time once it arrives on the process queue. Formally, we define $T_{responsetime} := T_{firstrun} - T_{arrival}$.

To begin, we make a list of assumptions about processes, relaxing them as we go.

(i) Each job runs for the same amount of time.

(ii) All jobs arrive at the same time.

(iii) Once started, each job runs to completion.

(iv) All jobs only use the CPU (i.e. no I/O)

(v) The run-time of each job is known.

8.1 The Process Queue

Definition: Process Queue

The **process queue**[a] is the list of processes that are ready to run on the CPU. Processes that are not ready to run are either not in the queue, at the end of the queue, or ignored by the scheduler.

[a] Typically, the process queue is organized based on the scheduling algorithm we use.

Example: Policy and Mechanism

Scheduling is a great example of policy and mechanism (see **1.3 Policy and Mechanism**). The policy determines which process runs next (by defining priorities and fairness). This policy is usually determined by the system's objectives and which metrics it's trying to optimize for. The mechanism is the implementation of such policies: managing process queues, *dispatching*[a], updating process states, and handling interrupts/context switches.
Notice that the policy and mechanism are independent of each other! The policy layer is not concerned with the low-level details of dispatching, context switches, etc. The mechanism, in turn, follows the guidelines set by the policy to ensure that the system operates as desired.

[a] Dispatching: Moving processes on/off of the CPU.

8.2 Preemptive and Non-Preemptive

Definition: Preemptive and Non-Preemptive Scheduling

Preemptive scheduling is when the OS will *forcibly* interrupt a running process to allocate the CPU to another process. The scheduler has the ability to interrupt a process before it completes its execution, ensuring that all processes get a fair share of CPU time.

Non-preemptive scheduling is when a running process retains control of the CPU until it *voluntarily* relinquishes control or completes its execution. The scheduler does not forcibly interrupt a process, allowing it to run to completion or until it *explicitly* yields the CPU.

Preemptive

Advantages

(i) Faster response times: We don't have to wait for a long running process to finish!

(ii) Fair usage: Each process will run for a fair amount of time.

(iii) Good for real-time and priority scheduling: Preemptive scheduling can handle time-critical tasks and support different priority levels effectively.

Disadvantages

(i) They are more complex to build: Preemptive scheduling requires additional mechanisms to handle context switches and process state saving, making it harder to implement.

(ii) Possibly lower throughput: Frequent preemptions can incur some overhead and reduce overall system throughput.

(iii) Potentially higher overhead: Context switching and frequent preemptions can lead to higher overhead in the system.

Non-Preemptive

Advantages

(i) They are simple: I mean, it's pretty straightforward.

(ii) Low overhead: Since the scheduler doesn't interrupt running processes, there is less overhead involved in context switching.

(iii) High throughput: Processes generally run to completion, which usually leads to better overall system throughput.

Disadvantages

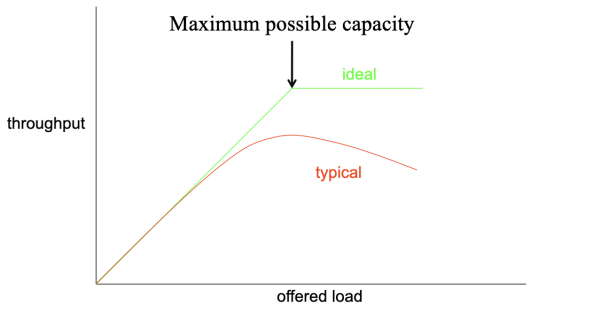
(i) Slower response times: Processes may have to wait for a long time before getting CPU time if a long running process is scheduled before it.

(ii) Unfair usage: Longer running process will inherently take more CPU time than shorter ones.

(iii) Bad for real-time and priority scheduling: Time-critical tasks may not run on time if a process is running and the deadline passes.

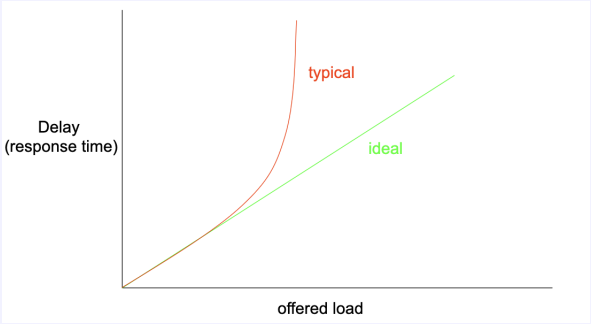
Aside: Expectations v. Reality

Example: Throughput Trouble



This happens because scheduling isn't free! It takes time to dispatch a process (overhead). More dispatches mean more overhead (lost time). Consequently, less time (per second) is available to run processes. Naturally, we can try minimizing the performance gap by reducing the overhead per dispatch and minimize the number of dispatches.

Example: Delay Doozy



This happens because real systems (unfortunately) have finite limits (like queue size). When limits are exceeded, requests are typically *dropped*[a]. From the requester's view, this looks like an infinite response time since the request isn't begin serviced at all! Even if there are mechanisms like automatic retries (e.g. TCP retransmissions), the retries themselves could also be dropped, further exacerbating the situation. During these periods of heavy loads, the system performance drops severely since overheads like context switching and memory management will explode. Careful system design and resource management can help to minimize this problem, but it is not guaranteed to *fix* the problem.

[a]Dropped: We simply don't process the request

Graceful Degradation

Definition: Overload and Graceful Degradation

A system is said to be **overloaded** when it can no longer meet its service goals.

Graceful degradation ensures that the system will continue service, but with degraded performance.

When a system is overloaded, we want to use the graceful degradation principle to at least do *some* work. We *never* want to allow throughput to drop to zero. This will allow response times to grow without limit; i.e. this will lead to infinite response times!

8.3 Non-Preemptive Algorithms

8.3.1 First Come First Serve

The First Come First Serve (FCFS) algorithm consists of running the processes in the order that they arrived. We run the first process on the ready queue until it completes or yields. One of the primary characteristics of this algorithm is that it has highly variable delays since it relies on the process order.

Operating Systems, Three Easy Pieces: FIFO/FCFS

What happens when we relax assumption *(i)* (See **Scheduling Metrics**)? Well, long-running processes will monopolize CPU time, delaying the execution of shorter processes! This results in poorer performance if a long-running process runs first.

While FCFS ensures that all processes will eventually be served, it may not be the most efficient since it can result in poor utilization of the CPU and inefficient response times. Long-running processes will monopolize CPU time, delaying the execution of shorter processes.

8.3.2 Shortest Job First

The Shortest Job First (SJF) algorithm runs the processes in the order of their run-times. We run the process with the shortest run-time on the ready queue until it completes or yields. One of the primary characteristics of this algorithm is that it optimizes for throughput (good for batch processing!).

Operating Systems, Three Easy Pieces: FIFO/FCFS

What happens when we relax assumption *(ii)* (See **Scheduling Metrics**)? Well, we run into the same problem as FCFS if the long-running process arrives first! The long-running process will monopolize CPU time, delaying the execution of the other processes!

8.4 Real-Time Schedulers

Real-time schedulers are used in systems with time-sensitive tasks. Deadlines can either be *hard*[1] or *soft*[2].

8.4.1 Hard

Hard real-time schedulers rigorously enforce deadlines through careful analysis and pre-defined schedules. Often times, this means that we know *in advance* the schedule of processes to be run. Therefore, there is no nondeterminism in your scheduler, and it is inherently non-preemptive. These real-time schedulers are uncommon.

Example: Deadly Deadline

Assume you have to write a scheduling algorithm to control a nuclear power plant. If you miss a particular process deadline, your plant will probably blow up. So, you carefully analyze your processes to come up with the *perfect* schedule that will *never* miss a deadline. Since you know the processes in advance, there is no nondeterminism in your schedule.

8.4.2 Soft

Soft real-time schedulers *want* to meet deadlines, but it won't be the end of the world if they don't. So, we want to optimize our scheduler to avoid missing deadlines. We can do so by giving each process a priority level: the higher the priority, the sooner the deadline.

One possible algorithm is Earliest Deadline First (EDF), which will sort jobs based on deadlines, minimizing total lateness.

If deadlines are missed, the system's response depends on its design and may involve dropping the job, falling behind, or dropping future jobs to compensate.

Example: Choppy Video

When watching a video, the reason why it may be choppy at times is because the scheduler will drop certain packets that missed their deadline! Thus, to the end user it appears choppy, probably because their internet sucks (they might be using eduroam).

8.5 Preemptive Algorithms

Operating Systems, Three Easy Pieces: Preemptive Algorithms

When we relax *(iii)* (See **Scheduling Metrics**), we get preemptive algorithms.

[1] Hard: Under no circumstances can the deadline be missed.
[2] Soft: It isn't the end of the world if the deadline is missed.

Preface: Implementing Preemption

Preemption can be caused by syscalls or clock interrupts. Before returning control to the process, the scheduler is consulted to determine if there are higher-priority ready processes or any processes that need to be woken up. The scheduler will find the highest priority ready process, switching to it (if it's not the current process), effectively preempting the current process.

We can use clock interrupts to do this. Modern CPU's have a clock peripheral device that can generate interrupts at fixed time intervals. These clock interrupts will temporarily halt the currently running process, transferring control to the scheduler, allowing for preemptive scheduling.

8.5.1 Round Robin

The Round Robin (RR) algorithm consists of assigning a time slice to all processes (usually the same size). Processes are scheduled as they arrive and run until they either block or their time slice expires. After running, the process is put at the end of the process queue. Eventually each process will get a turn to run for its allotted time slice.

Some key characteristics of RR is that this usually results in faster response times (for interactive applications), but since more context switches occur, they can be expensive. Additionally, runaway processes have less impact since they only take a fraction of the overall cycles, and will not run forever.

Operating Systems, Three Easy Pieces: Round Robin

Note that "fair" algorithms like RR will perform poorly in turnaround time. Thus, we use response time to measure such algorithms. Now, relaxing assumption *(iv)* (See **Scheduling Metrics**), we can see that there may be gaps where the CPU isn't doing anything! To remedy this, we want to move the process to the back of the process queue whenever the process halts, *regardless* of how (I/O, interrupt, etc.). This way, the CPU is *always* doing something useful.

8.5.2 Choosing a Time Slice

The duration of the time slice is crucial for optimizing performance. Longer time slices reduce the frequency of context switches, but are bad for response times. Shorter time slices are better for response time, but are more expensive since they require more context switches. Striking the right balance is important!

8.5.3 Cost of Context Switches

When a context switch occurs, the OS goes through several steps, handling the interrupt, saving register values, and invoking the scheduler, all of which incur overhead. Additionally, context switching involves changing the stack and handling non-resident process descriptions. Furthermore, the switch requires mapping out the old and new processes, affecting the efficiency of address space changes. More notably, the loss of instruction and data caches significantly impacts the speed of subsequent instructions, reducing overall performance.

8.6 Priority Algorithms

Definition: Starvation

Starvation refers to when processes that have low priority don't run very often or don't run at all.

Priority scheduling assigns each process a particular priority level (higher usually meaning more important). Priority scheduling depends on if the scheduler is preemptive or non-preemptive. Non-preemptive priority schedules simply dictate the order the processes will run in. However, if the scheduler is preemptive, we may have a case where, when a new process is created, it preempts the running process (assuming the new process has higher priority). One concern for preemptive priority algorithms is the possibility of starvation.

8.6.1 Hard and Soft

Definition: Hard and Soft Priorities

A **hard priority** refers to when processes with the highest priority has absolute precedence over lower-priority processes and can preempt them to gain immediate access to the resource.

A **soft priority** refers to when resources are allocated according to their priority level. Higher priority processes receive a larger share of resources, but lower-priority processes aren't blocked entirely.

Example: Linux Priorities

Linux uses soft priorities, represented by a "nice value" assigned to each process. The nice value indicates the share of CPU resources a process should receive. Users can adjust priorities using specific commands, but can only request *lower* priorities. Higher priorities can only be requested by privileged users.

8.6.2 Multi-Level Feedback Queue

Operating Systems, Three Easy Pieces: Multi-Level Feedback Queue

What happens when we relax assumption *(iii)* (See **Scheduling Metrics**)? We get the Multi-Level Feedback Queue! This scheduler is so important that there's an entire chapter dedicated to it!

The Multi-Level Feedback Queue (MLFQ) is a technique that uses multiple ready queues with varying time slices to accommodate different types of processes. We want to optimize both response time for interactive tasks and minimize overhead for longer background tasks. The foreground queue with shorter time slices (high priority) is used for quick responses. The background queue with longer time slices (low priority) minimizes overhead.

