Computer Systems

IN BA4

Notes by Ali EL AZDI

September 11, 2024

Introduction

This document is designed to offer a LaTeX-styled overview of the Computer Systems course, emphasizing brevity and clarity. Should there be any inaccuracies or areas for improvement, please reach out at ali.elazdi@epfl.ch for corrections. For the latest version of the PDF, you can check the following link: `https://elazdi-al.github.io/compsys/index.html`. Feel free to send a pull request to propose any changes you think might be a useful addition to the course content or a modification.
`https://github.com/elazdi-al/compsys/blob/main/main.pdf`

Contents

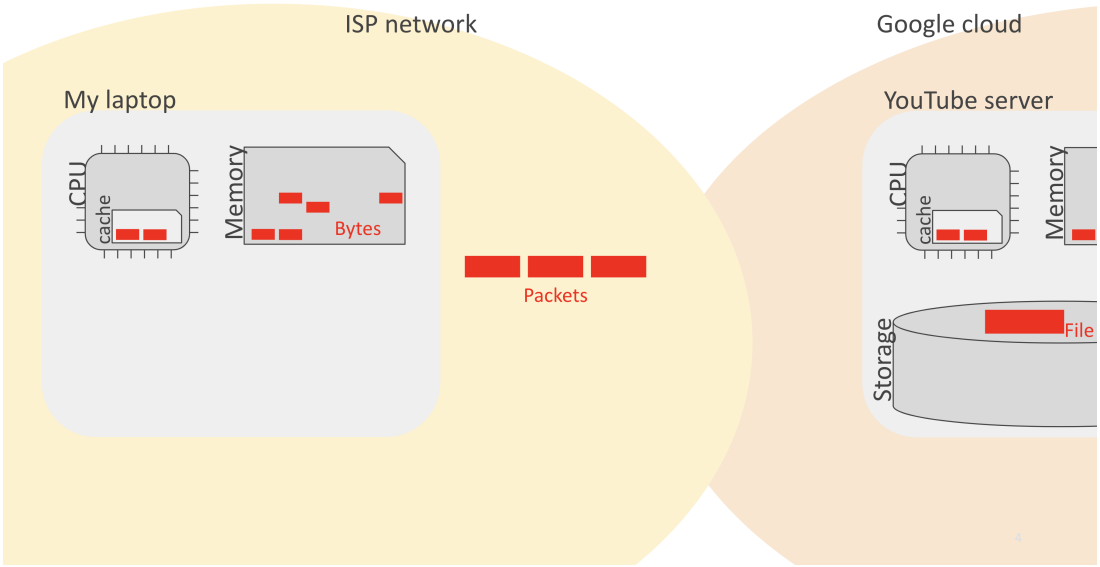
Chapter 1

Lecture 01: Introduction

In this lecture we explore the journey of a YouTube video—from its storage as a file to its transformation into bytes, its transmission over networks, and finally, its display on your device. We will introduce key concepts such as processes, threads, distributed applications, system calls, and the role of the operating system in managing hardware resources.

1.1 The Journey of a YouTube Video

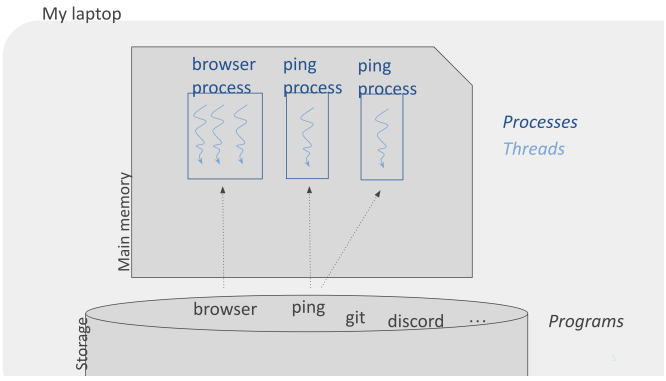
To illustrate these ideas, consider the journey of a YouTube video. The video begins its existence as a file stored on a storage device, is loaded into memory as bytes, transmitted as packets over the Internet, and finally rendered on your screen.



1.1.1 Start of the Journey: Inside the Laptop

The journey begins on your laptop.

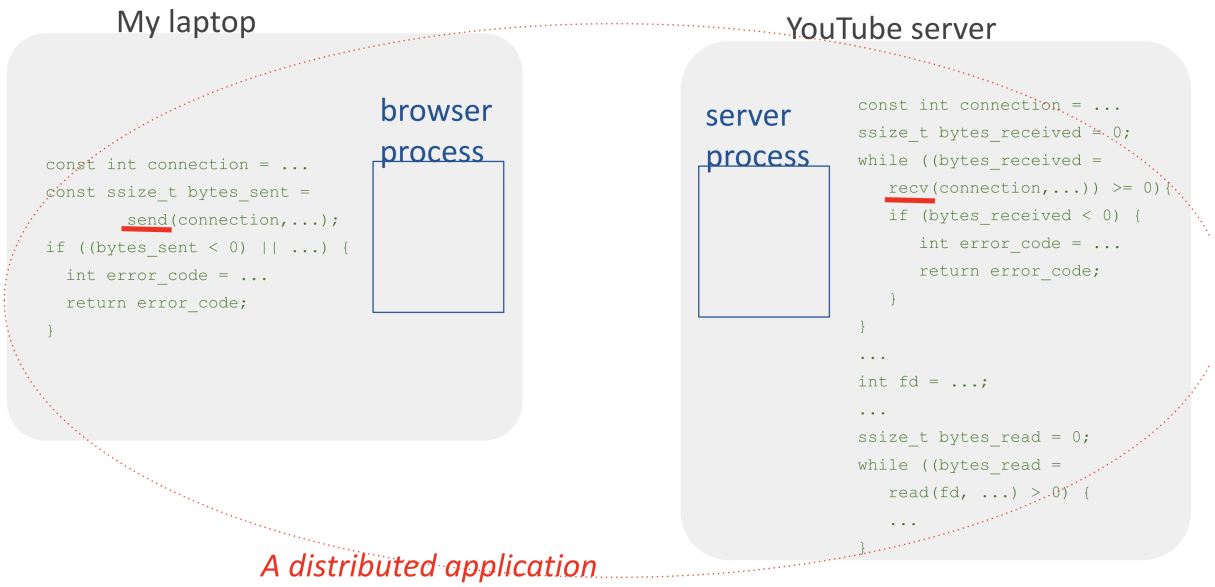
A computer hosts many different programs (e.g., a web browser, a ping utility, a git client). These programs, stored as files on disk, are invoked by user actions such as clicking an icon or typing a command. When a program is invoked, the computer creates a new *process* in main memory. A process represents a running instance of a program and may consist of one or more *threads*—individual units of execution within the process.



Definition (Program, Process, Thread). *A **program** is a set of instructions stored as a file on disk. When a program is invoked, the computer creates a **process**—a running instance of that program in main memory. A process may consist of one or more **threads**, which are the individual sequences of execution within the process.*

1.1.2 Accessing a Video: A Distributed Application

When you use your web browser to access a video, the browser sends a message (or request) to a remote YouTube server. The browser process (running on your laptop) and the server process (running on a different computer) work together as parts of a *distributed application*.

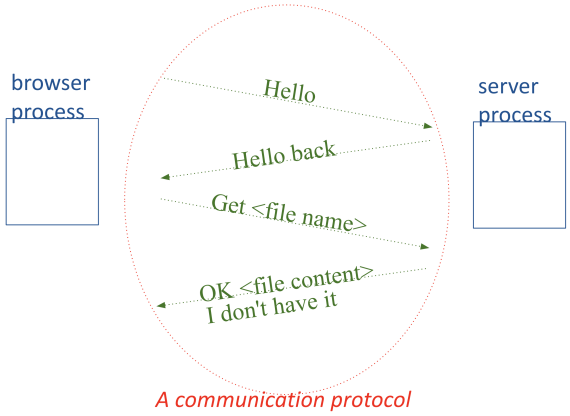


1.1.3 Communication Protocols

For two processes running on different devices to work together, they must follow a predetermined set of rules known as a **communication protocol**. For example, a simple protocol might involve:

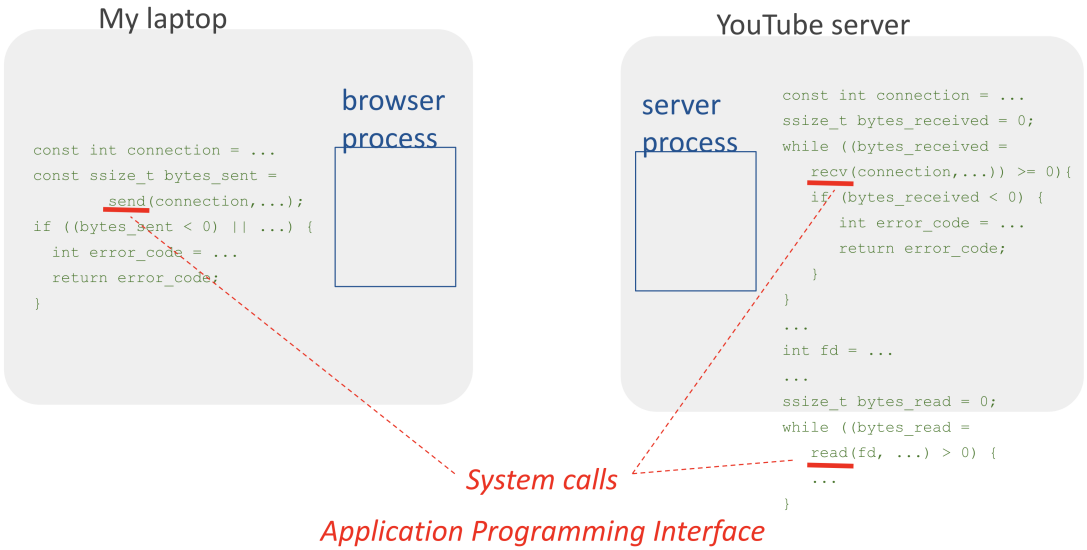
- One process sending "hello" and waiting for a "hello back."
- A subsequent request for a specific file (e.g., `xyz`) with the server responding with the file or an error message.

Much like human communication, these protocols ensure that both parties know what to expect, enabling effective interaction.



1.1.4 Distributed Applications and APIs

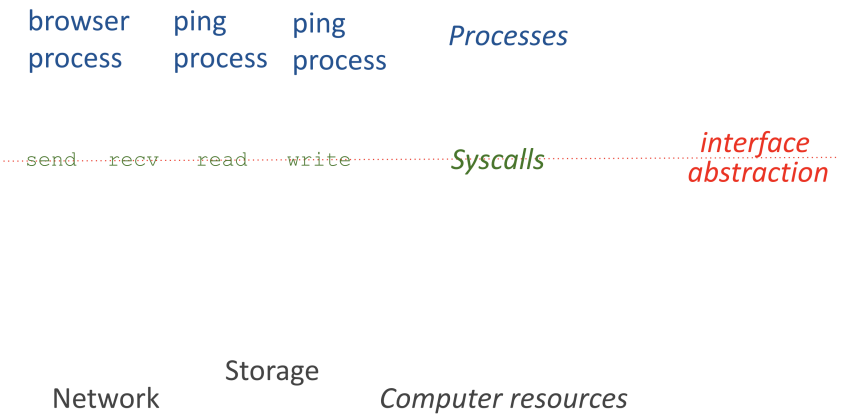
Distributed applications consist of separate pieces of code running as processes on different machines but working toward a common goal. These processes exchange messages over the Internet by following communication protocols. To simplify the development of these applications, developers use *system calls* (or **syscalls**). Syscalls are special functions provided by the operating system that allow processes to access resources (e.g., network and storage) without needing to know the low-level details.
The set of syscalls available to an application forms its **Application Programming Interface** (API), abstracting away the complexities of resource management.