When a new process enters the scheduler, it is initially placed in the high priority queue. Once its time slice expires, the process is then moved to the low priority queue. Periodically, all processes are moved back to the high priority queue to prevent starvation!

MLFQ achieves acceptable response times for interactive jobs without wasting CPU resources. Additionally, the scheduler dynamically adjusts based on the actual behavior of jobs, providing an automatic and adaptive scheduler!

Chapter 9

Memory Management

There are three primary goals for memory management:

(i) Transparency: Processes should only be able to see their own address space. That is, their address space is isolated, and processes are oblivious of any memory sharing with other processes. This ensures data privacy and security.

(ii) Efficiency: The system should utilize memory efficiently to optimize resource allocation and minimize waste. The allocation and (when necessary) relocation of memory should be performed with low run-time overhead to maximize system performance.

(iii) Protection and isolation: We want to ensure that the data within a process remains protected and is isolated from outside processes. That is, other processes can neither access it nor modify it.

The memory management problem involves several key aspects:

(i) Unpredictability: Processes usually cannot accurately predict the exact amount of memory they will require during execution.

(ii) Continuity Expectations: Processes expect to find their existing data where they left it, implying that processes assume they have a contiguous address space.

(iii) Limited Physical Memory: The total memory required by all the processes may exceed the available physical memory.

(iv) Efficient Process Switching: We need to optimize the delay for copying data.

(v) Low Overhead: The cost of memory management should be minimized to maximize system performance.

Preface: Physical and Virtual Addresses

Definition: Physical and Virtual Memory Addresses

Physical memory addresses refer to the actual hardware location of a particular memory block.

Virtual memory addresses are an *abstraction* over physical memory addresses, and are *not* the same as physical addresses. They represent a contiguous block of memory, when in reality the physical addresses may be scattered.

Virtual memory addresses are used in processes and allow for more flexibility in memory management, but require a virtual to physical translation unit.

Preface: Fragmentation

Definition: Internal and External Fragmentation

Internal fragmentation occurs when there is wasted space *inside* a block of allocated memory. That is, the requester was given more memory than he needed.

External fragmentation occurs when there is free space in the memory that cannot be used to satisfy any memory allocation request since the available free memory is fragmented into smaller, non-contiguous blocks. As a result, the total amount of free memory may satisfy the request, but since the memory isn't contiguous, we cannot fulfill it.

9.1 Fixed Partition Allocation

Fixed Partition Allocation divides the available memory into fixed-size partitions (go figure), and each partition is assigned to a specific process. The number of partitions is pre-allocated and is based off the expected number of processes and their sizes. Each process can only access the partition assigned to it and cannot access the memory allocated to other processes.

This approach is relatively easy to implement and was used in early batch processing systems. The de/allocation of partitions are straightforward and efficient. However, it is not very flexible since it requires a predetermined number of partitions and may lead to inefficient memory utilization if the partition sizes are not well-matched with the actual memory requirements of the processes.

To enforce memory protection, hardware support is often used. Special registers that hold the partition boundaries ensure that each process can only access memory within its allocated partition. However, fixed partition allocation does not use virtual addresses, meaning the processes use the physical addresses directly.

9.1.1 Problems

There are several problems with Fixed Partition Allocation:

(i) Static Allocation: Fixed Partition Allocation requires knowing the exact memory requirements of all processes in advance, making it challenging to handle dynamic memory demands.

(ii) Limitations: The number of partitions defines the maximum number of processes that can be accommodated concurrently. If the partitions are not efficiently sized or the number of processes exceed the partition count, some processes may not be able to run concurrently.

(iii) Inefficient: Some partitions may be unused or underutilized, leading to poor memory utilization.

(iv) Limited Sharing: Processes cannot easily share memory, making it difficult to implement communication and data sharing between processes.

(v) Internal Fragmentation: If partitions are not efficiently sized, they may succumb to internal fragmentation, leading to wasted memory.

Because of these reasons, this approach isn't commonly used in modern operating systems.

9.2 Dynamic Partition Allocation

Dynamic Partition Allocation allows variable-sized partitions which accommodate almost any size requested. Each partition has contiguous addresses, and processes are allowed to access the partitions it requested. These partitions can be shared between multiple processes, and a single process may have multiple partitions with different sizes and characteristics. However, in the basic scheme, memory addresses are still physical, which can lead to external fragmentation and limited address space.

9.2.1 Problems

There are several problems with Dynamic Partition Allocation:

(i) Not Relocatable: Once a partition is allocated, it is very hard to relocate its contents to another memory location, which can lead to inefficient memory utilization.

(ii) Not Expandable: Dynamic partitions are limited in their ability to grow/shrink to accommodate changing memory requirements of processes.

(iii) Limited Support: Dynamic partitions may struggle to support address spaces larger than physical memory, hindering their ability to handle memory-intensive jobs.

(iv) External Fragmentation: As processes de/allocate memory over time, dynamic partitions can suffer from external fragmentation, leading to wasted memory.

Relocation and expansion can be challenging in dynamic partition allocation because partitions are tied to specific address ranges during execution. Relocating the contents of a partition require updating all the pointers within the contents, which is not always feasible, especially if you don't know which memory locations contain pointers. Additionally, expanding a partition may be challenging since there may not be enough contiguous free space. These limit the usefulness of dynamic partition allocation.

Managing Variable-Sized Partitions

Definition: Free List

The **free list** is a data structure (often implmented with a list or array) that keeps track of all available chunks of unallocated memory.

To manage variable-sized partitions, the memory manager starts with a single large block of memory known as the "heap", and maintains a data structure called the "free list". When a process requests memory, the memory manager will consult the free list, carving a chunk (of the requested size) and putting the remainder back onto the free list. When processes deallocate memory, it is put back onto the free list.

9.3 Free-Space Management

Fixed sized blocks are easy to track: we can use a bitmap to indicate which blocks are free.

Variable chunks require more information. Each chunk of memory has a descriptor containing information about its free status, chunk size, and a pointer to the next chunk. These chunks are typically organized into a linked list.

Variable sized partitions are not as subject to internal fragmentation since processes, in theory, request the exact amount of memory needed. However, they are subject to external fragmentation. We can minimize external fragmentation by consulting an algorithm to determine how we manage our free list.

9.3.1 Best Fit

The best fit algorithm will search for the *smallest* available chunk that meets the size requirements. This minimizes memory waste and reduces internal fragmentation.

Advantages

(i) Increased odds of a (near) perfect fit.

Disadvantages

(i) An exhaustive search is necessary.

(ii) Quickly creates small fragments.

9.3.2 Worst Fit

The worst fit algorithm will search for the *largest* available chunk that meets the size requirements. This tries to minimize memory waste by creating large fragments.

Advantages

(i) Tends to create large fragments.

Disadvantages

(i) An exhaustive search is necessary.

(ii) Over time, small fragments are inevitable.

9.3.3 First Fit

The first fit algorithm will search for the *first* available chunk that meets the size requirements. This reduces overhead since there's no need (potentially) to search exhaustively.

Advantages

(i) Creates random sized fragments.

(ii) Doesn't require an exhaustive search.

Disadvantages

(i) The first chunks quickly fragment

(ii) Over time, the searches become longer.

9.3.4 Next Fit

The next fit algorithm will search for the *first* available chunk that meets the size requirements *starting from the last allocation point* instead of the beginning of the free list. This reduces overhead and tries to minimize fragmentation.

Advantages

(i) Creates random sized fragments.

(ii) Reduces overhead since we don't start from the beginning.

Disadvantages

(i) Over time, memory chunks still fragment.

(ii) Over time, the searches become longer.

9.4 Coalescing Partitions

Coalescing partitions is a technique used to reduce external fragmentation in variable-sized partition allocation algorithms. When a process frees a chunk of memory, the memory management system checks if the neighboring chunks are also free. If they are, the system combines them into a larger, contiguous block, reducing fragmentation.

Operating Systems, Three Easy Pieces: Free-Space Management

To simplify the coalescing process, we can organize the free list such that neighboring chunks are placed close to each other. One way to do this is by ordering the free list by addresses, making it more efficient to find neighboring chunks and merge them when necessary.

Coalescing helps minimize external fragmentation by reducing the number of small, unusable gaps between allocated memory blocks. However, it is worth noting that it doesn't *completely* eliminate external fragmentation since it can only merge *contiguous* chunks.

When multiple processes operate in parallel, it's challenging to predict which process will dominate and how they will interact, leading to potential fragmentation issues. The fraction of space typically allocated depends on the number and size of processes running; if a significant portion of memory is allocated, coalescing becomes less effective due to limited free space.

Additionally, the speed of allocated memory turnover affects coalescing; processes holding memory chunks for extended periods reduce the effectiveness of coalescing. Note that coalescing only minimizes fragmentation since external fragmentation will *always* occur over time.

9.5 Buffer Pools

Certain chunk sizes are requested more frequently than others. Key services like I/O, network protocols, etc. usually work with fixed-size buffers. Thus, we can reserve special pools of fixed-size buffers for popular buffer sizes.

Definition: Buffer Pool

A **buffer pool** is a reserved section of fixed-sized memory buffers used to handle frequently requested memory sizes (like Reiher's favorite 4K), reducing memory management overhead and external fragmentation.

Buffer pools are used to handle frequently requested memory sizes efficiently. When there are popular buffer sizes, the operating system can reserve special pools of fixed-size buffers. When a request for a matching buffer size arrives, it is taken out of the buffer pool as opposed to the free list. This reduces external fragmentation and memory management overhead.

However, the OS needs to determine an appropriate size for the buffer pool. Too small and it might not improve efficiency. Too large and it might lead to a lot of unused buffer space. It is also worth noting that buffer pools will only satisfy *perfectly* matching requests, since otherwise we get internal fragmentation.

9.5.1 Sizing

We dynamically adjust the size of the buffer pool based on the system load and buffer availability. When the pool runs low on fixed-size buffers, we simply acquire more memory from the free list[1] and divide it into new buffers. When the pool is too large, we release some buffers back into the free list. We can tune these thresholds (low space and high space) to determine when to adjust the buffer pool size. This approach makes the system highly adaptive to changing workloads and memory requirements.

9.6 Memory Leaks

Definition: Memory Leak

A **memory leak** is when memory is allocated but never freed, causing the memory to remain occupied indefinitely.

Memory leaks in the context of buffer pools occur when a process is done with a buffer, but fails to free it. This causes the buffer to remain in the pool indefinitely, wasting memory. Long running processes with memory leaks can result in substantial memory waste over time. Addressing memory leaks is crucial if we want efficient memory utilization.

[1] If the free list gets dangerously low, we ask each major service with a buffer pool to return space.

Example: Leaky Program

Assume you have a small program that allocates some memory and immediately terminates. You are surprised that there are no memory leaks! However, this is just because when a process dies, *all* of its memory gets reclaimed, *implicitly* freeing the memory you allocated in your program. But what if you had a `while (true)` loop in your code? Since you aren't doing anything with that memory, it's probably a good idea to let someone else use it! But, since you never explicitly freed it, the memory you allocated is inaccessible (until the process terminates)! Because of this, It is generally bad practice to not *explicitly* free any memory you *explicitly* allocate.

9.7 Garbage Collection

Garbage collection (GC) is a technique to address memory leaks and reclaim unused memory. Instead of relying on processes to release memory on their own, garbage collection monitors the amount of free memory left on the system. When there is *memory pressure*[2], garbage collection is triggered.

The system will search the data space to identify all reachable objects, noting their address and size. Then, any unreachable objects (inaccessible memory) is then reclaimed and added back into the free list. This ensures that memory is efficiently utilized and helps prevent memory leaks!

9.7.1 Determining Accessible Memory

In general, we want to identify and reclaim memory that is no longer used. To do this, we do the following:

(i) Find all pointers in allocated memory: We need to traverse *all* allocated memory to locate the pointers that reference other objects/data.

(ii) Determine the size of each pointer: We must determine the size and extent of the memory region each pointer references.

(iii) Determine what is/n't pointed to: We need to identify objects that are still accessible and in use by actively referenced pointers.

(iv) Free inaccessible memory: We put all inaccessible memory back into the free list.

GC can be difficult because it requires comprehensive scanning of the entire memory space to identify references and determine object boundaries. Furthermore, it also needs to be able to handle complex references (e.g. cyclic data structures) for accurate identification of unused memory. Additionally, GC takes time, and therefore can slow down system performance.

9.7.2 Problems

There are several problems we need to address:

(i) Identifying pointers are *hard*. Locations in a program's data or stack segments may *appear* to contain addresses but are actually just data (that *resembles* addresses). We need to accurately identify valid pointers to avoid reclaiming active memory.

(ii) Even if pointers are identified, we need to determine if they are still accessible and in use! This requires recursive analysis of dynamically allocated data structures to ensure that all referenced memory remains reachable. Even so, statically allocated data structures are harder to analyze.

(iii) We also need to determine the size of each object pointed to, which can be challenging! Measuring exact boundaries may be hard for complex data structures or objects with variables sizes.

[2]Memory pressure: When the amount of free memory becomes dangerously low.

9.8 Memory Compaction and Relocation

GC is simply another method to release memory, and therefore doesn't significantly impact fragmentation. Ongoing memory de/allocations can prevent coalescing, leading to fragmentation over time[3].

To counter this, we can compact active memory to one end, coalescing the other end to eliminate fragmentation! However, this requires relocation, which is extremely complicated, since we need to update all memory references correctly.

When a process is relocated, all addresses within the program will become invalid, resulting in potential errors and crashes. We would need to:

(i) Update all references in the code segment (e.g. calls and branches to other parts of code) to point to the correct memory addresses in the new location.

(ii) Update referees to variables in the data segment to point to the correct addresses in the new location.

(iii) Ensure that new pointers are adjusted to point to the correct addresses after relocation.

To solve the relocation problem, we can make the process location independent! By doing this, we can avoid the complexities of update memory references, *abstracting* away *(i)* – *(iii)*! We want to enable processes to execute in *any* part of memory without adjusting memory addresses.

We can achieve this using various techniques:

(i) Relative addressing: We can use offsets instead of absolute memory addresses to allow processes to be loaded into different memory locations with no address modifications.

(ii) Base and Bounds Registers: We can use hardware registers (base and bound registers) to automatically adjust memory references at runtime. These registers keep track of the base address and size (bound) of the process' memory space, allowing the CPU to translate relative addresses into absolute addresses during execution.

(iii) Virtual Memory and Paging: We can use VM techniques like paging to abstract the physical memory from the process' address space. This way, the process can work with the virtual addresses that get translated to physical addresses, making the process independent of its location.

Abstraction: Virtual Memory

Virtual memory is an *abstraction* over physical memory! A virtual address is *not* the same as its corresponding physical address, and requires a translation unit to convert between the two. However, this is a small price to pay for making memory relocation *significantly* easier!

9.8.1 Segment Relocation

Memory segment relocation is a technique that involves organizing a process' address space into multiple segments, each representing a contiguous block of memory with a specific purpose (see **6.1 Process Address Space**). The segments are then moved as a unit during relocation.

9.8.2 Base and Bounds Registers

Computer architecture may include special relocation registers known as segment *base registers*. These registers hold the starting address of each segment in physical memory. When the CPU accesses a memory location, it will add the address to the address of the base register, translating the virtual address to the physical address!

When a program is loaded into memory, the OS sets the base registers to the start of the program. When relocating, we simply update the base register accordingly. This way, the program can continue to run smoothly regardless of where it is located in physical memory.

[3] Frequent allocations can starve coalescing, reducing its effectiveness.

Corollary: Security

We still need to protect our memory! Protection refers to preventing a process from accessing memory outside its allocated memory. To achieve this, we utilize a length (or limit) register, often called the *bounds register*, which specifies the maximum valid offset from the start of the segment. Any address greater than the limit is considered illegal and should be inaccessible to the process.

When processes attempt to access an illegal address, the CPU triggers a segmentation exception or trap. This exception traps into the OS allowing it to take appropriate action (e.g. terminating the process or controlling the violation).

9.9 Swapping

Swapping is a technique to overcome the limitation of physical RAM by temporarily storing inactive processes' memory on disk. When a process isn't actively running (e.g. yields, blocked), its entire memory contents are copied to disk to free up RAM for other processes.

When a process is scheduled to run again, its memory is then copied back from disk onto RAM to continue execution. If the system has relocation hardware, the memory can be placed in different RAM locations, enabling processes to access their memory regardless of their physical addresses. This allows the system to use disk space as a virtual extension of physical memory, giving the *illusion* that each process gets all of RAM to itself.

However, swapping incurs overhead since copying is expensive! The cost of a context switch are *very* high, since we need to:

(i) Copy all of RAM out to disk.

(ii) Copy other stuff from disk to RAM.

before the new process can do anything. Moreover, we still cannot exceed the amount of physical RAM, which can limit memory-intensive processes or overall system performance.

9.10 Paging

Paging is a technique that divides both physical memory and virtual address space into fixed-size units[4] called *pages*. The pages in physical memory are referred to as *page frames*, while pages in virtual memory are just called *pages*.

Each virtual page is mapped to a physical page frame, but it's not fixed nor is it one-to-one. Instead, a per-page translation mechanism called a *memory management unit (MMU)*[5], is used to dynamically translate virtual page numbers to corresponding physical page frame numbers.

9.10.1 Big Page Tables

Definition: Translation Lookaside Buffer

The **Translation Lookaside Buffer (TLB)** is the MMU cache that stores a subset of recently accessed page table entries. This improves lookup times since we don't have to access page table entries from main memory for *every* memory access.

Traditionally, page tables were implementing using fast registers in the MMU. But with larger memory sizes and smaller page sizes, there will be a *lot* of pages.

Example: How Many Pages?

Suppose you have 64 GB memory with 4K page sizes. There would be $\frac{64GB}{16KB} = 16$ *million* pages! Unfortunately, we cannot afford to store 16 million pages into fast registers.

[4] Usually 1-4K bytes or words.

[5] The Memory Management Unit (MMU) is hardware (often integrated into the CPU) responsible for handling memory access and virtual-to-physical address translation.

Definition: TLB Hit and Miss

A **TLB hit** is when the required translation is found in the TLB.

A **TLB miss** is when the required translation is *not* in the TLB, and an access to the page table is necessary.

To remedy this, TLB's act as a high-speed cache for virtual-to-physical address translations, reducing overhead. When the CPU needs to translate a virtual address, it first checks the TLB. If the translation is found in the TLB, we return it. Otherwise, we need to consult the page table in main memory to retrieve the required page table entry. The TLB is then updated accordingly.

Unfortunately, the TLB has a limited size, and therefore not all entries can fit into it. This lead to the issue of cache invalidation and replacing entries when the TLB is full. Maintaining a high TLB hit ratio is crucial for efficient VM performance.

9.10.2 Swap Space

Definition: Page Fault

A **page fault** is when the required address is not in the TLB, is valid, but is not present (i.e. it is in the swap space or page file).

Since we have more pages than RAM, we need to store some of them somewhere other than RAM. Typically, some pages are kept on disk and are referred to as the *swap space* or *page file*. When a page fault occurs, the OS retrieves[6] the required page from the swap space, loading it into an available page frame in RAM. The program counter is backed up to retry the failed instruction after the page is loaded, allowing the process to continue running.

9.10.3 Ongoing Operations

The MMU has many ongoing operations. Here are three important ones:

(i) Adding/Removing Pages: When the current process dynamically de/allocates memory, the MMU needs to reflect this in the page table. The OS will directly update the active page table in memory to adjust the relevant page mappings. A privileged instruction is used to flush any stale cached entries in the MMU to ensure an accurate mapping.

(ii) Context Switching: When the system switches processes, the MMU needs to switch the appropriate page table for the new process. Each process has a separate page table, and a privileged instruction is used to load the pointer to the new page table. Before the new process begins execution, a reload instruction flushes any previously cached entries, preventing invalid access.

(iii) Page Sharing: Page sharing allows for memory sharing between multiple processes. The page tables can be configured to point to the same *physical* page. This means multiple processes can have access to the same page in RAM. Page sharing can be read/write or read-only depending on access requirements and memory protection mechanisms enforced by the OS. This approach is good for sharing read-only data (e.g. code segments, shared libraries), reducing redundancy.

9.10.4 Demand Paging

Demand paging is a technique where not all pages of a process are loaded into RAM at once. Rather, only pages that are actively being referenced are brought into memory when needed, allowing for better memory efficiency and less overhead during process scheduling and context switching.

Demand paging frees up RAM by keeping the majority of a process' data on disk until it's actually needed, enabling the system to accommodate more processes and improves overall memory utilization. Demand paging utilizes page faults to signal when pages need to be fetched from disk.

[6] In the meantime, other processes can execute

One problem of demand paging is performance optimization. Frequent page faults incur overhead since the time it takes to fetch from disk can add up. Efficient page replacement algorithms like *Least Recently Used (LRU)*[7] are used to minimize the number of page faults triggered to ensure that relevant pages are kept in RAM.

9.10.5 Locality of Reference

Locality of reference suggests that the next address a program will access is most likely to be close to the one it just accessed. This is usually present in programs since they often execute sequences of consecutive (or nearby) instructions, have short branches, access data in the current/previous stack frame, and (tend to) access recently allocated heap structures. While there are no guarantees, identifying these trends help reduce the number of page faults.

[7]The page that hasn't been accessed for the longest period of time is booted from RAM.

Chapter 10

Virtual Memory

Virtual memory (VM) is a generalization of demand paging, and is a technique that provides a large and uniform address space to each process. It gives the *illusion* that each process has access to a vast amount of memory (much larger than the physical RAM available).

Abstraction: Virtual Memory and Paging

Virtual memory is an *abstraction* over demand paging. It extends the concept of paging by providing a much larger address space to each process by using dynamic paging and swapping.

Processes can directly request segments in this space, and the virtual memory system handles the mapping of virtual to physical addresses via page tables. VM allows processes to run even if their entire address space doesn't fit into physical memory. Instead, only actively referenced portions are loaded into RAM, while the rest sits on disk.

10.1 Replacement Algorithms

Replacement algorithms are the key technology to making virtual memory work. We want to have the relevant pages loaded into RAM when a process needs to access them.

We do this by relying on the principle of locality of reference. Doing this allows us to make smart choices about which pages to keep in memory and which ones to kick to disk.

Page replacement happens when we need to free up space in memory for new pages. Whenever a page fault occurs, we select an appropriate page to replace via an algorithm (like LRU).

10.1.1 The Optimal Algorithm (Belady's Algorithm)

The optimal replacement algorithm replaces the page that will be accessed furthest into the future, minimizing the number of page faults. However, this requires an oracle, and as such, it is impossible to implement. This algorithm is also known as "Belady's Algorithm".

10.1.2 FIFO and Random

According to Reiher (and common sense), these are dogshit so I'm not going to cover them.

10.1.3 Least Frequently Used

The Least Frequently Used (LFU) policy kicks the least frequently used page back to disk. It isn't the best in the world.

10.1.4 Least Recently Used and Clock Algorithm

Least Recently Used (LRU) policy kicks the page with the oldest timestamp back to disk. This is done (naïvely) by timestamping each time a page is accessed. When a page needs to be replaced, we

search through all pages to kick the one with the oldest timestamp. This incurs a lot of overhead and therefore is not commonly used.

Rather, we usually use an approximate LRU, or "Clock Algorithm". We maintain a circular buffer of pages in memory, with each page having an additional reference/use bit (typically stored in the MMU). When a page is accessed, we set the reference bit to 1, indicating it has been recently used. When determining the page to replace, we scan starting at a fixed point, replacing the first page with reference bit 0.

This algorithm, while not a perfect one-to-one with LRU, offers performance on par with LRU for a fraction of the cost. Therefore, it is usually implemented in favor of a "true" LRU.

10.2 Page Replacement

We don't want to clear out *all* page frames on each context switch as it can be inefficient. There are several ways to deal with this:

(i) Single Global Pool: All processes share a single pool of page frames in memory. When a process runs, it uses any available free page frame for its page.

(ii) Fixed Allocation per Process: Each process is assigned a fixed number of page frames when it starts. The number of page frames stays constant throughout execution.

(iii) Working Set-Based Allocations: The working set of a process represents the set of pages it is actively referencing at any given time. Page frames are allocated dynamically based on a process' working set. When a process runs, its working set is loaded into RAM, and when it's switch out, its page frames are potentially freed.

10.2.1 Single Global Pool

In the Single Global Pool approach, all page frames in memory are treated as a shared resource, and an approximation of LRU is used as the replacement algorithm. However, this sucks when paired with round-robin scheduling (see **8.5.1 Round Robin**).

Example: Fair or Unfair

In RR scheduling, the process that was last in the queue will find all of its pages swapped out. Thus, when this process runs, it will experience a high volume of page faults since all of its pages were replaced.

This is because this approach doesn't account for the specific working sets of each process. For this reason, this approach is usually pretty dogshit.

10.2.2 Per-Process Pools

In the Per-Process Pool approach, a fixed number of pages are allocated for each process, and the approximate LRU is used *separately* for each process. This allows for dynamic and customized allocation of page frames to each process.