Definition (Interface). *An **interface** is a set of rules that defines how different components communicate. For instance, when sending a letter via the postal system, one must follow specific rules (e.g., write the address and affix a stamp). This interface abstracts the complexities of the postal system so that users do not need to understand its internal operations.*

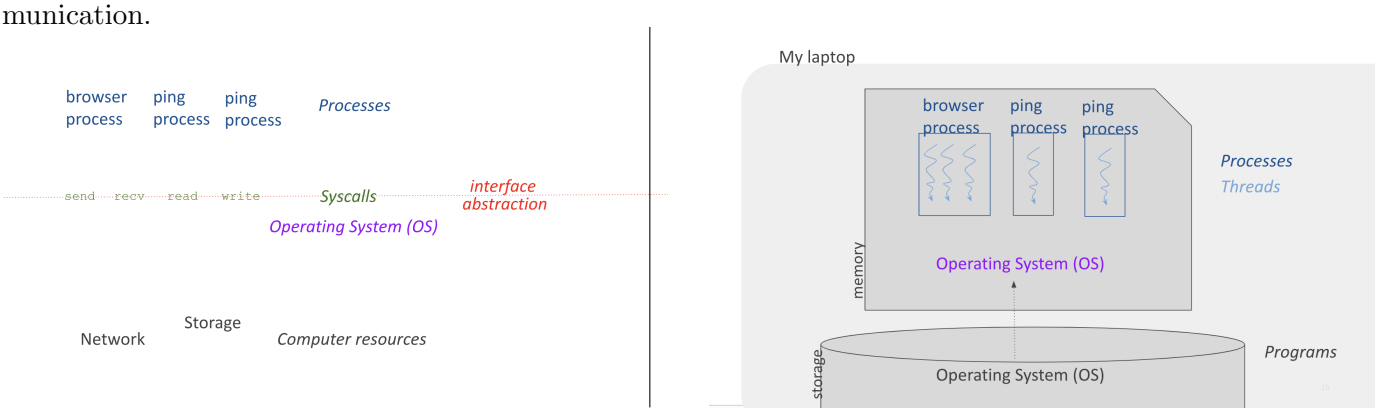
1.1.5 System Calls (Syscalls)

Syscalls form the interface between a process and external resources (like network and storage). They provide an abstraction of these resources, allowing a process to use them without knowing their intricate details. For example, when a process makes a syscall such as `send` or `recv`, the operating system's network stack handles the details of the communication.



1.2 The Operating System

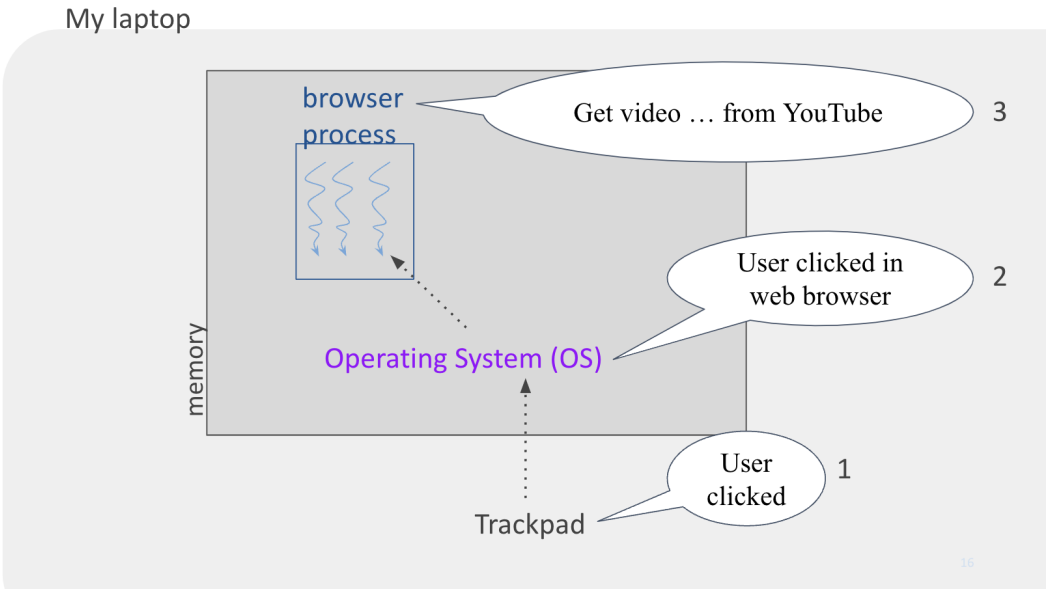
Conceptually, the OS sits between running processes and the underlying hardware resources. It provides the syscall interface and handles tasks such as file system management and network communication.



1.2.1 Example: Execution When Fetching a Video from YouTube

When you click a YouTube link, the following sequence of events occurs:

1. Your trackpad detects the click and notifies the OS.
2. The OS identifies that the click occurred within the web browser window and alerts the corresponding process.
3. The browser process initiates a chain of events that eventually fetches and displays the video.

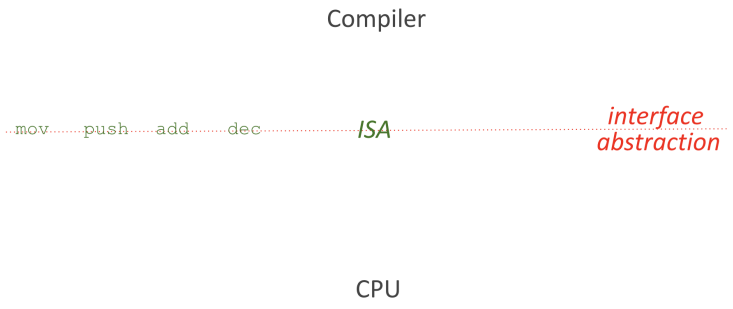


1.3 Program and ISA

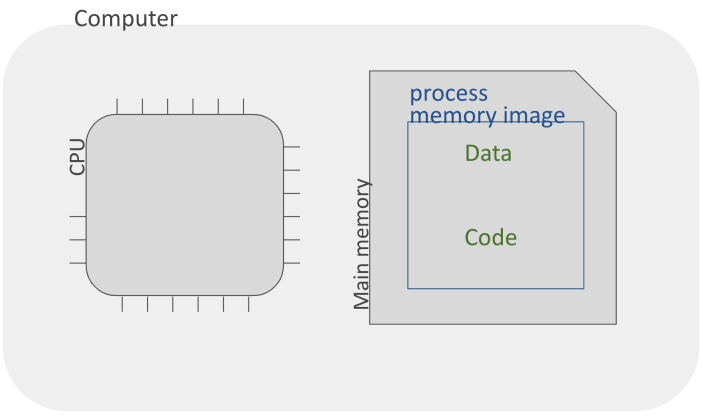
1.3.1 Program

A **program** is a set of instructions written by a human in a high-level programming language (such as C, Java, or Python) that implements an algorithm. When compiled, a program is translated into an *executable* (or binary) that the CPU can run. The executable is expressed in the language defined by the computer's **Instruction Set Architecture** (ISA).

Definition (ISA). *The **Instruction Set Architecture** (ISA) is the set of all instructions that a CPU can understand and execute. It forms an interface between the compiler (which translates high-level code into machine code) and the CPU.*



Definition (Von Neumann Architecture). *The vast majority of computers today follow the **Von Neumann architecture**, which is characterized by a single main memory that holds both data and instructions.*



The Von Neumann architecture

Definition (CPU Frequency). *A CPU's frequency indicates how many cycles it can perform in one second. For example, a 4.05 GHz CPU performs 4.05 billion cycles per second. In this context, a* ***cycle*** *is the minimum time needed for the CPU to complete an operation or for a result to become ready.*

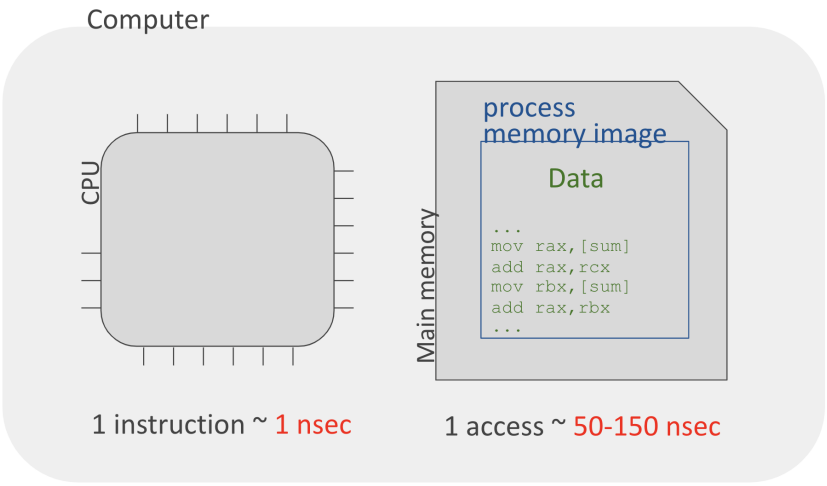
Question: *What is the meaning of the Hz metric?*

Answer: *Hertz (Hz) measures the number of cycles per second.* ***Question:*** *In the context of a CPU, what is a cycle?*

Answer: *A cycle is the minimum unit of time required for the CPU to produce a result from executing an instruction.*

1.4 Frequency Imbalance and CPU Caching

Modern systems exhibit a *frequency imbalance* between the CPU and main memory. While the CPU might complete an instruction in approximately 1 nsec, accessing data from DRAM typically takes 50–150 nsec. As a result, the CPU can spend a significant amount of time idle, waiting for data.

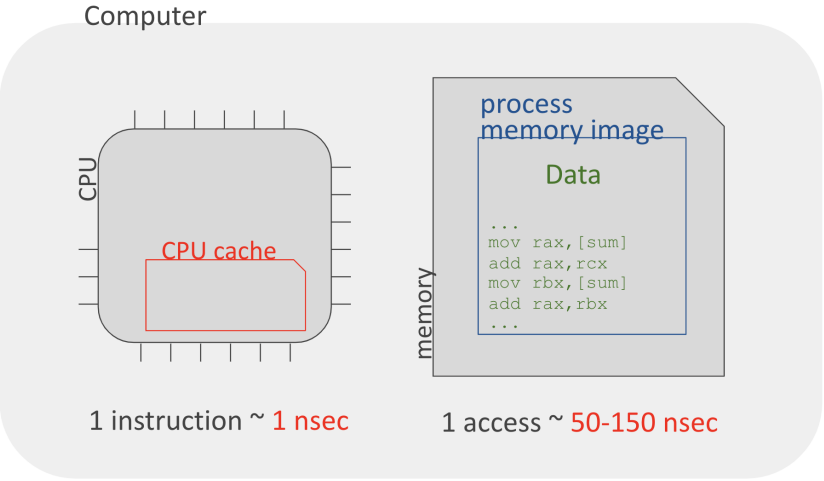


Question: How can we improve the efficiency of a system with such a frequency imbalance?

Answer: We can improve efficiency by using caching. A small, fast memory (the CPU cache) stores recently accessed data so that subsequent accesses are much faster.

1.4.1 CPU Caching

CPU caching adds a small amount of high-speed memory inside the CPU. When data is requested, the cache is checked first. If the data is already in the cache (a cache hit), the CPU retrieves it quickly. Otherwise (a cache miss), the data is fetched from the slower main memory and stored in the cache for future accesses.



1.5 Memory Accesses vs. I/O

1.5.1 Memory Accesses

When a process reads or writes data in main memory, it uses fast CPU instructions (load/store). These operations are efficient and do not interrupt the normal flow of the process.

1.5.2 Back to YouTube Fetching: System Calls in Action

Returning to our YouTube example, when the browser process calls a `send` syscall to request a video, the CPU stops executing the browser's code and switches to executing the more privileged code in the OS. This is because accessing external resources (network or storage) requires a syscall.

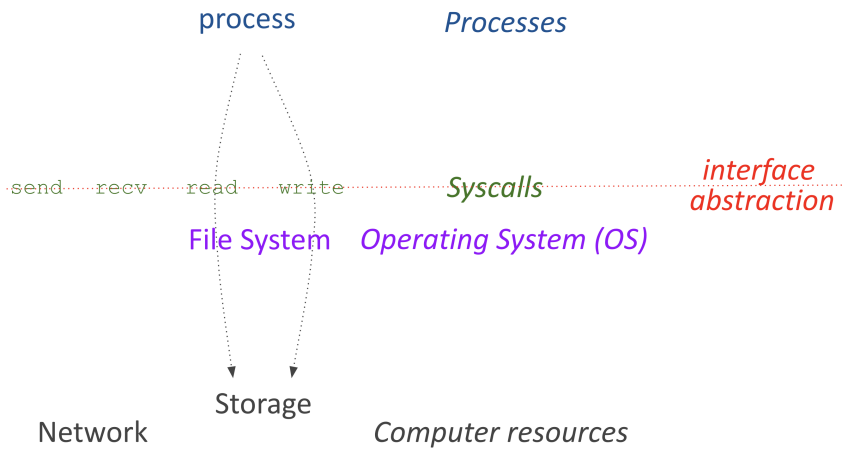
The browser C code → Compiler → *The browser executable*

```
...
const int connection = ...
const ssize_t bytes_send =
        send(connection,...);
if ((bytes_sent < 0) || ...) {
  int error_code = ...
  return error_code;
}
...
```

```
...
mov rax, 44
mov rdi, [connection]
mov rsi, ...
mov rdx, ...
syscall
...
```

1.5.3 Mixing Interfaces

When a process makes a syscall for network communication (such as `send` or `recv`), the CPU transitions from running the process's code to running the OS code associated with the network stack. This is an example of mixing different interfaces: the process interface (its own code) and the OS interface (syscalls).



Definition (Memory Access and I/O). ***Memory Access*** *refers to the CPU's direct read-/write operations using load/store instructions in main memory. In contrast,* ***I/O (Input/Output)*** *involves accessing external devices (such as storage or network) via system calls. I/O operations are generally more expensive because they require the CPU to switch context to execute privileged OS code.*

Exam Question: How is reading from main memory different from reading from storage or the network?

Answer: Reading from main memory uses direct CPU instructions (load/store) and is very fast (tens to hundreds of nanoseconds), whereas reading from storage or the network requires a syscall, which interrupts the process and involves additional overhead (microseconds to milliseconds).

Definition (I/O). ***I/O*** *(Input/Output) refers to operations that allow a process to access resources outside of its immediate control, such as storage devices or the network, via system calls. I/O operations are generally slower than direct memory accesses.*

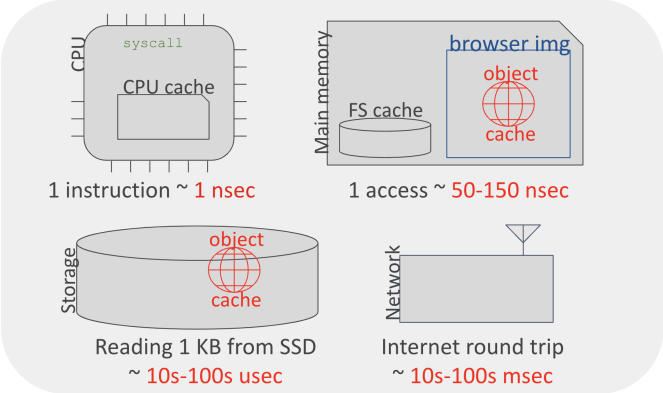
Exam Question: If a program does not create or manipulate any data, will executing this program require reading anything from memory?

Answer: Yes, executing the program will still require reading something from memory. Even if the program does not create or manipulate any data, the CPU must at least fetch the instructions of the program itself from memory

1.6 Communication Over the Internet

Internet communication involves transferring data over a network where different latencies are encountered:

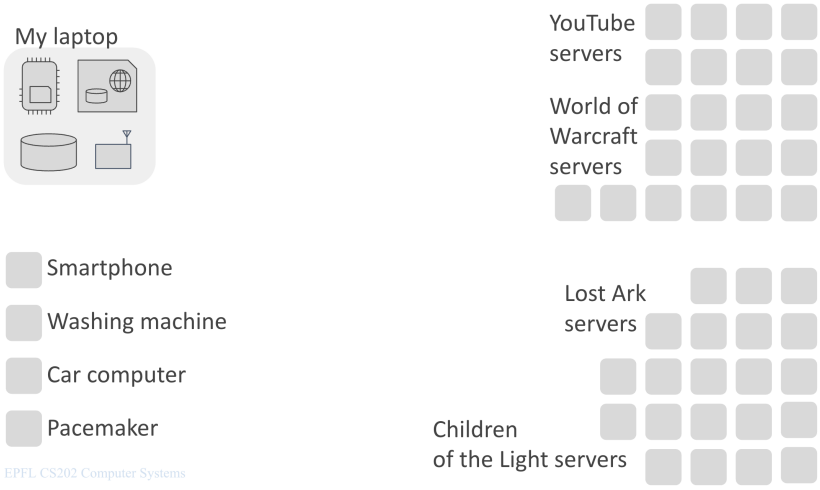
- Simple CPU instructions: ∼1 nsec.
- Main memory accesses: tens to hundreds of nanoseconds.
- Reading 1 KB from an SSD: tens to hundreds of microseconds.
- Requesting data over the Internet: several to hundreds of milliseconds.



These delays make network communication expensive in terms of time, which is why caching is critical.

1.6.1 End Systems

The Internet is composed of **end systems**—devices that use the network for communication. These include laptops, smartphones, household appliances, connected cars, and even medical devices, as well as large cloud servers.

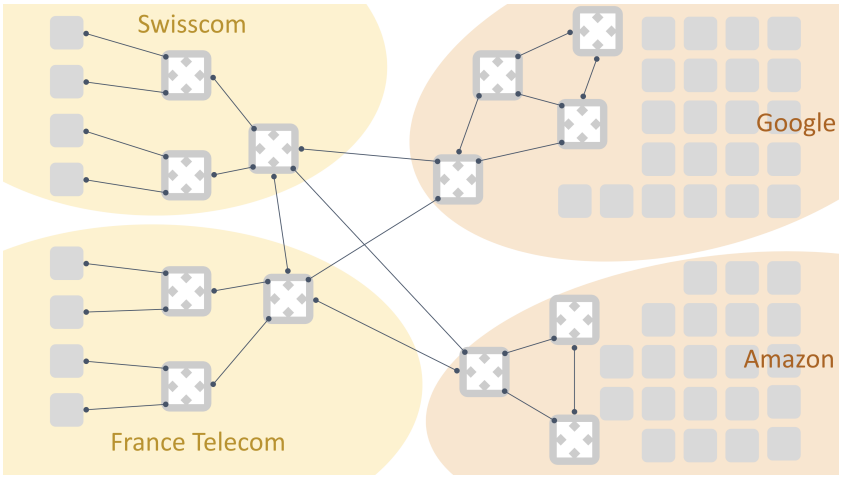


1.6.2 Packet Switches and Network Links

In addition to end systems, the Internet relies on:

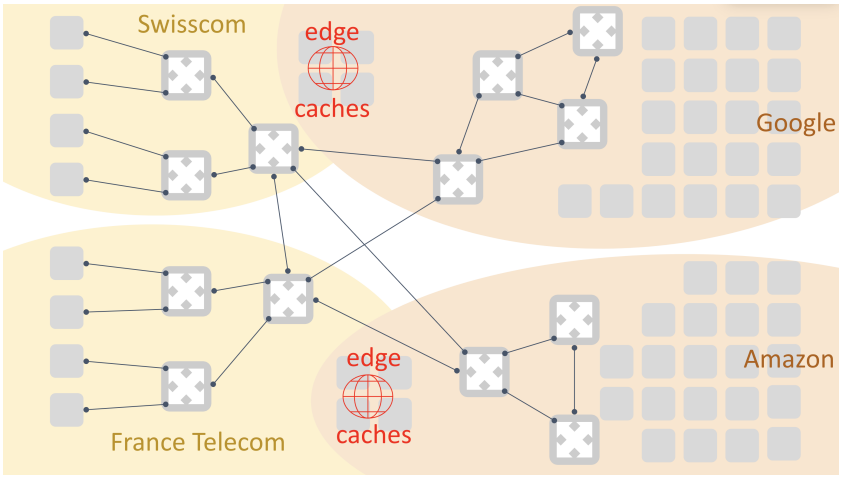
- **Packet Switches**: Devices that route data between end systems.
- **Network Links**: Physical connections that interconnect packet switches and end systems.

These components are managed by Internet Service Providers (ISPs) as well as major cloud providers.



1.6.3 Edge Caches

To reduce the load on cloud data centers and improve user performance, large cloud providers often deploy **edge caches** within ISP networks. These caches store frequently accessed content closer to the end-users, reducing latency and network traffic.

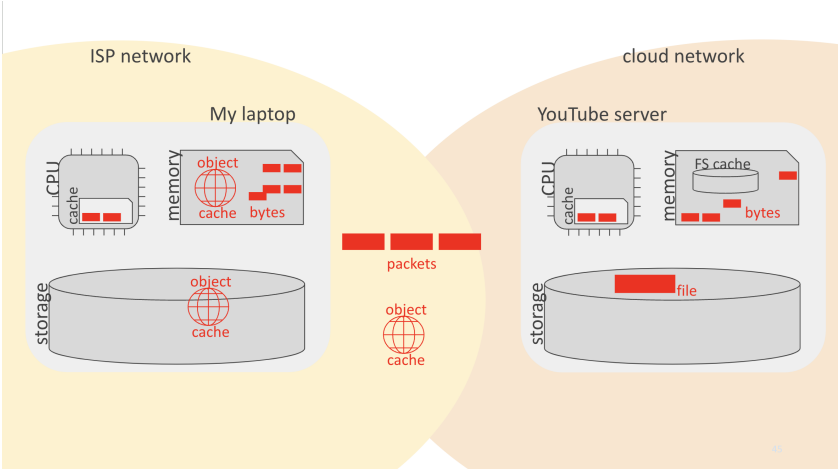


1.7 Summary

In this lecture, we traced the journey of a YouTube video and introduced several fundamental concepts:

- **Programs, Processes, and Threads:** Programs stored on disk become processes (and threads) when executed.
- **Distributed Applications:** Different processes communicate over networks using well-defined communication protocols.

- **Interfaces and Abstractions:** System calls, APIs, and caching abstract the complexity of hardware resources.
- **The Operating System:** Acts as an intermediary between processes and hardware resources.
- **Performance Considerations:** Frequency imbalances between the CPU and memory are mitigated by caching at various levels (CPU cache, file system cache, object caches).



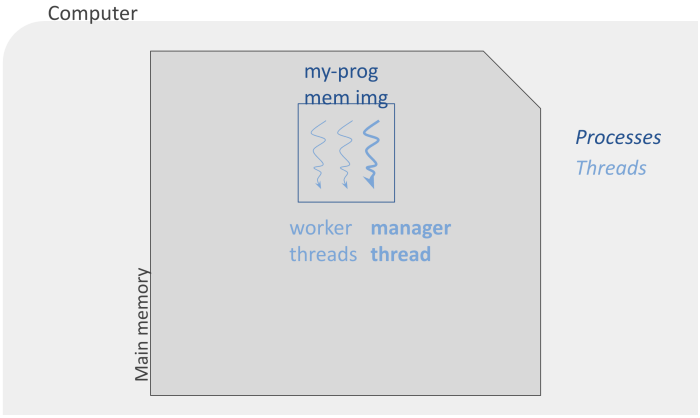
Chapter 2

L2 - All About Processes

This chapter provides an overview of the fundamental concepts underlying modern process management and memory organization in computer systems. We discuss multithreading, CPU registers, the role of compilers, and memory organization—including both stack and heap memory—as well as virtualization techniques that allow multiple processes to coexist seamlessly.