However, a fixed number of pages per process sucks because different processes exhibit varying levels of locality, and the pages needed by each process usually change over time. Moreover, processes have different natural scheduling intervals, and as such, their memory requirements vary throughout execution!

10.2.3 Working Sets

Definition: Working Set

A **working set** is defined to be the set of pages a process actively referenced within a fixed sampling interval in the immediate past.

Working sets are used to allocate page frames to each running process based on its specific memory needs. We allocate a sufficient number of page frames to hold each process' working set. This ensures that the frequency of accessed pages are kept in memory, reducing the volume of page faults. Each process will manage its own set of pages, usually using an approximate LRU for page replacement.

Working sets dynamically adjust the allocation of page frames for each process based on its current behavior, optimizing memory usage and responding to changes in workload and access patterns.

Optimal Working Sets

The optimal working set for a process is the set of pages it needs during its next time slice (or a specific period of time). Allocating less than the optimal amount leads to a lot of page faults.

We determine the size of the working set by observing the process' behavior over time. Tracking the frequency of page faults and memory references, the system can identify which pages the process frequently accesses, including them in the working set.

Page Stealing: Working Set-Clock Algorithm

The Working Set-Clock algorithm tracks the last use time for each page for its owning process, replacing the page that was least recently used.

10.3 Thrashing

Definition: Thrashing

Thrashing refers to when a system spends a significant amount of time/resources swapping pages between RAM and disk, but isn't able to efficiently make any progress in the executing process.

Thrashing occurs when the total demand for memory by *all* running processes exceeds the available physical memory. When thrashing happens, we seldom execute any useful instructions since so much time is spent swapping pages. This results in an underutilized CPU and degraded performance.

Thrashing typically happens hen the working set of each process exceeds the available physical memory. When there are not enough page frames to accommodate the working sets of all active processes, they constantly compete for memory.

To protect against thrashing, the OS takes proactive measures like reducing the degree of multiprogramming (i.e. limiting the number of active processes), using page replacement algorithms that prioritize larger working sets, or allocating more physical memory (lol). The goal is to avoid thrashing by ensuring that each process has enough pages in memory to execute efficiently.

Example: Everyone Gets a Turn

When reducing multiprogramming, we can use a RR approach for swapping processes in and out of disk, ensuring that all processes get a fair share of CPU time while minimizing thrashing!

10.4 Clean and Dirty Pages

Definition: Clean and Dirty Pages

A **clean page** refers to a page in memory that hasn't been modified since it was brought from disk. They can be safely replaced without needed to write them back to disk since their contents match that of the one on disk.

A **dirty page** refers to a page in memory that has been modified or updated after it was brought from disk. That is, the in-memory version of the page is different from the one on disk. If a dirty page needs to be removed from memory, the page must be written back to disk to update the appropriate page on disk, ensuring that the most recent version of the data is saved before kicking it from memory.

When given a choice, the OS will prioritize kicking clean pages since they can be safely removed without the need for I/O. Dirty pages however, need to be written back to disk to preserve data integrity.

10.5 Preemptive Page Laundering

Preemptive page laundering is a technique used to increase the flexibility of the memory manager by converting dirty pages to clean ones. Rather than waiting to write when pages gets kicked, we initiate a background write-out of dirty pages that are not actively in use. This way, we reduce the risk of thrashing and increase the number of clean pages we have.

Part II

Concurrency

Chapter 11

Threads

Definition: Thread

A **thread** is a unit of execution and scheduling in a program. Each thread has its own stack, program counter, and registers, allowing it to operate independently of other threads.

Threads within the same process share the same code and data segment, making them more efficient and less resource-intensive than processes.

In multi-threaded programs, multiple threads can run concurrently within the same process. They share the process' resources (e.g. memory, files), but each thread maintains its own execution state. This allows for better communication and coordination between different parts of the program.

They can be implemented and managed in various ways. User-level threads are managed by the process itself and rely on voluntary yielding. Scheduled system threads are managed by the OS and can be preemptively scheduled, meaning the OS can interrupt it to give other threads CPU time.

Corollary: Why Not Processes?

Processes are expensive! Since each one has private resources and a private address space, it makes interprocess communication difficult. Additionally, certain programs may not require such strong separation. So, we can use threads to remedy such limitations of processes.

11.1 Process v. Thread

When should you use a process? A thread?

Processes

(i) Running multiple, *distinct* programs.

(ii) Creation/destruction are *rare* events.

(iii) Running with distinct privileges.

(iv) Limited interactions/shared resources.

(v) Strong separation between other processes.

Threads

(i) Parallel activities in a *single* program.

(ii) Creation/destruction are *frequent*.

(iii) All can run with the same privilege.

(iv) Need to shared resources.

(v) Frequent message/signal exchange.

(vi) When you don't need protection from each other.

11.1.1 Tradeoffs

Processes

Advantages

(i) String isolation: Processes have distinct address spaces.

(ii) Fault tolerance: Processes don't affect other processes.

(iii) Easier resource cleanup: When a process terminates, all resources are automatically freed.

Disadvantages

(i) Slower communication: Interprocess communication is more complex and slower.

(ii) Potential duplication: Since each process has a distinct address space, this allows for unnecessary duplication.

Threads

Advantages

(i) Efficient communication: Threads share resources making communication and data sharing much faster.

(ii) Lightweight: Threads have lower overhead since they share resources.

(iii) Faster context switching: Switching between threads is faster than switching between processes because of *(i)* and *(ii)*.

Disadvantages

(i) Synchronization issues: Threads must be synchronized to run properly.

(ii) Increased complexity: Multithreaded code is hard.

11.2 Thread Stacks and State

Each thread has its own stack, registers, program counter, and process status. The maximum stack size for each thread is specified at creation, and needs to be managed carefully to avoid stack overflow or memory waste. Since a process can contain many threads, they cannot all grow towards a single hole. Thus, the thread creator needs to know the maximum required stack size. Moreover, stack space must be reclaimed whenever threads exit.

11.3 User v. Kernel Threads

Kernel

(i) Provided and managed by the OS kernel.

(ii) Share the same address space.

(iii) Scheduled by the kernel.

User

(i) Managed by the user with no intervention from the OS kernel.

(ii) Invisible to the kernel.

(iii) Scheduled by the user.

Chapter 12

Interprocess Communication

Definition: Interprocess Communication

Interprocess communication (IPC) refers to the techniques provided by the OS to enable communication and data exchange between different processes.

There are several IPC mechanisms:

(i) Pipes: A unidirectional communication channel that allows data to flow from one process to another.

(ii) Named Pipes (FIFO's): Similar to *(i)*, but can be accessed by multiple processes for bidirectional communication.

(iii) Message Queues: Allows processes to exchange messages via a system managed message queue.

(iv) Shared Memory: Allows processes to share a region of memory.

(v) Sockets: A network communication mechanism that enables processes running on different machines to communicate.

The OS supports IPC by providing system calls. They typically require activity from *both* communicating processes and are mediated by the OS for protection and to ensure correct behavior.

12.1 Goals

IPC aims to achieve:

(i) Simplicity: The mechanism should be easy to understand, use, and implement to avoid unnecessary complexities.

(ii) Convenience: It should provide a convenient interface for developers to exchange information and coordinate actions between processes.

(iii) Generality: The IPC mechanism should be flexible and versatile enough to handle various types of IPC scenarios.

(iv) Efficiency: It should be efficient in terms of time and resource usage to minimize overhead and maximize performance.

(v) Robustness and reliability: The IPC mechanism should be resilient to errors and failures, ensuring that communication is consistent and dependable.

Some of these goals are contradictory, and thus multiple different IPC mechanisms are provided to optimize for different goals.

12.2 Synchronous and Asynchronous

12.2.1 Synchronous

Both read/write operations block until the data is sent, delivered, or received, respectively. This means that the processes involved have to wait until the data exchange is complete before continuing execution. It is simple for programmers to understand but can introduce delays if processes frequently need to wait for data.

12.2.2 Asynchronous

Both read/write operations return promptly. Reads return quickly even if no new data is available. Writes return after the system accepts the data without waiting for confirmation of transmission, delivery, or reception.

Since asynchronous IPC doesn't block processes, they can continue execution, which may introduce data synchronization problems.

Example: 404

Assume you set up asynchronous IPC, and have process A read some data from process B. Since it's asynchronous, A won't wait for B to send everything over and will continue executing! So, it may be the case that your code starts executing instructions on data that you don't have.

To handle asynchronous operations, we need to introduce an auxiliary mechanism to learn when there's new data, often called a "wait for any of these" operation. This operation allows processes to efficiently wait for any of the asynchronous operations to complete.

12.3 Mechanics

Typical IPC operations include the following:

(i) Create/destroy an IPC channel: These operations involve setting up and tearing down communication channels between processes. The creation of a channel allows processes to exchange data with each other, while its destruction terminates the communication link.

(ii) Write/send/put: This operation involves inserting data into the channel, allowing a process to send information to another process. The data placed in the channel will be made available for the receiving process to read.

(iii) Read/receive/get: The read operation extracts data from the channel, enabling a process to receive information sent by another process. The data is retrieved from the channel and made available for processing by the receiving process.

(iv) Channel content query: This operation allows processes to check the amount of data currently present in the communication channel. This is useful to monitor the status of the channel and ensure efficient data transfer.

(v) Connection establishment and query: Processes may require control over how their channel ends are connected to each other. These operations involve setting up and managing the connections between the two ends of the channel. Information like the identities of the end-points and the status of connections can also be queried using these operations.

12.4 Messages and Streams

Each style of data exchange are suited for particular kinds of interactions.

12.4.1 Streams

Stream-based IPC is when data flows in a continuous stream of bytes. Processes can read/write in various sizes, and the size of read/write buffers are not directly related. This gives greater flexibility in handling data of different sizes. Streams are more suitable for continuous data exchange like real-time streaming, where data arrives and is processed in an uninterrupted flow.

12.4.2 Messages

Messages (or datagrams) transmit distinct messages, each having its own length (with limitations). They are typically read/written as a whole unit; i.e. each message is treated as a separate entity. Messages are more suitable for discrete data exchanges, where separate units of data needs to be transmitted and processed independently.

12.5 Flow Control

Definition: Flow Control

Flow control is the process of regulating data transmission between a fast sender and a slow receiver so as to not overwhelm the reader with data.

In queued IPC, data is buffered n the OS until the receiver is ready to accept it. However, various factors can increase the required buffer space (e.g. fast sender, non-responsive receiver). To limit the required buffer space and ensure effective flow control, we can:

(i) Enforce sender-side flow control: The sender can be blocked or refuse communication if the receiver's buffer is full.

(ii) Enforce receiver-side flow control: The receiver can block the sender or flush old data if it cannot keep up.

(iii) Implement feedback mechanisms: Network protocols or the OS can provide feedback to the sender so that it can adjust its data transmission appropriately.

12.6 Reliability and Robustness

Within a single machine, the OS ensures that data isn't lost accidentally during transmission. While on a single machine, data is never lost, it may never get processed, since the receiver could be invalid, dead, or unresponsive.

Data can get lost when communicating across a network due to network issues/failures. When this happens, we need additional mechanisms like acknowledgments and retransmission protocols to guarantee reliable data delivery.

Reliability involves determining when to acknowledge successful delivery, the level of persistence in delivery attempts, and the handling of IPC data after receiver restarts. The timing of acknowledgment can be when the message is queued locally in the sender's system, added to the receiver's input queue, or explicitly read by the receiver.

For network communication, the system may attempt multiple retransmissions and explore alternate routes or servers to ensure delivery. Whether IPC data survives receiver restarts depends on the application; some systems may allow persistence for seamless data continuation, while others may require resending messages after restarts. The choice of reliability options depends on the application's requirements and the trade-off between reliability and overhead.

12.7 Pipelines

Pipelines allow for data to flow through a series of programs, passing a simple byte stream buffered in the OS[1]. It is a straightforward and efficient way to pass data between programs.

[1] We don't need temporary files!

Example: Pipe Up!

Consider the following: `ls | grep 'CS111'`. This is an example of a pipe! the data from `ls` gets sent to the standard input of `grep`! We also implemented this in lab 1.

They are secure and *trusted*[2], but can be limiting. Error conditions include EOF and detecting failures in the programs that are part of the pipeline.

12.8 Sockets

Sockets allow IPC between addresses and ports (communication between processes on different machines). They offer various data options (e.g. reliable or best-effort), streams, messages, remote procedure calls, etc. They involve complex flow control and error handling with features such as retransmissions, timeouts, and handling node failures.

They allow for reconnection and fail-over in case of disruptions. However, this generality adds complexity such as trust, security, privacy, and integrity concerns. They are usually used for network communication due to their flexibility.

12.9 Shared Memory

Shared memory is when the operating system allows processes to share read/write memory segments that are mapped into multiple process' address spaces. The OS doesn't mediate data transfer between processes. Rather, it provides the memory segments and trusts that the applications manage sharing control.

This direct memory access allows for faster communication, but the simplicity of the approach also assumes that cooperating processes will handle synchronization and data integrity. Shared memory also only works on local machines, limiting its scope to IPC on a single device.

12.9.1 Synchronization

Synchronization is the process of coordinating and ensuring multiple events/actions occur in the correct order. This is an issue that multi-threaded applications must deal with, and can get complex. It is crucial for *parallelism*[3], and to ensure correctness in our programs.

Aside: Parallelism

Parallelism gives us many benefits! It lets us run multiple tasks concurrently, improving throughput. Parallelism also facilitates breaking down complex tasks into smaller, more manageable pieces, supporting modularity. When using parallelism, if one thread/process fails, it doesn't affect the others, improving robustness and isolating problems.

Aside from local benefits, parallelism also support common paradigms like client-server computing, which are inherently parallel in nature. Furthermore, real-world phenomena involve the cooperative interaction of multiple processes or entities, and we can use parallelism to model these behaviors!

[2]They are trusted since all programs are controlled by a single user.
[3]Parallelism: Multiple threads/processes executing concurrently.

Chapter 13

Synchronization

Synchronization refers to the process of coordinating the execution of multiple concurrent threads/processes to ensure orderly and predictable behavior in a parallel system. It involves two interdependent subproblems: the critical section serialization and the notification of asynchronous completion.

While true parallelism can be complex and difficult to understand, many systems employ pseudo-parallelism, where the focus is on controlling and coordinating key points of interaction rather than attempting a truly parallel execution. Synchronization mechanisms often address both problems simultaneously, as solving one implicitly addresses the other. However, understanding and solving critical section serialization and notification of asynchronous completion can be solved separately to manage the complexities of parallel systems effectively.

13.1 Race Conditions

Race conditions occur when the outcome of a program depends on the order in which concurrent threads/processes run. As such, they can affect the correctness of a given program.

Example: No I Wrote First!

A race condition can happen when we read and write to the same piece of data from different threads (that aren't synchronized). `counter = counter + 1` is a great example! Who knows *when* each thread will increment `counter`, and which value they'll go off of? No one!

We employ strategies like mutual exclusion, synchronization, and transactions to try and mitigate race conditions in concurrent systems.

Corollary: Nondeterminism

Nondeterministic execution is more general than race conditions, and refer to *any* execution that make behavior less predictable.

Example: I/O

Suppose you write code that takes in input. When your program waits for a read from your keyboard, the time it takes to read the data may vary drastically, depending on how dumb your user is! This makes I/O inherently nondeterministic.

Addressing these issues requires careful synchronization and coordination mechanisms.

13.2 The Critical Section

Definition: Critical Section

A *critical section* is a code segment where shared resources are accessed and modified. Naturally, it must not be accessed simultaneously by more than one entity to avoid race conditions and other synchronization issues.

The state of the shared can be altered by the critical section, including changes to its contents or relationships with other resources. Therefore, it is crucial to control access to it to avoid conflicts and preserve the integrity of the shared resource.

Correctness depends on the execution order of the threads/processes/CPU's, which is influenced by the scheduler. Additionally, the relative timing of asynchronous and independent events can also impact the behavior of critical sections and therefore the overall system. Managing critical sections involves coordinating the access to the shared resource to prevent undesirable interleavings

13.3 Interrupt Disables

One solution to the critical section problem is to use interrupt disables, which temporarily block some or all interrupts! By temporarily blocking some or all interrupts, we can ensure that there will be no preemption by interrupt handlers (or threads) during a critical section.

Interrupt disables have a multitude of abilities and dangers. On the bright side, they prevent time-slice interrupts and avoid re-entry of device driver code, ensuring that the critical section is executed without interruption. However, some risks of disabling interrupts can include delaying important operations (like preemptive scheduling) or being permanently disabled due to a bug. Additionally, disabling interrupts isn't an option in user mode, and requires the use of privileged instructions (for safety purposes). It is worth noting that they don't solve *all* synchronization problems, especially on multi-core machines.

13.4 Mutual Exclusion

Definition: Mutual Exclusion

Mutual exclusion ensures that only one thread can execute a critical section at a time. To ensure proper synchronization, we need to enforce mutual exclusion; i.e. if one thread is running the critical section, the other definitely isn't.

13.4.1 Atomicity

Definition: Atomicity

Atomicity is defined by two aspects: "Before or After" and "All or None".

(i) Before or After: Given two threads A and B, if A enters the critical section *before* B starts, B will enter the critical section *after* A completes (and vice versa). That is, there is *no* overlap between threads.

(ii) All or None: Updates within the critical section are performed entirely or not at all. If an update starts, it will either complete successfully and apply changes or revert back to its original state (before the critical section).

Achieving both aspects of atomicity is essential for correctness and consistency.

13.4.2 Locking

Locking is a technique used to protect critical sections. It involves a data structure called a "lock" which serves as a synchronization mechanism.

When a thread wants access to a critical section, it will attempt to acquire the lock associated with it. If it's available, the thread locks it and can access the critical section. Otherwise, the thread must wait (e.g. blocked, suspended) until the lock becomes available.

Locks ensure that only one thread can hold a lock at any time, enforcing mutual exclusion and preventing multiple threads from simultaneously accessing a critical section. Proper use of locks avoid race conditions and maintain correctness of parallel programs. However, they have their separate issues like deadlocks and lock contention.

Implementing Locks

Unfortunately, ISA's usually don't include instructions for building locks, so we need to build locks in software. However, this raises other issues of enforcing *their* mutual exclusion. Luckily, we can solve these issues with hardware assistance!

Individual CPU instructions are atomic, so we want to implement a lock with a *single* instruction! Remember, acquiring a lock requires that we:

(i) Check no one else has it.

(ii) Change the lock to "acquired".

Lucky for us, hardware designers have solutions for that!

Example: Test and Set

Below is a *representation* of how locks are implemented. Remember, the *real* instructions are silicon, not in C!

```
bool test_and_set(char *p) {
  bool rc;
  rc = *p;   // note the current value
  *p = true; // set the value to true (i.e. acquired)
  return rc; // return the OLD value
}
```

Now, when we evaluate `if !test_and_set(flag)`, we know that if `rc` was false, no one else ran it, so we can acquire the lock! If `rc` was true, then someone else already ran it, so they have the lock.

Corollary: Spin Waiting

When you don't get the lock, you can do something known as *spin waiting*! Essentially, you put the lock request in a while loop until you get the lock.

Advantages

(i) It properly enforces access to critical sections! This also assumes you implemented locks properly.

(ii) They're simple to program. I mean it's usually just a while loop.

Disadvantages

(i) It's wasteful. Spinning uses CPU cycles which is inefficient!

(ii) The cycles burned could be used by the locking party to finish its work!

(iii) Bugs can lead to infinite spin-waits.

13.5 Asynchronous Completion

The asynchronous completion problem occurs when parallel activities run at different speeds, and one activity needs ton wait for the other to complete without hindering performance. examples include I/O, network request responses, or real-time delays.

13.5.1 Spinning

Spinning sometimes makes sense:

(i) When the operation proceeds in parallel, such as a hardware device accepting a command or another core quickly releasing a held spin lock.

(ii) When the operation is guaranteed to happen soon, spinning can be less expensive than using sleep/wakeup mechanism.

(iii) When spinning does not significantly delay the operation or impact system resources, like when burning CPU cycles does not hinder other processes or slow I/O operations due to memory bandwidth.

(iv) When contention for the resource is expected to be rare, as having multiple waiters for the same resource could substantially increase overhead and inefficiencies.

13.5.2 Yield and Spin

Yield and Spin is a technique where a thread repeatedly checks if a particular event has occurred, yielding the CPU if it hasn't. The thread keeps checking and yielding until the event is ready. This approach avoid busy-waiting, maximizing useful CPU time.

Problems

Some problems of Yield and Spin include:

(i) Extra Context Switches: Frequent yielding and spinning can lead to additional context switches, which are expensive operations and can impact overall system performance.

(ii) Wasted Cycles: If a process spins every time it is scheduled, it can waste CPU cycles, especially if the event it is waiting for doesn't occur in the expected timeframe.

(iii) Delayed Scheduling: There's no guarantee that a process will be scheduled to check for the event in a timely manner, potentially causing delays in responding to the event.

(iv) Poor Performance with Multiple Waiters: When multiple processes are waiting for the same event, the Yield and Spin approach can result in unfairness, where some processes are repeatedly scheduled and others are not, leading to an inefficient use of resources.

13.5.3 Completion Events

Completion events provide a more efficient approach for synchronization compared to spinlocks or busy waiting. Instead of repeatedly checking for the availability of a lock or resource, a thread or process that cannot acquire the lock can choose to block and be notified later when the lock becomes available.

This mechanism is also applicable to situations where a process needs to wait for other processes or I/O operations to complete before proceeding further.

Example: What to Do?

A thread can wait for an I/O operation to finish, or another process to complete its task, without wasting CPU cycles.

These completion events are implemented using condition variables provided by the operating system, which allow threads to block efficiently and wake up only when the desired condition is met, thus avoiding unnecessary context switches and improving overall system performance.

13.5.4 Condition Variables

Condition variables allow threads to block and wait for a specific even or state change before proceeding. When the desired condition is met, another thread signals the condition variable which unblocks the waiting threads and allows them to continue execution.

Condition variables are usually provided by the OS or implemented in thread libraries. When a thread waits on a condition variable, it's blocked and removed from the ready queue. The OS or thread library then monitors the variable and unblocks the waiting thread(s) when the desired event occurs, placing it back on the ready queue (or possibly preempting the current thread).

Corollary: Multiple Waits

Threads can wait on several different things, so we want to wake threads up only when they *should*. So, the OS or thread library should allow easy selection of the correct thread(s) to wake up. Usually, we use a waiting queue to handle multiple waits.

13.5.5 Waiting Lists

Rather than spinning for a lock, we often use waiting lists, a shared data structure that keeps track of threads waiting for a particular lock. However, this implementation may have circular dependencies (which should be avoided!): since the waiting list itself is shared, we should probably put a lock on it to prevent concurrent access.

Example: Sleeping Beauty

Suppose thread B locks a resource that thread A wants to use. So, thread A will call `sleep()` to wait for the lock to be free. At the same time, thread B finishes and calls `wakeup()` to release the lock. Assuming no other threads are waiting for the resource, we have a race condition! `wakeup()` may potentially have no immediate effect (since there's no one waiting for the resource), and thread A might sleep indefinitely.

The example above illustrates an example of *deadlock*[1]. Fortunately, this is a mutual exclusion problem! `sleep()` is the *critical section* that we need to handle. We need to prevent `wakeup()` and other people from joining the waiting list.

Corollary: Fairness

Fairness in the context of mutual exclusion refers to the guarantee that all processes, threads, or machines requiring access to a resource will eventually be granted access. The goal is to prevent starvation, where some processes are consistently denied access while others frequently obtain it.

Achieving fairness in locking mechanisms can be challenging and depends on the choice of approach. For instance, using First-In-First-Out (FIFO) locks ensures that access is granted based on the order of arrival, enforcing fairness in request sequence.

Additionally, priority inversion avoidance techniques can be employed in systems with priority levels assigned to processes. Implementing yielding in spinlocks can also promote fairness by allowing other processes to make progress while a lock is held.

Advanced locking techniques like ticket locks or MCS locks inherently provide fairness properties by ensuring waiting processes are granted access in a fair manner. Although achieving perfect fairness in all scenarios might not always be feasible, considering fairness in the design of the system can help maintain equitable resource access among multiple contenders.

[1] Deadlock refers to the situation where a set of threads are unable to proceed with execution because they're waiting for a resource that is held by another process in the set. Consequently, all processes in the set wait indefinitely.

Chapter 14

Synchronization Primitives

Synchronization primitives are tools to manage the concurrent execution of multi-threaded programs. They help prevent race conditions and ensure that shared resources are accessed in a controlled and safe manner. They provide mechanisms for controlling the order of execution, ensuring data consistency, and preventing any race conditions.

14.1 Semaphores

Definition: Semaphores

Semaphores are a synchronization mechanism that controls access to shared resources and manages concurrent execution of multiple processes or threads.

While semaphores are *theoretically* well-defined, they are not always the most practical choice for real-world synchronization primitives, since there are gaps between theory and *practical* implementation; i.e. some abstract concepts of semaphores may not directly translate to intuitive solutions in real code.