2.1 Multithreading

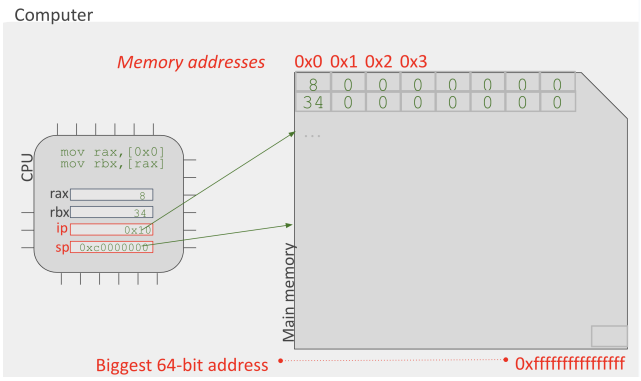
When a program runs, the processor loads it into main memory and creates a thread. In a multithreaded program, there is typically one *manager* thread that delegates work to several *worker* threads. For instance, when computing the sum of a large set of numbers, the workload can be divided into subsets, with each worker thread processing a portion of the data while the manager coordinates the overall computation.



2.2 Registers

CPU registers are small storage locations within the processor that hold data and instructions needed during execution. For example, the `mov` instruction might transfer data from one register (or memory location) to another. Key registers include:

- **Instruction Pointer (IP):** Keeps track of the next instruction to be executed.
- **Stack Pointer (SP):** Points to the current top of the stack in main memory.



Definition (Compiler). *A compiler translates high-level source code (such as C) into low-level executable code (often Assembly language). This translation involves parsing, optimization, and the generation of machine-specific instructions.*

2.3 Memory Organization

I'll go a little bit deeper for our fellow syscoms, I also recommend understanding how LIFO works before reading this, if you're too lazy for that, it's basically in the name Last In First Out, means that the last item pushed onto the stack is the first one to be removed, just like stacking plates— you take the top plate first before reaching the ones below.
In modern computer architectures, a process's memory is divided into several distinct segments, each serving a specific role during program execution. Understanding these segments is fundamental for effective programming and debugging.

Definition (Memory Segments). *A process's memory image is typically divided into the following segments:*

- ***Text Segment:*** *Contains the executable code and embedded constants. It is usually marked as read-only to prevent accidental modification.*
- ***Data Segment:*** *Stores global and static variables. This segment is often subdivided into:*
  - ***Initialized Data:*** *Variables explicitly initialized by the programmer.*
  - ***Uninitialized Data (BSS):*** *Variables that are declared but not explicitly initialized, and are set to zero by default.*
- ***Heap Segment:*** *Used for dynamic memory allocation. Memory here is allocated and deallocated during runtime by the programmer (or automatically via garbage collection in some languages). The heap typically grows upward (from lower to higher memory addresses).*
- ***Stack:*** *Manages function calls, local variables, and function parameters. The stack is automatically managed by the CPU, growing downward (from higher to lower memory addresses) as functions are called.*

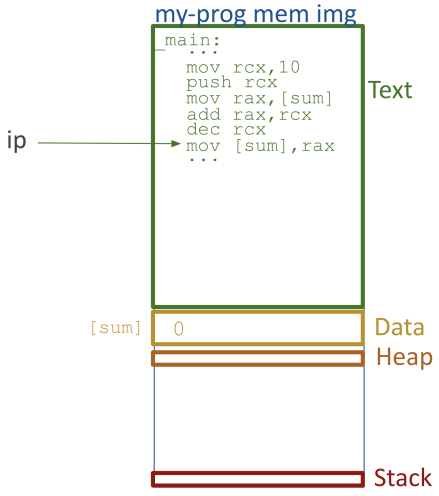
2.3.1 The Stack

The stack is a dedicated region of memory that the CPU uses to manage function calls and local variables. When a function is invoked:

1. The CPU executes a `call` instruction, which pushes the return address onto the stack.
2. A new *stack frame* is created to store local variables and function-specific data.
3. Upon function return, the stack frame is removed (or "unwound"), and control returns to the calling function.

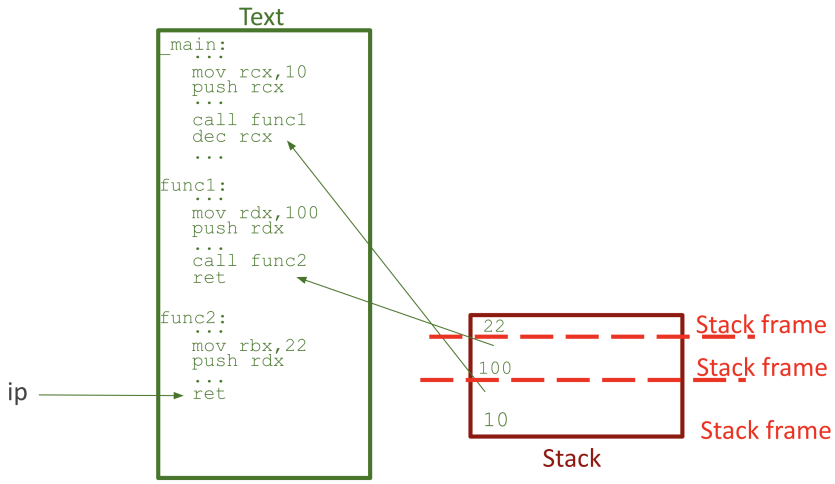
Key Characteristics of the Stack:

- **Automatic Management:** The CPU automatically handles pushing and popping of data.
- **Growth Direction:** Grows downward, from higher to lower memory addresses.
- **Contents:** Stores return addresses, local variables, and sometimes function arguments.



Stack Frames

Each function call creates its own *stack frame*, a self-contained section that isolates the function's local data. This segmentation helps maintain the correct scope and lifetime for local variables and ensures that return addresses are preserved. The following diagram illustrates the organization of stack frames during nested function calls:



2.3.2 Heap Memory

The heap is used for dynamic memory allocation, where memory is allocated and deallocated as needed during runtime. Unlike the stack:

- **Manual vs. Automatic Management:** In languages such as C or C++, the programmer is responsible for explicitly allocating (using `malloc` or `new`) and deallocating (using `free` or `delete`) heap memory. In contrast, some modern languages employ automatic garbage collection.
- **Growth Direction:** The heap typically grows upward, from lower to higher memory addresses.
- **Lifetime:** Data allocated on the heap persists beyond the scope of the function that created it, until it is explicitly freed or garbage collected.

2.3.3 Data and Text Segments

Text Segment: This segment contains the program's executable code and constant values. Its read-only nature helps prevent inadvertent modifications during execution. **Data Segment:** This segment holds global and static variables. It is divided into:

- **Initialized Data:** Variables that have been assigned an initial value at compile time.
- **Uninitialized Data (BSS):** Variables that are declared but not explicitly initialized; these are automatically set to zero at program startup.

Definition (CPU Registers). *Two registers are critical for process execution:*

- ***Instruction Pointer (IP):*** *Points to the next instruction in the text segment.*
- ***Stack Pointer (SP):*** *Points to the top of the stack.*

Definition (Process and Thread Identifiers).
A process is considered to be running *if at least one of its threads is executing; otherwise, it is not running.*

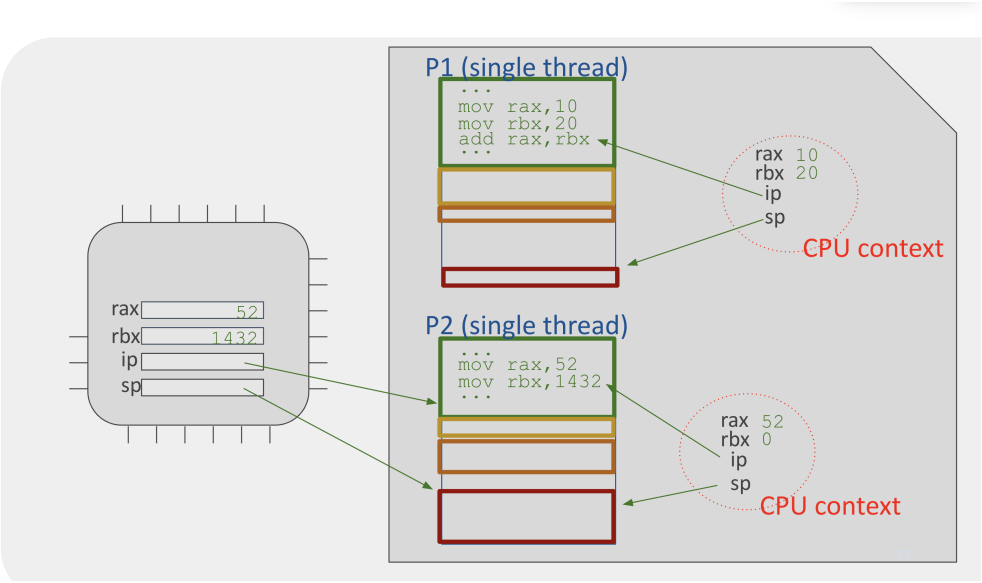
- ***Process ID (PID):*** *A unique identifier assigned to each process.*
- ***Thread ID (TID):*** *A unique identifier for each thread, which may be unique within a process or across the entire system, depending on the operating system.*

Definition (Resource Sharing). *Processes and threads share system resources such as CPU and memory. Each thread is given the illusion of having exclusive access to the CPU, and each process appears to have dedicated memory, even though these resources are actually shared.*

Definition (CPU Sharing). *The CPU is time-shared among multiple threads. This virtualization is achieved through context switching, where the CPU rapidly switches between threads, giving each one the impression of exclusive use of the processor.*

Example: Two Programs Running on a Single Core

Example 2.3.3.1. *Consider two programs running on a single-core processor. Each program is assigned a CPU context, which includes register values such as* `rax`, `rbx`, *the stack pointer (SP), and the instruction pointer (IP). When switching between programs, the CPU saves the current context to memory and loads the context of the next program, allowing the programs to resume correctly.*



Definition (Thread's CPU Context). *A thread's CPU context comprises the values of all CPU registers at the moment it was last executing. In a single-threaded process, this context represents the entire process state.*

Definition (Context Switching). *Context switching is the process by which the CPU switches from executing one thread to another. It involves:*

1. *Saving the current thread's CPU context to memory.*
2. *Restoring the CPU context of the thread to be executed next.*

   Each thread has the illusion that it's occupying the CPU ***alone****.*

This mechanism enables CPU virtualization but introduces performance overhead due to additional memory accesses.

Definition (Process). *A process is defined by:*

- *A unique Process ID (PID).*
- *A memory image that includes the text, data, heap, and stack segments.*
- *The CPU contexts of each thread within the process.*
- *Associated resources such as file descriptors.*

Remark: If two threads belong to the same process, does each have its own CPU context, or do they share one? The answer is that each thread should be able to continue its execution independently.

Definition (Memory Sharing). *Memory in a system is space-shared among processes; however, virtualization ensures that each process operates within its own isolated address space. This is achieved through virtual-to-physical address translation.*

Definition (Virtual and Physical Addresses). *Virtual addresses allow processes to operate as if they have exclusive access to memory. For example, two processes might both use the virtual address* `0x400000`*; however, these addresses map to different physical addresses:*

- *Process* P_1*: Virtual* `0x400000` $\rightarrow$ *Physical* `0x1234AFF8`
- *Process* P_2*: Virtual* `0x400000` $\rightarrow$ *Physical* `0xABCD5678`

Definition (Virtual Address Space). *Each process is allocated its own virtual address space, which is shared among all its threads. This abstraction allows developers to ignore the complexities of physical memory allocation.*

Definition (Address Translation). *Address translation is the process by which a virtual memory address is converted into a physical memory address. While essential for memory virtualization, this translation incurs a performance cost.*

2.3.4 Stack Smashing

Stack smashing is a type of buffer overflow vulnerability where an attacker deliberately overwrites parts of the memory on the call stack. This typically happens when a program writes more data into a fixed-size buffer than it can accommodate, thereby corrupting adjacent memory areas such as the function's return address.