Corollary: Computational Semaphores

Computational semaphores are *the* classic synchronization mechanism[1] as its behavior is well-defined and is the foundation for most synchronization studies. They are more powerful than simple locks, providing a richer set of features. For the rest of this section, when we refer to semaphores, we are referring to ***computational semaphores***.

Example: Waiting in Line

Semaphores incorporate a FIFO waiting queue, ensuring that *all* processes/threads are granted access in the order they requested it, avoiding potential issues like starvation. Additionally, unlike binary flags used in simple locks, computational semaphores use a (modifiable) counter to manage access to allow a limited number of concurrent access to a critical section.

14.1.1 Structure

A semaphore consists of two primary components: the *integer counter* and *FIFO waiting queue*. The integer counter represents the current state of the semaphore and manages access to a critical section. The FIFO waiting queue manages which order the threads are granted access to the critical section.

[1] Computational semaphores were introduced by Edsger Dijkstra in 1968.

Aside: Why Semaphores Don't Broadcast

Semaphores, in their basic form, don't provide a built-in mechanism to broadcast because they weren't designed to. They are designed to control access to a critical section by limiting the number of threads that can access the resource simultaneously rather than manage communication between multiple threads.

14.1.2 Operations

There are two operations: *Proberen* and *Verhogen*.

P (Proberen/Test)

The P operation[2] is used when a thread wants to access a critical section protected by a semaphore. Its steps are as follows:

(i) Decrement the *integer counter*.

(ii) If the resulting value is non-negative ($0 \leq$), the thread is allowed to proceed, and enters the critical section.

(iii) If the resulting value is negative (< 0), the resource is currently unavailable, so the thread is added to the waiting queue.

V (Verhogen/Raise)

The V operation[3] is used to release the semaphore and *signal* that the shared resource is now available for use. Its steps are as follows:

(i) Increment the *integer counter*.

(ii) If the waiting queue is non-empty, wake up one of the waiting threads and allow it to enter the critical section.

These operations ensure that threads interact with the shared resource in a coordinated and controlled manner. The P operation enforces mutual exclusion by allowing only one thread access at a time, while the V operation ensures that there is no resource starvation by allowing *everyone* to *eventually* access the critical section via the *FIFO waiting queue*.

14.1.3 Use Cases

Example: Exclusion

Semaphores can be used to ensure exclusion: only one thread can access a critical section of code at a time, preventing data corruption.
We can do this by doing the following:

(i) Initialize the *integer counter* to 1[a].

(ii) When a thread wants to take a lock, use *P/wait*. Here, the first *wait* will succeed (counter = 0). Any subsequent *wait*'s will block (counter = −1).

(iii) When a thread is done and wants to release the lock, use *V/signal*. Now, counter = 0, and if there are any waiting threads, unblock the first one. This thread will now take the lock.

[a] The integer counter reflects the number of threads allowed to hold the lock on a critical section. Initializing the counter to 1 ensures mutual exclusion!

[2] Also known as "wait".
[3] Also known as "signal".

Example: Notifications

Semaphores can be used to implement notifications or signaling mechanisms between threads. We can do this by doing the following:

(i) Initialize the *integer counter* to 0[a].

(ii) When a thread wants to wait for a notification (completion event), use *P/wait*. If the counter is positive (0 <), the operation will succeed immediately since a completion event has occurred, and the thread can proceed. If the counter is 0, there are no completed events, so *wait* will block.

(iii) When a thread wants to signal an event completion (notification), use *V/signal*. The counter is incremented, indicating an event has been completed. If there are waiting threads, the first thread in line will be unblocked and allowed to proceed. This thread will decrement the counter back to 0.

[a]The integer counter reflects the number of completed events that need to be signaled. Initially, there are no completed events that need to be signaled. Therefore, we initialize the counter to 0!

Example: Counting

Semaphores can be used to manage a fixed number of resources.
We can do this by doing the following:

(i) Initialize the *integer counter* to n[a].

(ii) When a thread wants to consume a resource, use *P/wait*. If the counter is positive (0 <), there are available resources, so the thread takes the resource and decrements the counter. If the counter is 0, all resources are currently consumed, so *wait* will block.

(iii) When a thread wants to release the resource, use *V/signal*. The counter is incremented to indicate that the resource it was using is now available. If there are waiting threads, the first thread in line will be unblocked and allowed to proceed. This thread will decrement the counter.

[a]The integer counter reflects the number of available resources to be managed. We initialize the counter to n since there are initially n available resources!

14.1.4 Limitations

While semaphores are a fundamental synchronization mechanism, they have certain limitations which impacts their overall usability and practicality.

(i) Too *Basic*: Semaphores are considered a basic mechanism for synchronization. They were designed to be simple tools that could be used in mathematical proofs to demonstrate the correctness of concurrent algorithms.

(ii) Limited Features: Because they were designed to be simple tools for theoretical analysis, they are ill-fitted for practical synchronization in real-world symptoms.

(iii) Deadlocks and Blocking: It is relatively easy to unintentionally create deadlocks with semaphores. Moreover, we cannot check if a lock is available without potentially blocking if it's not.

(iv) Shared Access: Semaphores don't inherently support complex synchronization scenarios such as read/write shared access[4].

(v) Recovery: If a process crashes or becomes unresponsive while holding a semaphore, the semaphore can become "wedged", where it cannot use *V/signal* to resolve itself.

[4]In read/write shared access, multiple readers can access a shared resource, but exclusive access is needed for writers.

(vi) Priority Inheritance: Semaphores don't address priority inversion issues, where lower-priority tasks can block higher priority ones.

Despite these limitations, semaphores are widely supported and used in most operating systems and concurrent programming environments. They serve as a fundamental synchronization tool due to their simplicity.

14.2 Mutexes

Definition: Mutex

A **mutex** is a synchronization mechanism used in (mostly) Unix/Linux environments to provide mutual exclusion and lock sections of code, ensuring only one thread can access a critical section at any given moment.

Mutexes are designed to lock (typically) small and critical sections of code, implying that these locks are expected to be held *briefly*[5]. They are typically used for multiple threads[6] of the same process and have low overhead and are very general.

14.2.1 Object Locking

Recall that mutexes only protect critical sections of *code*. To protect persistent *objects*, we want to lock the objects themselves. Object locks are more versatile than code-level locking since they can protect resources that persist beyond the lifetime of a program's execution (e.g. a file) and offer adjustable *granularity*[7].

Though object-level locking provides mutual exclusion, it can bottleneck performance and limit the scalability of a project (via excessive locking). Furthermore, object locks can potentially cause deadlocks. Thus, object-level locking is very specific, as we need to carefully work with the concurrency requirements and choose an appropriate granularity to ensure effective and efficient locking mechanisms.

Corollary: Advisory v. Enforced Locking

Definition: Advisory and Enforced Locking

Advisory locking[a] is a locking mechanism that is *purely suggested* by the application. It relies on developers respecting the locking protocols being used and are *non-blocking*[b]. Mutexes and `flock()`'s are examples of advisory locks.

[a] Also known as: user-level or application-level locking.

[b] Non-blocking: Advisory locks do *not* automatically block threads that attempt to access a locked resource. Thus, it is the application's responsibility to check and handle locks.

Enforced locking[a] is a locking mechanism that is *strictly enforced* by the OS or underlying system infrastructure. It is typically used when data integrity is crucial. They are done within the implementation of object methods and are guaranteed to happen[b].

[a] Also known as: mandatory or kernel-level locking.

[b] Blocking: Enforced locks will automatically block threads that attempt to access a locked resource, enforcing mutual exclusion.

We will take a look at an example of an *advisory* lock and an *enforced*[8] lock.

[5] Locks are expected to be held briefly *relative* to the program.

[6] Mutexes are *assumed* to be used for threads operating on shared data; i.e. all threads are operating in a single address space. This implies that mutexes (protecting *code*) won't work for multi-process programs

[7] Granularity: We may want to either lock an entire object at a time or lock specific methods or sections *within* the object.

[8] Whether or not `lockf` is enforced depends on the underlying file system.

Example: A `flock` of Seagulls

Let's take a look at Linux's file descriptor locking: `int flock(fd, operation)`. Supported operations include:

(i) `LOCK_SH`(ared): Place a *shared* lock on `fd`. More than one process may hold a shared lock for a given `fd` at a given time.

(ii) `LOCK_EX`(clusive): Place an *exclusive* lock. Here, only one process may hold an exclusive lock for a given `fd` at a given time.

(iii) `LOCK_UN`(nlock): Remove an existing lock held by the calling process.

This lock applies to *open* instances of the same `fd`. Note that *distinct* opens are *not* affected. Moreover, these locks are *advisory* and not strictly enforced.

Example: A `lockf` of eagullsS

Let's look at the Linux ranged file lock: `int lockf(fd, cmd, offset, len)`. Supported commands include:

(i) `F_LOCK`: Set an *exclusive* lock on the specified section of the file. If (part of) this section is already locked, the call *blocks* until the previous lock is released. If this section overlaps an earlier locked section, both are merged. File locks are released as soon as the process holding the locks closes some `fd` for the file. Note that a child process does *not* inherit these locks.

(ii) `F_TLOCK`: Same as *(i)*, but the call *never* blocks. Instead, we return an error if the file is already locked.

(iii) `F_ULOCK`: Unlock the indicated section of the file. This may split a single locked section into two locked sections.

(iv) `F_TEST`: Test the lock: returns 0 if the specified section is un/locked by this process; returns -1 and sets `errno = EAGAIN/EACCES` if another process holds a lock.

This lock applies to the *file*, not its open instance. Thus, this lock is process specific and closing *any* `fd` for the file releases for *all* of the process' `fd`'s for that file. Depending on the underlying system, these locks are *enforced*.

14.3 Locking Problems

We will talk about two main problems related to locking: performance/overhead and contention.

14.3.1 Performance and Overhead

Locking mechanisms are usually implemented as syscalls, incurring traditional syscall overheads[9]. When locking operations are called frequently (e.g. a *heavily* threaded application, the cumulative overhead of the syscalls may become significant enough to notice, leading to contention and degraded performance, reducing overall efficiency.

Enforced locks often incur more overhead than advisory locks since the OS needs to manage and coordinate access to the locked resource among multiple threads/processes. However, even if locking isn't enforced or is implemented outside of the OS, we still need to execute extra instructions to lock and unlock.

[9] If locking operations are performed frequently, we may incur high overhead since it is essentially calling syscalls frequently.

The granularity of locks can also impact performance. *Fine-grained locks*[10] can reduce contention but may incur overhead, since we un/lock more frequently. *Course-grained locks*[11] can reduce overhead but may result in more contention among threads/processes.

Unfortunately, locking code in operating systems have already been highly optimized, thus there is not much more that can be done.

Aside: Locking Costs

Locking is typically used when we need to protect critical sections to ensure correctness. Since many critical sections are very brief (e.g. in/out in nano-seconds), the overhead of the locking operations may be much higher than the time spent in the critical section. This is why we care about locking overheads!

14.3.2 Contention

When a thread doesn't get a lock, we block! However, blocking introduces significant overhead compared to simply acquiring a lock, and the cost can vary depending on the likelihood of contention.

Definition: Contention

Contention refers to when multiple threads/processes are competing for access to a shared resource.

The expected cost of acquiring a lock, taking into account the probability of contention, is defined as:

$$C_{expected} = (C_{block} \cdot P_{conflict}) + (C_{get} \cdot (1 - P_{conflict}))$$

where

(i) $C_{expected}$ is the expected cost of acquiring a lock.

(ii) C_{block} is the cost associated with a thread being blocked due to contention. This include the overhead of context switches, potential queuing, and any other blocking costs.

(iii) C_{get} is the cost associated with successfully acquiring the lock without contention. Typically, this cost is lower than *(ii)*.

(iv) $P_{conflict}$ is the probability of a contention occurring.

$(C_{block} \cdot P_{conflict})$ represents the cost when there is contention, while $(C_{get} \cdot (1 - P_{conflict}))$ represents the cost when there is no contention (i.e. acquires the lock without blocking). Considering both scenarios and their costs, this formula provides an estimation of the overall expected cost of acquiring the lock.

14.3.3 Reducing Contention

There are many solutions to reduce contention. Here, we will cover the following:

(i) Eliminate the critical section entirely.

(ii) Eliminate preemption during critical sections.

(iii) Reduce the time spent in critical sections.

(iv) Reduce the frequency of entering critical sections.

(v) Reduce *exclusive* use of serialized resources.

(vi) Spread requests out over more resources.

[10] Fine-grained: Locks that protect small sections of code.
[11] Course-grained: Locks that protect larger sections of code.