How It Works: When a function is called, a stack frame is created that contains local variables, return addresses, and other control data. If a buffer does not have proper bounds checking, an input exceeding the buffer's capacity can overflow into these critical areas. For example:

1. A fixed-size buffer is allocated on the stack.
2. Excess input data overwrites memory beyond the buffer.
3. The return address (or other control data) is corrupted.
4. On function return, control is transferred to an address chosen by the attacker, potentially executing malicious code.

2.3.5 Summary: CPU and Memory Virtualization

- **CPU Virtualization:** Threads time-share the CPU through context switching, which gives each thread the illusion of exclusive CPU access.
- **Memory Virtualization:** Processes space-share memory via virtual-to-physical address translation, ensuring that each process operates in its own isolated address space.

2.3.6 Conclusion

Modern operating systems are designed to enable multiple programs to share CPU and memory resources seamlessly. Through context switching and address translation, both the CPU and memory are effectively virtualized. This abstraction simplifies development, as compilers and developers can design programs without needing to manage these low-level resource-sharing details directly.

Chapter 3

L3 - Sharing the CPU

This chapter introduces the mechanisms by which an operating system (OS) manages access to the CPU, ensuring both security and fairness among processes. We discuss CPU privilege levels, the limited direct execution model, and the role of system calls (syscalls) in process management.

3.1 The OS as a Special Program

The operating system (OS) is a fundamental software layer that manages hardware resources and provides essential services for user applications. Unlike typical applications, the OS operates at different privilege levels and ensures secure and efficient execution of processes and threads.

3.1.1 Limited Direct Execution

Professor Analogy cont. - The children have direct access to the TV but are restricted by Kids Mode.
Limited direct execution is a method that allows a thread to run directly on the CPU while enforcing certain restrictions:

- *Direct Execution:* The CPU executes the thread's instructions without any intermediate emulation.
- *Limited:* The thread cannot perform operations that require high privileges; instead, it must request system assistance via syscalls.

The CPU runs in Limited Direct Execution. Let's see how it's limited

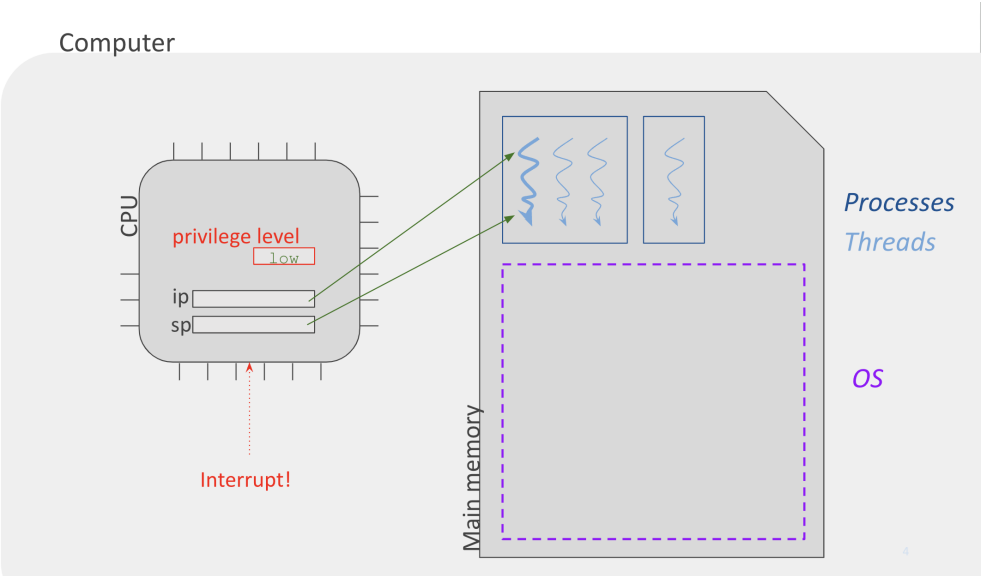
3.1.2 CPU Privilege Levels and Execution Modes

Professor Analogy - Imagine you have children who want to watch TV. You can either keep the remote with you and ask them to call you whenever they need to change the channel, or you can enable Kids Mode on the TV, allowing them limited access without needing to ask for permission each time.

Personal Remark - Kernal code always *run in High Privilege mode, and because of this we say that the OS* may *run in High Privilege mode (kernel is inside of the os), do not confuse both !*

The OS shares the CPU and main memory with normal user processes and threads. However, its execution mode differs based on its role at any given time:

- When the OS executes, the CPU *may* be in **high privilege mode** (often called kernel mode), allowing it unrestricted access to all system resources.
- When a normal process or thread executes, the CPU is in **low privilege mode** (user mode), restricting access to critical system resources.



This privilege system exists primarily for security reasons, enforcing the **principle of least privilege**, where each process has only the necessary access rights required to perform its task.

3.1.3 The Kernel: Core Component of the OS

A key component of the OS is the **kernel**, responsible for:

- Creating and managing processes and threads.
- Allocating system resources (CPU time, memory, I/O devices, etc.).
- Ensuring that each process has a designated portion of memory, including stack, data, and text segments.
- Enforcing security and isolation between processes.

The kernel always runs in **high privilege mode** (kernel mode), which is why this mode is sometimes referred to as *kernel mode*.

3.1.4 Process Management and Context Switching

The OS does not execute continuously but rather runs only when necessary. It performs essential tasks, prepares the environment for the next process, and then switches out, allowing user processes to execute.

- The OS uses **context switching** to transition between processes efficiently, preserving the state of the current process before switching to another.

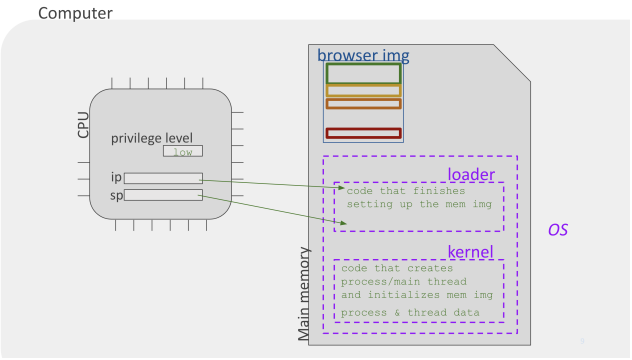
- By switching between kernel mode and user mode, the OS ensures that user applications run securely and do not directly manipulate hardware resources.

The Loader: Setting Up Process Memory

Another critical component of the OS is the **loader**, which is responsible for preparing a program for execution:

- The loader completes the setup of the process's memory image.
- It copies the command-line arguments used to launch the program (e.g., `ls -a`, where `-a` is an argument).
- These arguments are stored in the **stack** of the main thread of the new process to ensure accessibility.

If these arguments were stored in the loader's stack instead, the process would not be able to access them after execution begins.

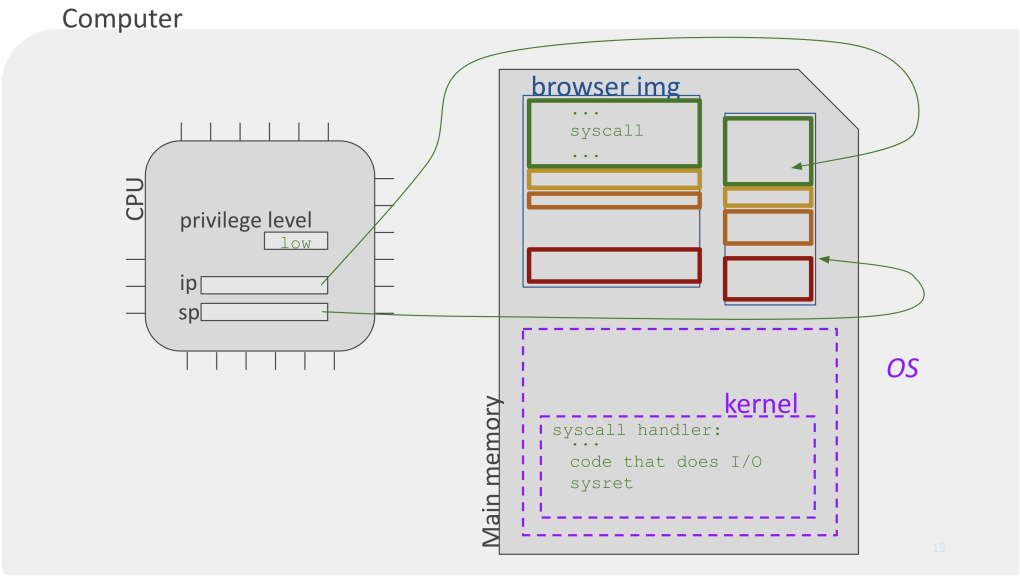


This layered privilege model ensures a secure and efficient execution environment, maintaining stability and protection across all processes within the system.

3.1.5 Syscalls

Personal Remark - Syscalls are NOT needed when executing OS code, they only allow to cross the priviledged barrier when running a unpriviledged code. A *syscall* is the mechanism by which a user process requests services from the OS. When a syscall is invoked:

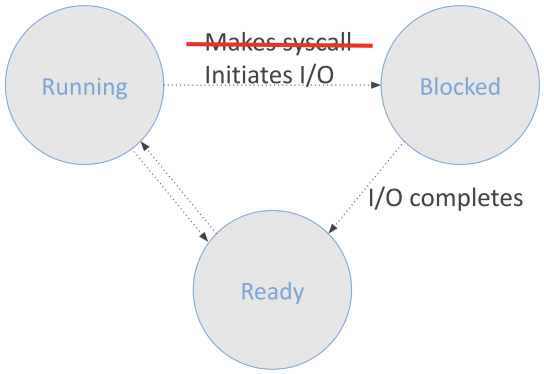
1. The CPU temporarily elevates the privilege level.
2. The control is transferred to the OS to execute the privileged code.
3. If the syscall involves an I/O operation, the process may be blocked while waiting for a response, and the CPU may perform a context switch to another thread.
4. Once the operation is complete, the CPU returns to low privilege.


Computer
CPU
privilege level
low
ip
sp
browser img
...
syscall
...
OS
kernel
syscall handler:
...
code that does I/O
sysret
Main memory

3.1.6 Process I/O and Scheduling

When a process initiates an I/O operation:

- It transitions from *Running* to *Blocked* as it waits for the I/O to complete.
- Once the I/O is complete, the process moves to the *Ready* state.
- The OS scheduler then selects processes from the Ready queue to run, ensuring fair CPU utilization.



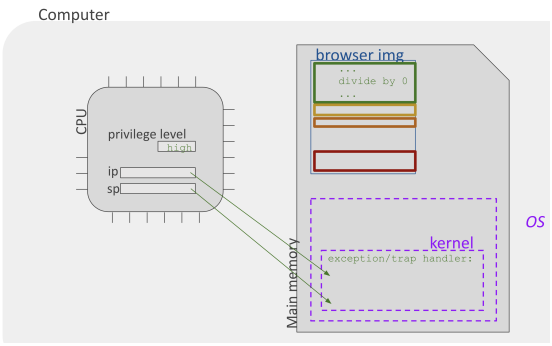
3.2 The Kernel's Job cont.

The Kernel handles several types of events:

- **Syscalls:** Requests from running threads for system-level services.
- **Exceptions/Traps:** Synchronous signals generated when a thread executes an illegal or erroneous operation (e.g., division by zero).
- **Interrupts:** Asynchronous signals from external devices (e.g., mouse events, network packets) requiring immediate attention.

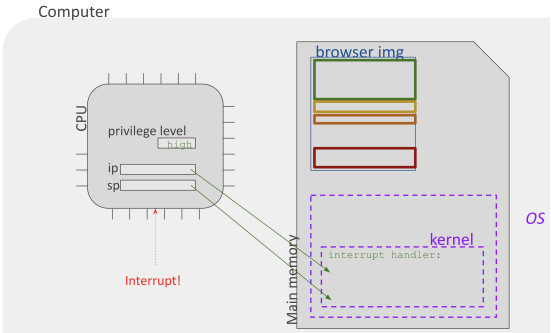
Exception Handling

What happens if the browser executes unauthorized code or encounters an error, such as dividing by zero ?
When a process executes an illegal operation, the CPU raises an exception. The kernel then takes over to handle the error safely.



Interrupt Handling

Interrupts are triggered by external events and are managed by both hardware and software. The hardware raises the interrupt, and the kernel (via an interrupt handler) processes it.

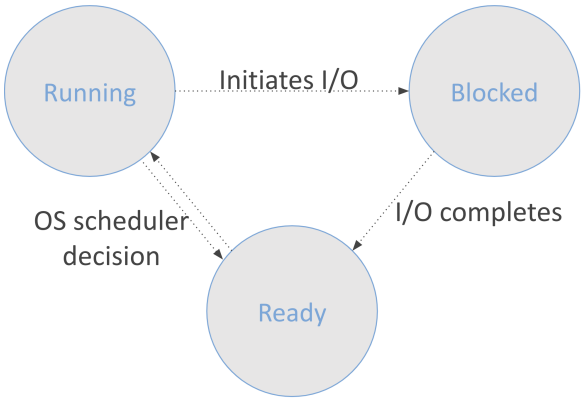


The Timer Interrupt

Definition (Timer Interrupt). *The timer interrupt is raised at regular intervals (typically every few milliseconds). Its handler invokes the OS scheduler to decide which process runs next, ensuring that no process monopolizes the CPU.*

The OS Scheduler

The OS scheduler manages the state transitions of processes (Running, Blocked, Ready) based on various scheduling algorithms, thereby ensuring equitable CPU access.



Summary - Limited Direct Execution

- Normal threads execute in low privilege.
- Operations requiring high privilege are performed via syscalls, exceptions, or interrupts, which invoke the kernel.
- Timer interrupts ensure that the OS scheduler periodically gains control, maintaining fairness.

Limited direct execution is essential for safely and efficiently sharing the CPU among multiple processes.

3.3 Executing Syscalls — Process Management

Processes are created, modified, and terminated using various syscalls. We now discuss the key syscalls involved in process management.

3.3.1 Syscall Definitions

Definition (Exit Syscall). *The `exit` syscall terminates a process. It never returns because, by the time control would return to the calling process, the process no longer exists.*

```
1  _exit(0);
2  .
3  .
4  .
5  .
6  .
7  .
8  .
9  .
10 .
11 .
12 .
```

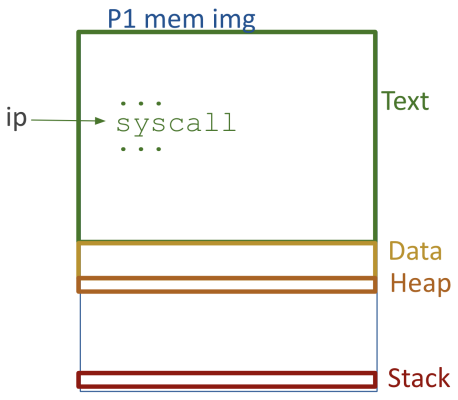


Figure 3.1: Exit Syscall: Code snippet and stack visualization.

Definition (Exec Syscall). *The `exec` syscall replaces the current process image with a new program. It preserves the process ID and file descriptors while discarding the old program's code, data, and stack. On success, it does not return; on failure, it returns `-1`.*

```
execvp("date", args);
```

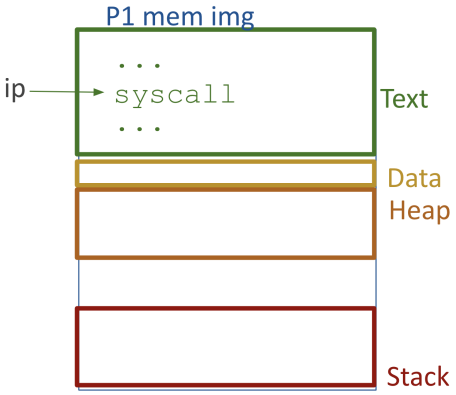


Figure 3.2: Exec Syscall: Code snippet and process image.

Definition (Fork Syscall). *The `fork` syscall creates a new child process by duplicating the calling process. Both the parent and child continue execution from the point of the fork, but in separate memory spaces. The fork returns `0` to the child and the child's process ID (PID) to the parent.*

```
int fs = fork();
if (fs == 0) {
   // Child process code
} else {
   // Parent process code
}
```

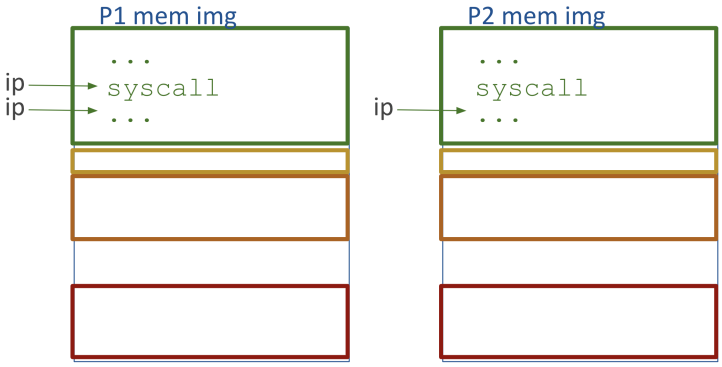


Figure 3.3: Fork Syscall: Example code and the resulting stack layout.

Definition (Wait Syscall). *The `wait` syscall allows a parent process to block until one of its child processes terminates. If no child process exists, `wait()` returns an error.*

3.3.2 Process Creation and Cleanup

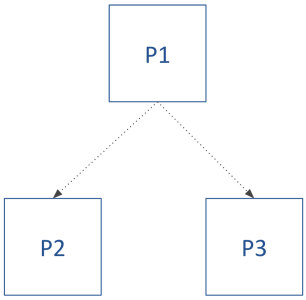
Processes are typically created by combining the `fork` and `exec` syscalls. For example:

```
int fs = fork();
if (fs == 0) {
   execvp("date", args);
} else {
   wait(fs);
}
```

When a parent process calls `wait()`, it is blocked until a child terminates, ensuring proper cleanup of child processes.

3.4 The OS Process Graph

The OS maintains a process graph where each square represents a process and each arrow indicates the parent-child relationship. These kind of graphs are crucial for understanding process creation and hierarchy.



3.5 Key Processes in the OS

Some critical processes managed by the OS include:

- **GUI Processes:** Manage the graphical user interface.
- **Terminal Processes:** Handle command-line interactions.
- **Init Process:** The first process created by the kernel, responsible for starting system services.
- **Idle Process:** Executes when no other process is runnable.

Conclusion - The Role of Syscalls

Syscalls provide the interface through which processes access system resources such as storage and networks. They also facilitate self-management operations, including process creation, modification, and cleanup. Through mechanisms such as limited direct execution, exceptions, and interrupts, the OS ensures that the CPU is shared safely and efficiently among all processes.

Chapter 4

L4 - Memory

This chapter covers the fundamentals of main memory, process memory images, memory virtualization, and the CPU's role in managing memory.

4.1 Main Memory

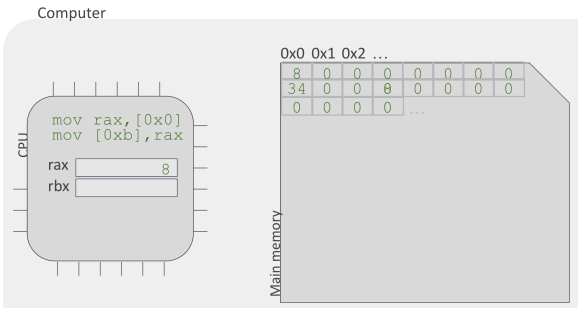
Main memory is conceptualized as a linear array of bytes, where each byte has a unique memory address (e.g., `0x0`, `0x1`, `0x2`, etc.). Each byte can store any value that fits within its 8-bit capacity, and importantly, the value stored in a given byte is independent of its memory address. For instance, the byte at address `0x0` may contain the value 8, 0, or any other valid 8-bit number.

4.1.1 Memory Operations by the CPU

The CPU interacts with main memory by executing specific instructions to read from and write to it. These operations are fundamental to both data processing and code execution:

- **Read Operation:** The CPU issues an instruction to read a block of bytes (for example, 8 bytes starting at address `0x0`) and loads the result into a register (such as `rax`).
- **Write Operation:** The CPU executes an instruction that writes data from a register (e.g., `rax`) into a block of memory (for example, starting at address `0xb`).

Although main memory stores only numbers, the CPU interprets these numbers differently depending on whether they represent data (such as variables) or executable code (such as the instruction `mov rax, [0x0]`).



4.1.2 Instruction Pointer

A key component in the CPU's control mechanism is the *instruction pointer* (IP), in some contexts (ie. FDS, Comparch), this register is also known as the *program counter* (PC), but the term "instruction pointer" more precisely describes its function.

Definition (Instruction Pointer).

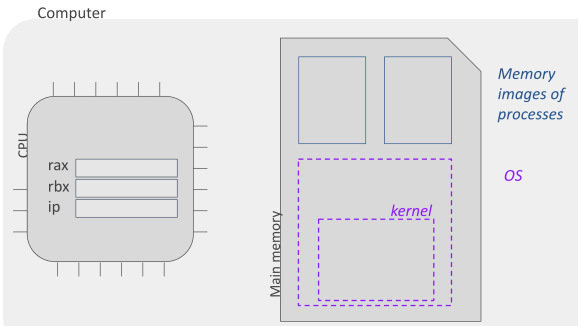
*The **instruction pointer** is a CPU register that holds the memory address of the next instruction to be executed.*

4.1.3 Subparts of Main Memory

Main memory contains not only the memory images of individual processes but also the code and data essential to the operating system (OS). The OS comprises several critical components that ensure the proper operation and usability of the computer. These components include:

- **Process Memory Images:** Every process has its own memory image, typically divided into:
  - **Data Segment:** Stores global variables.
  - **Stack Segment:** Contains local variables, return addresses, and other function call-related data.
  - **Heap Segment:** Holds dynamically allocated memory (e.g., allocated via `malloc`).

- **Operating System Code:** This comprises all the code necessary for the computer's operation and usability. OS code is organized into:
  - *Kernel:* The central component of the OS, running in high-privilege mode. It manages system resources, hardware interactions, and security, ensuring the core functions of the computer operate correctly. It is neither a process nor a library (end of lecture explains).
    - It creates and deletes processes and threads.
    - It initiates I/O.
    - It handles errors and interrupts.
    - It decides which thread will run next.
  - *Processes:* Such as the graphical user interface (GUI) and terminal applications, which provide user-level interaction with the system.
  - *Libraries:* Modules like the standard C library that provide a suite of functions, which are dynamically integrated into processes when called.



4.2 Process Memory Image

Definition (Process Memory Image).

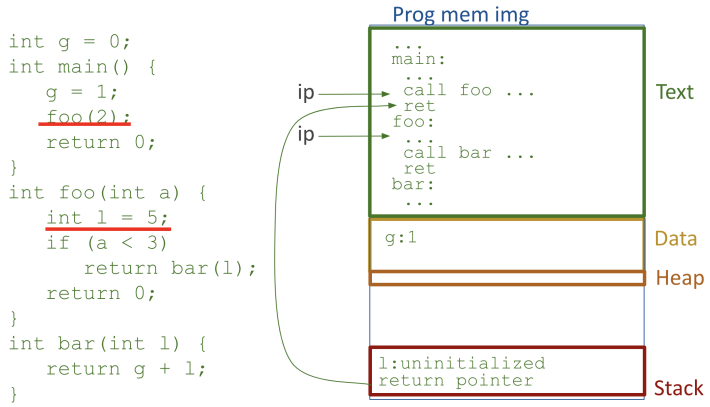
*A **process memory image** is the complete layout of a process's memory, comprising:*

- *The* text segment *for the process's code.*
- *The* data segment *for global variables.*
- *The* stack segment *for local variables and return pointers.*
- *The* heap segment *for dynamically allocated memory. (eg. malloc)*

Exam Question: We provide you with a C program. Your task is to draw the memory image of the corresponding process at different points in the program.