Eliminating Critical Sections

We can solve the contention problem by simply eliminating the source of contention: critical sections. In this approach, we eliminate shared resources entirely, giving everyone their own copy. Additionally, we use atomic instructions wherever possible. This approach is great when feasible, but we often cannot simply avoid critical sections.

Eliminating Preemption in Critical Sections

In this approach, we do not allow preemption when inside of a critical section. If critical sections cannot be preempted, we eliminate synchronization problems! However, this usually involves disabling interrupts and is not always viable.

Reducing Time Spent in Critical Sections

Here, we move potentially blocking operations[12] outside of critical sections and minimize the code inside the critical section. We want to include only code that is subject to destructive races. Unfortunately, while this approach is intuitive, it may complicate the code, unnaturally separating parts of a single operation.

Reducing the Frequency of Entering Critical Sections

This approach involves minimizing the number of times threads/processes need to access shared resources or critical sections. We achieve this by simply using high-contention resources/operations less often by either optimizing our algorithms to reduce the reliance on them or simply use them less. Additionally, we can perform batch operations rather than executing individual operations one by one.

In some scenarios, we can employ *sloppy counters*. Here, each thread maintains and updates a private counter as opposed to the shared global counter. This however implies that the global counter is not always up-to-date. Additionally, thread failure can lose updates if it hasn't written to the global. Alternatively we can sum single-writer private counters when we need to access the global counter.

Remove Exclusivity Requirements

We want to reduce the amount of resources that require *exclusive* access. For example, in read/write locks, only *writers* require exclusive access. Thus, we allow many *readers* to access the shared resource, only enforcing exclusivity when *writing*[13].

Spread Requests Over More Resources

This approach changes the lock granularity (See **14.3.1 Performance and Overhead**) to reduce contention. Coarse-grained locks are simpler and more idiot-proof, but increase resource contention. Fine-grained locks spread activity over many locks to reduce contention, but increase overhead.

14.3.4 The Convoy Effect

Definition: Convoy Effect

The **convoy effect** refers to the scenario where multiple threads/processes are blocked while waiting for access to a single shared resource. These threads/processes form a queue or *convoy*, with each one waiting for the resource to become available.

While the convoy is waiting for the resource, no other work can be done. This means that parallelism is effectively eliminated during the time the convoy is waiting, since even if there are other unrelated tasks that *could* run in parallel, they are delayed due to the resource contention. Since processes are waiting in line, they are *forced* to execute in a sequential manner. Process $i+1$ can only proceed after

[12] Potentially blocking operations include (but are not limited to): allocating memory, I/O, etc.

[13] This requires a new policy: how do we determine when writers are allowed in? A bad policy can lead to potential starvation if writers must wait for readers.

process i releases the resource. This causes the resource to be a bottleneck in the system; that is, the overall system performance is limited by the speed at which the resource can be accessed and released. Thus, the system's throughput and efficiency can be severely compromised.

In a FIFO queue formation[14], the formula to estimate the probability of contention is defined as

$$P_{conflict} = 1 - \left(1 - \frac{T_{wait} + T_{critical}}{T_{total}}\right)^{threads}$$

where

(i) $P_{conflict}$ is the probability of contention.

(ii) T_{wait} is the time spent waiting for the resource.

(iii) $T_{critical}$ is the time spent actively using the resource in a critical section, where it has exclusive access.

(iv) T_{total} is the total time a thread takes to complete its execution, including both the time spent waiting(*(ii)*) and the time spent using the resource (*(iii)*).

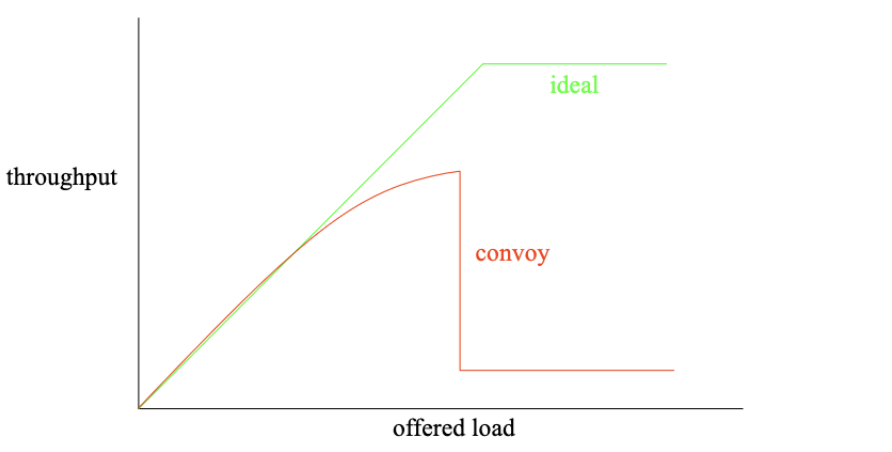
(v) *threads* is the number of threads.

In the general case[15], we get

$$P_{conflict} = 1 - \left(1 - \frac{T_{critical}}{T_{total}}\right)^{threads}$$

In both cases, as contention increases, the probability of contention increases. If T_{wait} becomes *long enough*[16], newcomers joining the queu can make the wait time even longer, potentially ceasing parallelism.

Example: The Convoy Effect



Here we see the graph illustrating the convoy effect. The inflection point of the convoy line represents the system bottlenecking due to a high T_{wait}.

[14]FIFO formation happens when threads wait in line to access the resource.
[15]The general case assumes there is no FIFO queue formation.
[16]T_{wait} reaches teh mean inter-arrival time.

14.4 Priority Inversion

Definition: Priority Inversion

Priority inversion refers to a situation in priority-based scheduling systems using locks where a higher-priority process is blocked and delayed by a lower-priority process that currently has a lock on the desired resource. This effectively reduces the higher-priority process to the lower-priority process.

Priority inversion can lead to reduced performance, since the program may be bottlenecked by the lowest-priority process. This leads to an inefficient utilization of resources and increases the risk of deadlock, especially with intermediate priorities[17]. Moreover, the program becomes unpredictable and has the potential to cause system crashes.

Example: Get on my Level

Suppose we have two processes, L (low-priority) and H (high-priority) and a mutex M. Say L is holding M and is preempted since H has higher priority and is ready to run. However, H blocks for M since L is currently holding it! This reduces H's priority to L's priority.

14.4.1 Priority Inheritance

A common solution to the priority inversion problem is *priority inheritance*. It involves temporarily raising the priority of the lower-priority task, L, (holding the lock) to that of the highest-priority task waiting for the locked resource, H. This prevents L from being preempted by a higher priority task waiting on the locked resource. Once the resource is released, the priority of L is reduced to its original value.

14.5 Deadlocks

Definition: Deadlock

A **deadlock** is a situation where two entities are unable to proceed with execution because they are each waiting for the resource that the other entity holds.

Understanding deadlocks are important for a myriad of reasons. They pose a major risk in complex applications since they can lead to system failures. Moreover, they are usually hard to detect and their occurrence is usually nondeterministic. Hence, it is usually commonplace to prevent the possibility of deadlocks at design time.

Example: The Dining Philosophers Problem

Suppose there are 5 philosophers at a table along with 5 plates of pasta and 5 forks. Each philosopher needs two forks to eat pasta, but must pick them up one at a time. Assume that the philosophers will not negotiate with one another and they may try to eat at any given moment. Given these constraints, we see a potential deadlock.
If all 5 philosophers attempt to eat at the same time, each grabs one fork. Since the philosophers do not negotiate with each other, they wait on another philosopher to relinquish a fork. However, since all 5 philosophers are waiting, we see that none of the philosophers will eat, leading to deadlock!

Deadlocks are usually hard to detect since process' resource needs are constantly changing, as they depend on what data they are operating on, where in computation they are, and what errors have

[17] Intermediate priorities may lead to a cyclic dependency which will cause a deadlock scenario.

occurred. Additionally, modern software usually relies on many *independent*[18] *services*[19] that each requires a variety of resources.

14.5.1 Resource Types

Definition: Commodity and General Resources

Commodity resources are ones that are given to processes in *quantities*. They are usually shareable among processes and can be divided/allocated as needed.

General (serially reusable) resources are ones where processes require exclusive access to a particular instance.

Example: Commitment Issues (Over-commitment)

Suppose you have a system with 8 GB of physical memory and a Virtual Machine that allocates a total of 16 GB of virtual memory to 4 *VM's*[a]. So, each VM thinks it has 4 GB of memory. Now assume that all VM's start using their allocated memory extensively. As they begin to exceed the amount of physical memory available, potential deadlocks can arise if the resource manager is not careful.

[a] In this example, VM will refer to Virtual Machines.

Example: Cyclic Dependencies

Suppose there are two processes A and B and two semaphores X and Y. Now, assume A acquires X and B acquires Y. A deadlock happens when:

(i) A tries to acquire Y but is blocked since it's being held by B.

(ii) B tries to acquire X but is blocked since it's being held by A.

So, we have that A and B are both waiting on each other, forming a cyclic dependence. These types of deadlocks are usually prevented at design time.

14.5.2 Deadlock Conditions

For a deadlock to occur, *all* four conditions *must* be met:

(i) Mutual exclusion

(ii) Incremental allocation

(iii) No preemption

(iv) Circular waiting

Condition I: Mutual Exclusion

The resource in question *must* be one that requires exclusive access; i.e. once a resource has been given to a process, *all* other requesting processes *must* wait. Otherwise, we wouldn't have deadlock since we can just give an instance of the resource to all the requesting processes.

[18] Independent: Services need not be aware of others' existence.

[19] Services encapsulate a *lot* of complexity, and as such, we don't know what resources they require nor do we know when/how they are serialized.

Condition II: Incremental Allocation

Entities are allowed to ask for resources at any time rather than requesting them before execution. Otherwise, they either get everything they need and run to completion or don't get everything they need and abort execution. In either case, we don't get deadlock.

Condition III: No Preemption

When an entity has access to a resource, we cannot preempt it; i.e. once an entity has access to a resource, we must wait for it to free the resource. Otherwise, we can resolve all deadlocks by preempting the offending process(es).

Condition IV: Circular Waiting

We *must* have a situation that causes a circular wait; i.e. we have a cycle in a graph of resource requests. Otherwise, there is no cycle, so *someone* can complete without anyone releasing a resource. This allows even a long chain of dependencies to *eventually*[20] unwind.

14.5.3 Avoiding Deadlock: The Reservation System

Note how we said in the previous section that *all* four conditions must be met for a deadlock to arise. However, that is much easier said than done. We can, however, use methods that guarantee that no deadlock can occur by nature. We introduce the notion of *reservations*.

In the reservation system, the resource manager tracks outstanding reservations requested by processes. Rather than fulfilling any and all requests, we only honor reservations if and only if the resources are truly available. This prevents situations where too many tasks expect resources that aren't there (over-commitment), detecting over-subscriptions early.

In the reservation system, we may run into resource starvation. Because reservations are only honored if the resource is available, some processes may never receive the resources they reserved. While this leads to resource scarcity, it does *not* lead to a deadlock! This allows us to handle this scarcity in a *controlled* way.

We clearly face a dilemma: should the resource manager over-commit or under-utilize? Striking a balance is key to maintain optimal resource utilization without risking critical resource shortages.

Handling Reservations

Entities seldom require *all* available resources at all times. As a corollary, we extrapolate that all clients don't need their maximum allocated resources simultaneously. This raises the question: Can we *safely* over-book resources.

Example: Fly Fucked

Airplanes often over-book on every flight, banking on people not showing up. Fortunately for them, they are often correct! Can we apply the same concept to operating systems?

Let's define what a safe allocation is:

Definition: Safe Allocation

A **safe allocation** is one where everyone can *eventually*[a] complete their tasks. That is, we ensure that no deadlocks!

[a] Again, we only promise that all entities eventually complete, not that they complete in a swift manner.

[20] We promise that it eventually unwinds, not that it unwinds quickly.

Corollary: Commodity Resource Management in Real Systems

Many real-world systems use advanced reservations to manage resource efficiently. In these systems, one a reservation is accepted, the system is obligated to honor it. This way, we prevent resource starvation since allocation failures only occur at the time of reservation, usually before the start of execution. This makes system behavior more predictable and easier to handle. This also forces the client to deal with the reservation failure, meaning we don't have to!

14.5.4 Reservation Failures

While reservations eliminate deadlock, applications still need to deal with reservation failures. We should design applications to handle failures *gracefully*[21]. Additionally, we need to be able to communicate to the requester that a reservation failed (e.g. error messages or return codes). Finally, the application *must* be able to continue running. Naturally, this forces all critical resources to be reserved prior to execution.

While rejecting requests are not ideal, they are better than system failure later down the road. This way, the advanced notice allows applications to adjust their services to work without the unavailable resource. If the application is in the process of fulfilling a request and the resource becomes unavailable, complications arise. At this point, if a failure occurs, we need to roll back the actions that have already taken. This process can be complex or even impossible depending on the situation.