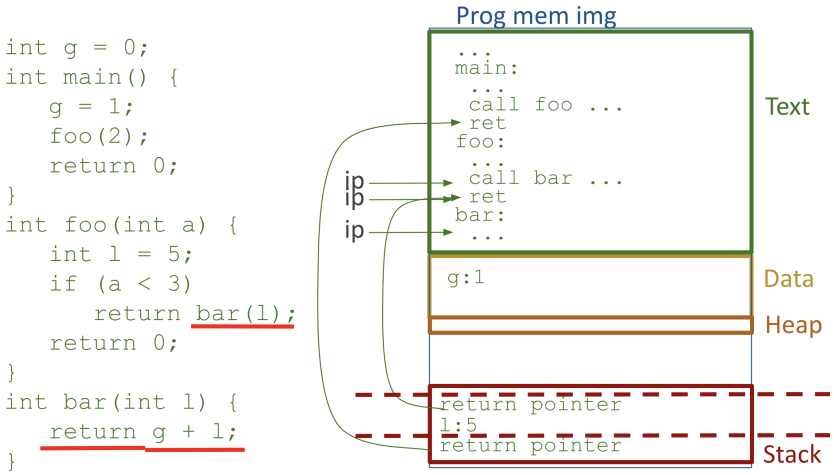
1. Mark each segment, even if it is empty.
2. Draw a schema of the code in the **Text Segment**, including only function names and calls in assembly.
3. Identify global variables and place them in the **Data Segment** (e.g., `g:0`).
4. Simulate each step of the program's execution to fill the **Heap** and **Stack** segments accordingly. This includes local variables, memory allocations, and function calls.

```
int g = 0;
int main() {
   g = 1;
   foo(2);
   return 0;
}
int foo(int a) {
   int l = 5;
   if (a < 3)
      return bar(l);
   return 0;
}
int bar(int l) {
   return g + 1;
}
```



Exam Question: Show the memory image and indicate the moment when the stack reaches its maximum size.
For the program shown above, the expected result would be:

```
int g = 0;
int main() {
   g = 1;
   foo(2);
   return 0;
}
int foo(int a) {
   int l = 5;
   if (a < 3)
      return bar(l);
   return 0;
}
int bar(int l) {
   return g + 1;
}
```



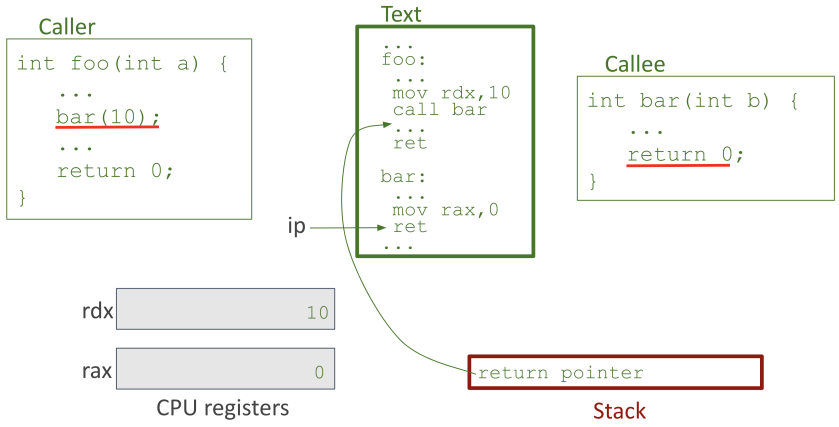
4.2.1 Optional - Stack and Register Functioning

In modern computer architectures, the process of a function call involves a coordinated interplay between CPU registers and the stack.

Definition (Function Call Mechanism).

During a function call:

1. *The **caller** passes arguments to the **callee** by storing values in designated CPU registers.*
2. *The **callee** processes the call and returns a result by placing it in a specific register (for example, the `rax` register).*



Example 4.2.1.1 (Illustrative Function Call)**.** *Consider the scenario where function `Foo` calls function `Bar`:*

1. ***Argument Passing:** `Foo` stores the argument value (e.g., **10**) in a CPU register.*
2. ***Return Value Handling:** An implicit agreement between foo and bar that the return value will be stored in a particular register (eg. rax). `Bar` processes the argument and stores its return value in another register (e.g., `rax`). Later, `Foo` retrieves this value by accessing that register.*

A caller and a callee share common infrastructure by using CPU registers to maintain their context during the call. In addition, the stack is used to preserve register states when necessary:

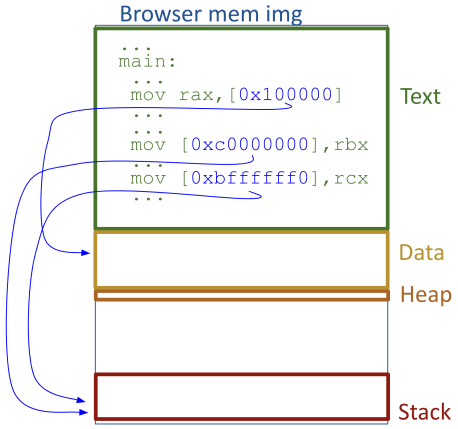
- **Caller-Saved Registers:** The caller saves certain registers to the stack before the call and restores them after the call returns.
- **Callee-Saved Registers:** The callee saves its registers at the beginning of the function and restores them before returning.

The stack, the register the calling conventions form a caller/callee interface.
Adhering to these calling conventions is critical; for instance, if the callee writes into the caller's stack frame, it may lead to stack smashing and compromise program stability.

4.3 Memory Virtualization

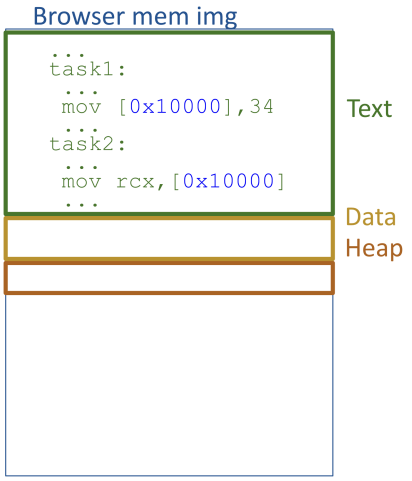
In modern operating systems, each process references memory using virtual addresses. Underneath, these virtual addresses are translated to physical addresses. Importantly, each process has its own virtual address space, which means that two processes may use the same virtual address while referring to entirely different physical locations. This design creates the **safe illusion** that main memory "belongs" exclusively to each process, greatly simplifying program development and enhancing security.

Each address in the image refers to an address within its own stack. The addresses shown are *virtual*, meaning they are process-specific (e.g., address 0 in one process does not necessarily correspond to address 0 in another).



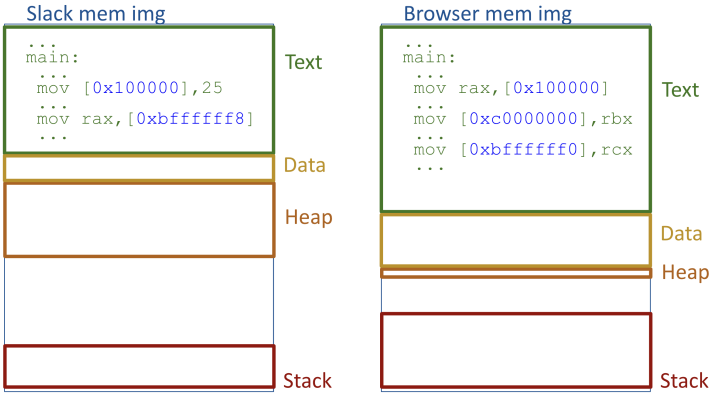
Exam Question: If two memory instructions (in the same process) read the same virtual address, is it the same physical address?

Answer: Yes, when they are translated to the same physical memory address, as they belong to the same virtual address space.



Exam Question: Are these two processes accessing the same memory location?

Answer: No, they are not actually accessing the same physical memory. Although both use the address 0x10000, each process runs in its own virtual address space.



The mechanism of memory virtualization creates a *safe illusion* in which it appears that the main memory is exclusively owned by each process. This design not only simplifies the generation of executable programs but also enforces security by ensuring that a process can only access its own memory image.

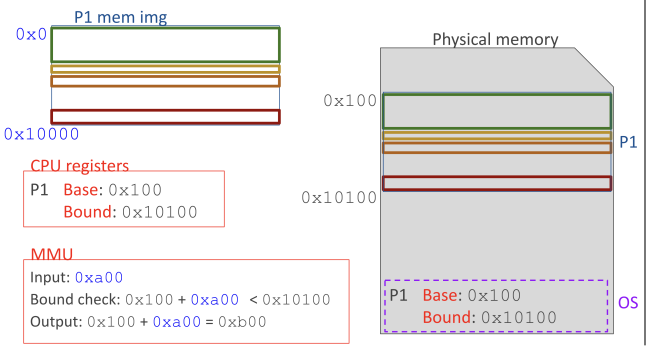
Definition (Contiguous Memory).

Contiguous memory refers to a block of physical memory addresses that are sequentially arranged. In this allocation scheme, the entire memory image of a process is stored in one unbroken segment, simplifying the translation from virtual to physical addresses.

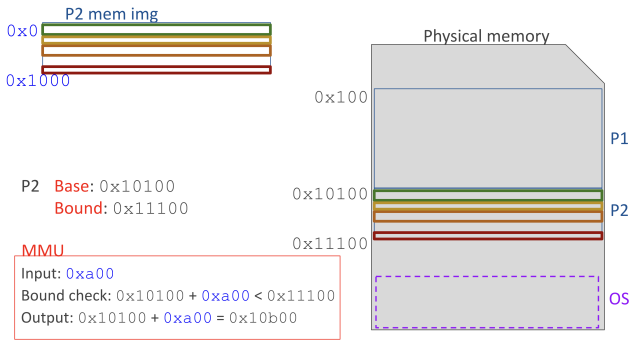
4.3.1 Memory Management Unit — Simple Implementation

The **Memory Management Unit (MMU)** is a specialized piece of *hardware* that translates virtual memory addresses into physical addresses.

For each process, the **OS kernel** sets up *base* and *bound* registers (which are physically stored in the CPU). The MMU then uses these values to ensure that the process's memory image is allocated in a contiguous block of physical memory.



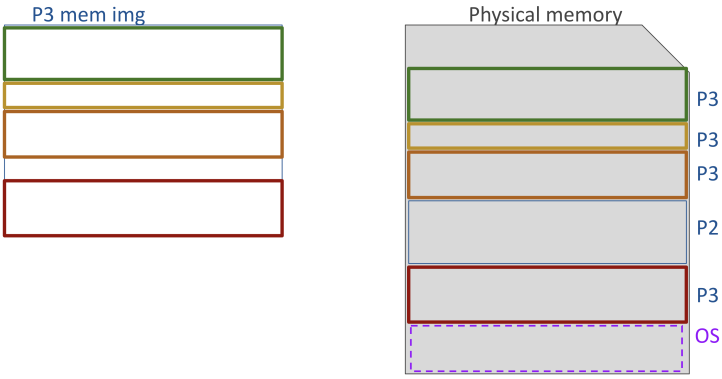
When a second process (P_2) is introduced, the MMU checks its corresponding base and bound registers to determine the physical memory range in which P_2 should be placed.



In this **base–bound** scheme, each process's memory image starts at its base address and extends just before its bound address. This approach is *safe* (preventing a process from accessing memory outside its allocation) and preserves the *illusion* of owning the entire memory.