Example: Reservation Rejected!

Consider a web server that relies on a database. If the database becomes temporarily inaccessible (e.g. maintenance or network issues), we can adjust our services to display a cached version of the web page or show an error message instead of crashing entirely! We can do this by implementing mechanisms like graceful degradation. Let's look at an example of a situation where a reservation is rejected in the middle of fulfilling a request. Suppose we have a shopping application is processing a customer's order, and the payment gateway becomes inaccessible between deducting funds and confirming the order. Here, we need to refund the money which would be a nontrivial task.

14.5.5 Avoiding Deadlock: Additional Measures

Deadlock prevention focuses on ensuring that a specific lock doesn't lead to deadlock by addressing one of the four deadlock conditions (See **14.5.2 Deadlock Conditions**). Remember, if *any* of the four conditions are not met, we don't have deadlock! Let's take a look at how we can prevent deadlock by failing to satisfy each of the four conditions.

Condition I: Mutual Exclusion

We can prevent deadlock by utilizing shared and private resources. Recall that deadlocks cannot occur over shareable resources, which are typically managed using atomic instructions. Like shareable resources, deadlock cannot occur over private resources since they are owned by individual processes!

Condition II: Incremental Allocation

To combat deadlock, we can allocate all our resources in a single operation as well as use non-blocking requests. Here, if we cannot allocate everything, we return failure and don't lock, proceeding otherwise. Note that in this approach, we get an all or nothing allocation. Non-blocking requests will fail if the request cannot be satisfied immediately. Finally, we can disallow blocking while holding resources. That is, we must release all held locks before blocking, reacquiring them *after* you return[22].

[21] For example, we can refuse to perform a new request, but continue running.

[22] It is important to note that while this approach may solve the deadlock problem, it can introduce new ones. Remember, CS is all about give and take!

Condition III: No Preemption

We can implement resource confiscation to combat deadlocks. Here, we seize and reallocate resources from existing processes. We achieve this via resource "leases", with time-outs and "lock breaking"[23]. This approach requires us to enforce resource revocation. We can do this by either invalidating resource ownership *(i)* or terminating the previous owner *(ii)*. To achieve *(i)*, when a resource is taken away from a process, the ownership of that resource is invalidated. Thus, the process loses access to the resource and prevents it from attempting to use a confiscated resource! To achieve *(ii)*, we simply terminate the previous owner. This option is usually a last resort and is only used when resource invalidation is not possible[24].

Aside: Seizing Resources

The operating system can only seize a resource if the process has to use a system service to access the resource. If the process has direct access to the object (e.g. the object is part of the process' address space), then we're fucked and usually have to resort to terminating the process to free the resource.

Condition IV: Circular Waiting

We can enforce *total resource ordering*. Here, all processes requesting resources follow the same order when allocating them. Given two resources R_1 and R_2, *all* processes are *required* to allocate R_1 before attempting to allocate R_2. This way, we prevent circular dependencies by creating a DAG[25]! To implement total resource ordering, we need to order the resources! We can do this by ordering them by resource type (e.g. groups before members) or by relationship (e.g. parents before children).

Aside: The Lock Dance

Suppose two process (P_1 and P_2) needs to allocate resources (R_1 and R_2) out of order. Initially, we have $P_1 : R_1$ and $P_2 : R_2$, but our desired state is $P_1 : R_2$ and $P_2 : R_1$. Let's do the lock dance!

(i) P_1 releases R_1 ($P_1 : \emptyset$, $P_2 : R_2$)

(ii) P_2 releases R_2 ($P_1 : \emptyset$, $P_2 : \emptyset$)

(iii) P_2 acquires R_1 ($P_1 : R_2$, $P_2 : \emptyset$)

(iv) P_1 acquires R_2 ($P_1 : R_2$, $P_2 : R_1$)

Conclusion

There is no universal solution to all deadlocks. Determining which solution to use is on a case-by-case basis, and fortunately for us, we only need to implement one of the four!

14.5.6 A Deadlock "Solution"

We can also elect to simply ignore the problem. In many cases, deadlocks are *very* improbable and implementing any of the solutions provided above can be very expensive. So, we can just forget about them and pray to the CS gods that we don't run into a deadlock.

Deadlock Detection and Recovery

We can allow deadlocks to occur, but we want to be able to detect them once they occur (the sooner the better). Once we detect them, we need a method of breaking them to continue execution. Whether or not this is a good idea depends.

In general, it's probably a more practical approach since the overhead and complexity of deadlock-proofing your system may be too expensive. However, in critical systems where resource availability is

[23] Lock breaking: When a process' "lease" is up, the resource is automatically taken away and returned to the resource pool.

[24] Like Condition II, lock breaking may fix deadlock but can damage resources!

[25] DAG: Directed Acyclic Graph

crucial (e.g. real time systems), we might want to invest the time and energy into preventing deadlocks, since they can lead to severe disruptions, data loss, or even safety hazards.

Implementing Detection

We can implement deadlock detection by using a wait-for graph or an equivalent data structure. When a lock request is made, the system updates the wait-for graph and checks for the presence of a deadlock (cycle detection).

Whether or not it's better to reject a lock request (that will lead to deadlock) and not let the requester block depends. If we reject a lock request, there's no deadlock, but it might result in process disruptions and inefficiencies. If we allow the requester to block, we may get deadlock, but the system will be more resource-efficient overall.

Implementing detection may be challenging however, since we need to identify *all* resources that can be locked.

Application Level Deadlocking

Some applications include their own internal locking mechanisms independent of the operating system. Since the OS doesn't know about these locks, it cannot offer any help. Here, deadlock detection may be appropriate since the application itself has the necessary information to identify and handle deadlocks.

14.6 Health Monitoring

Not everything is a deadlock! There are a lot of reasons systems hang and make no progress. Occasionally, it really is a deadlock. Other times, it's something else (e.g. live-lock, lock flaws, etc.). If there are no locks, it's definitely not deadlock. Even if there are, it may not necessarily be deadlock.

We can use service/application *health monitoring* in lieu of deadlock detection. Here, we monitor application progress and submit test transactions. If responses take *too long*[26]. Health monitoring is much easier to implement than deadlock detection and can detect a wider range of problems, including (but not limited to):

(i) Live-lock: A process is running but unable to proceed due to unmet dependencies.

(ii) "Sleeping Beauty": A process waits indefinitely for an event(s) that will never happen.

(iii) Priority inversions: Health monitoring can detect when priority inversions cause system hangups!

14.6.1 Monitoring Process Health

We can use a variety of metrics to determine process health. We will cover:

(i) Obvious Failures

(ii) Passive Observation

(iii) External Monitoring

(iv) Internal Instrumentation

Obvious Failures

We can actively check for obvious failures like abnormal process exits or core dumps. These indications can be monitored and analyzed to identify the cause of failure.

Passive Observation

We can identify processes that are unresponsive or hanging by monitoring CPU usage, blocking status, and network/disk I/O. If a process is not consuming CPU time or blocked for extended periods of time, it may be stuck. Likewise, if a process is not performing expected I/O operations or engaging in network activities, it may potentially be stuck.

[26] Too long is subjective and is dependent on the system requirements.

External Monitoring

We can interact with processes to asses their responsiveness. Common techniques are pings, null requests, and standard test requests. We can ping the process to measure response time. If the process fails to respond within a given time frame, it may be unresponsive. Like pings, we can send a null/dummy request expecting acknowledgment by the process. Lack of acknowledgment can indicate unresponsiveness. More specifically, standardized test requests expect specific responses from the process. Failure to provide the expected response may trigger alerts.

Internal Instrumentation

We can embed monitoring and testing directly within the codebase of the process via white box audits, exercisers, and continuous monitoring. White box audits involve reviewing internal code adn data structures to identify potential vulnerabilities, inefficiencies, or errors. Exercisers are designed to simulate various scenarios to stress-test the process, which can help uncover unexpected behavior and/or vulnerabilities, particularly in complex systems. Continuous monitoring keeps track of key performance metrics and states within the process. Monitoring tools collect data and trigger alerts when the numbers deviate from the expected behavior.

14.6.2 Unhealthy Processes

When processes are unhealthy, killing and restarting any and all affected software is a common and easy recovery strategy. However, deciding which processes to kill and restart requires a balance between addressing the issue and minimizing disruptions.

It is important to note that unresponsive processes may not necessarily be the cause of the problem. Thus, we need to take into consideration how kills and restarts can impact current clients which are dependent on service API's and protocols. Applications must be designed to handle different types of restarts: cold, warm, or partial. In highly available systems, restart groups are well-defined and specify the groups of processes to be restarted together as well as inter-group dependencies.

14.6.3 Failure Recovery

Definition: Cold Restart

A **cold restart** involves terminating the entire process and restarting it from scratch[a]

[a] This ensures a clean slate but may incur longer downtime.

Definition: Warm Restart

A **warm restart** involves restarting the process but maintaining existing connections/sessions[a]

[a] This minimizes disruption but may not solve the problem.

Definition: Partial Restart

A **partial restart** involves restarting only specific components of the process[a].

The failure recovery methodology involves retries, rollbacks, and continued functionality. First, we retry a request (if possible), but set limits on how many/how long we wait. If a request ultimately fails, we want to roll back failed operations to before the request and return an error. Finally, we continue with reduced capacity/functionality, accepting only requests we can handle, rejecting otherwise. Additionally, we implement automatic restarts (cold, warm, partial) and use escalation mechanisms to address failed recovery attempts.

Interlude: Making Synchronization Easier

Locks, semaphores, and mutexes are hard to use correctly, since they may not be used when they're needed, may be used incorrectly, and may lead to deadlock or other hanging situations.

One way we can make synchronization easier is to automate the generation of serialization mechanisms. Here, shared resources (objects with methods that require serialization) are identified. Code is written to operate on shared resources without *explicitly* adding synchronization. The compiler then generates the required locking and releasing mechanisms to ensure proper serialization.

14.7 Monitors

Definition: Monitors

Monitors are objects with built-in synchronization mechanisms.

We introduce the concept of monitors as protected classes. Here, each monitor object is associated with a mutex that is automatically acquired when a method is invoked on the monitor and released upon method return. Monitors provide encapsulation, allowing developers to manually avoid identifying critical sections, and clients are relieved of the responsibility of managing locks.

14.7.1 Simplicity v. Performance

Monitors provide simplicity in synchronization by automatically locking the *entire* object during method invocations. Since objects are locked for the entire duration of any method invocation, it is a conservative approach to synchronization and can potentially lead to performance issues. Locking the entire object can eliminate parallelism and create the potential for thread contention, resulting in convoys (See **14.3.4 The Convoy Effect**). When implementing synchronization primitives, we consider the trade-offs of simplicity v. performance. Fine-grained locking is difficult and error prone, but often performs well while coarse-grained locking may be simpler to implement, but can create bottlenecks.

14.7.2 Java's Synchronized Methods

In Java synchronized methods, each object has an associated mutex and is acquired for specified synchronized methods. Here, not *all* methods need to be synchronized; synchronization is needed only for the methods with access to shared resources. The mutex is automatically released upon the *final* return of the synchronized method.

Here, nested calls by the same thread do not reacquire the mutex, avoiding unnecessary locking and unlocking. Static synchronized methods lock the class-level mutex, affecting all instances of the class. Advantages include finer lock granularity and a reduced risk of deadlocks, but the cost usually involves the responsibility of developers to *correctly* identify which methods need synchronization.

Chapter 15

Device Drivers

15.1 Overview

This chapter covers the OS's role in properly handling devices via device drivers.

Definition: Peripheral Device

A **peripheral device** is an external device that is attached to the computer to perform specific functions.

Each peripheral device has code associated with it for performing operations and integration into the system. In modern commodity operating systems, the amount of code dedicated to handling devices is substantial and can outweigh other parts of the system's codebase!

Peripherals are usually connected to the computer via a bus, which facilitates communication between peripherals and other system components. They are designed to perform specific (usually predefined) functions rather than independently executing arbitrary computations. The system sends a signal on the bus to instruct the peripheral to perform its designated task, which is then performed asynchronously alongside other system activities. Once finished, the signal is sent back from the peripheral to the system on the bus to communicate the results of the operation.

15.2 Devices and Performance

Compared to the CPU, bus, and RAM, most peripheral devices are *slow*[1]. Consequently, managing devices *efficiently* presents performance challenges. The system operates at the speed of the CPU, bus, and RAM, but correct behavior often requires interactions with *slow*[1] devices.

[1] Sometimes several orders of magnitude slower than the CPU, bus, or RAM.