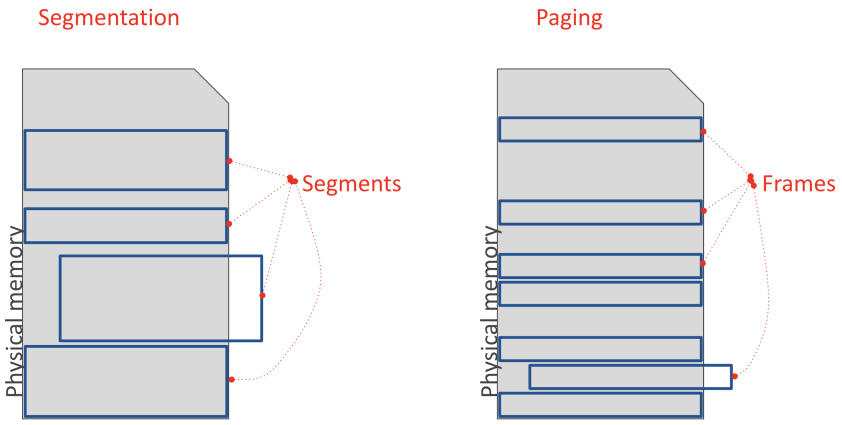
However, because each process must reside in one contiguous memory block, **fragmentation** can occur.

For example, when process P_1 terminates, it might leave a gap that is too small for a new process P_3, even if the total available memory is sufficient.

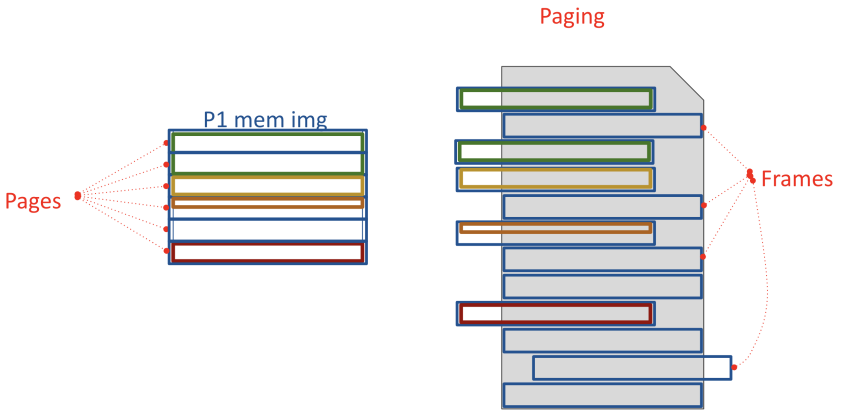


To address the inefficiency of requiring each process to occupy a single contiguous block of memory, an effective strategy is to *divide the process's address space into smaller chunks*, allowing noncontiguous allocation. Two primary techniques for accomplishing this are:

- **Paging:** The address space is split into *fixed-size pages*, which map onto equally sized *physical frames*. This approach can reduce external fragmentation but can introduce *internal fragmentation* if a process does not fully use the last frame of its allocation. Paging is straightforward to manage and scales well for large address spaces.
- **Segmentation:** The address space is divided into *variable-sized segments* (e.g., code, data, stack). This fits naturally with the logical structure of programs and can minimize *internal* waste; however, it can result in *external* fragmentation when segments cannot fit into available gaps in physical memory.



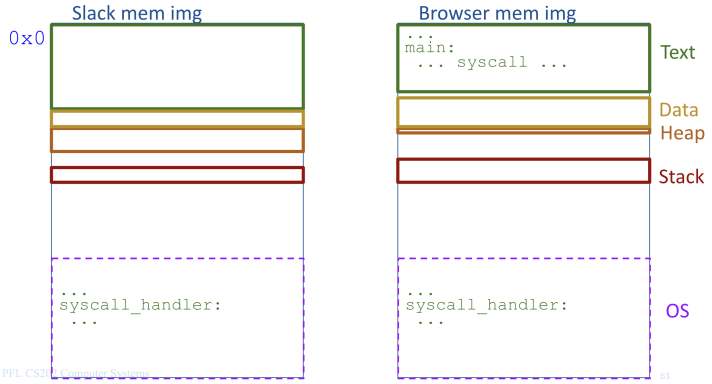
In **paging**, the MMU maintains a *page table* to translate from virtual pages to fixed-size physical frames:



In either scheme, the MMU—*configured* by the kernel with base, bound, or other address-translation structures—ensures each process can access only the memory it has been allocated. This *hardware-based* translation mechanism preserves system safety and helps improve physical memory utilization by allowing noncontiguous allocation.

4.4 Optional - Operating System Mapping in Process Memory

In modern operating systems, the OS is mapped into every process's virtual address space. This design allows a process to make system calls efficiently, as the CPU switches to pre-mapped high-address instructions during such transitions. This integration supports secure and fast interactions between user applications and system-level functions.

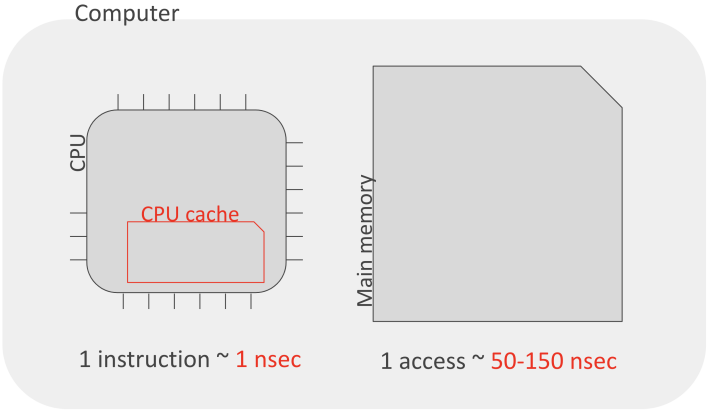


4.5 CPU Caching and Memory Hierarchy

Efficient computation in modern CPUs relies on a well-designed memory hierarchy that mitigates the performance gap between the processor and main memory. Central to this hierarchy is the CPU cache, which stores recently and frequently accessed data.

4.5.1 Overview of CPU Cache

The CPU cache is a small, high-speed memory located close to the processor core. It temporarily holds data and instructions that the CPU is likely to reuse, significantly reducing the latency compared to fetching data from main memory. This approach minimizes delays due to the slower speed of main memory and ensures smoother processor performance.



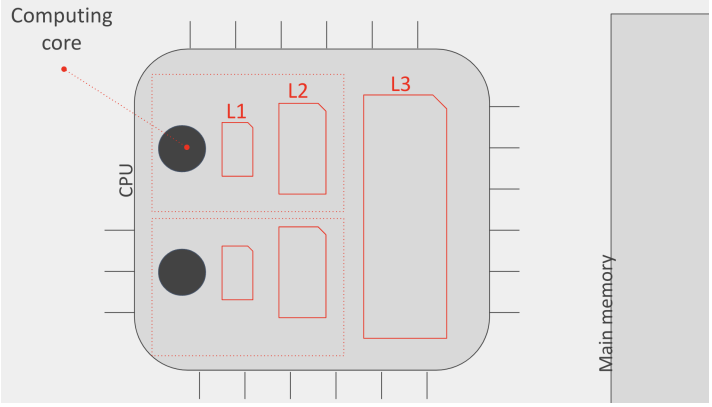
4.5.2 Multi-Level Cache Architecture

Modern CPUs employ a multi-level cache system to balance speed and storage capacity:
Definition (Cache Levels). *The cache hierarchy typically consists of:*

- ***L_1 Cache:*** *The smallest and fastest cache, often divided into separate instruction and data caches.*
- ***L_2 Cache:*** *Larger than L_1 and slightly slower, serving as an intermediary between L_1 and L_3.*

- ***L_3 Cache:*** *The largest and slowest cache, usually shared among multiple cores in multi-core processors.*

The arrangement from smaller and faster (L_1) to larger and slower (L_3) reflects a deliberate trade-off between speed and capacity.



4.5.3 Cache Organization in Multi-Core Processors

Today's processors often include multiple computing cores, each with dedicated L_1 and L_2 caches while sharing a common L_3 cache. This design:

- Provides high-speed access to data for individual cores.
- Balances the overall workload by reducing contention for shared resources.

Without such a hierarchical system, a single cache (e.g., L_1) might evict infrequently used yet critical instructions, thereby degrading performance.

Example 4.5.3.1. *Consider a scenario in which a core with only an L_1 cache continuously evicts a seldom-used, but vital instruction. The presence of additional cache levels (L_2 and L_3) provides extra storage layers, ensuring that even infrequently accessed data remains available when needed.*

4.5.4 Summary of the Memory Hierarchy

The overall memory hierarchy in a modern CPU is structured as follows:

1. **L_1 Cache:** Fastest, smallest, with separate instruction and data caches.
2. **L_2 Cache:** Intermediate in both size and speed.
3. **L_3 Cache:** Largest, slowest, shared among cores.
4. **Main Memory:** Accessed only when data is not found in any cache.

The CPU always accesses the memory hierarchy starting at the fastest level (L_1) and moving downward, ensuring that processing is carried out as efficiently as possible.

Chapter 5

L5 - Paging

Fellow syscoms, this is yet another chapter that was already studied in computer architecture, however, don't be discouraged, I'll do my best to make things clear. Again, please send me a text if anything lacks clarity, with that said, good luck.

For context, we've established that a specialized hardware was required to efficiently translate virtual addresses into physical addresses. However, this introduces another critical requirement: optimizing memory usage. To address this, paging becomes essential, enabling dynamic memory allocation and effectively managing address space constraints.

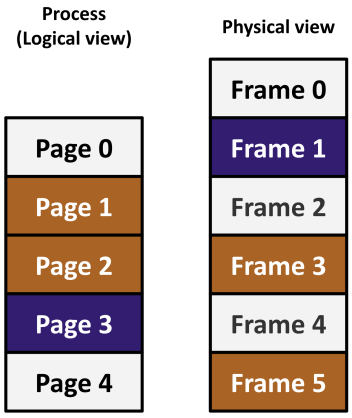
5.1 Page-based Memory Management Unit (MMU)

In modern operating systems, paging is used to manage memory efficiently by dividing both the virtual address space and physical memory into fixed-size blocks. This section provides a detailed overview of how paging works and how addresses are translated.

5.1.1 Overview of Paging

Personal Remark: Although the physical frames allocated to a process may be non-contiguous, the virtual address space appears contiguous to the process.

- **Pages:** Fixed-size blocks that partition the virtual address space.
- **Frames:** Fixed-size blocks that partition physical memory.
- **Mapping:** Each page is associated with a frame via a mapping (i.e., {page → frame}). This allows the operating system to apply protection at the page level.

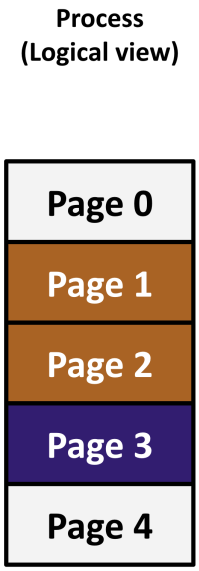


5.1.2 Size of a Page

A page is the smallest unit of memory allocation in a paging system. Its size is chosen based on the following considerations:

- **Minimizing Internal Fragmentation:** Typical page sizes range from 4 KB to 16 KB.
- **Management Overhead:** Smaller pages lead to larger page tables, while larger pages can waste memory.

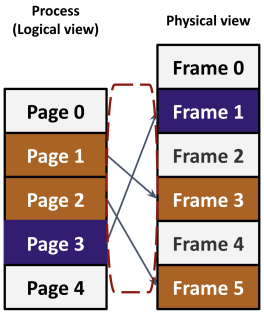
Super Pages: These are larger blocks made up of multiple contiguous pages (e.g., 2 MB or 1 GB). They reduce the overhead associated with page translation.



5.1.3 Memory Management Scheme

The operating system manages memory using the following scheme:

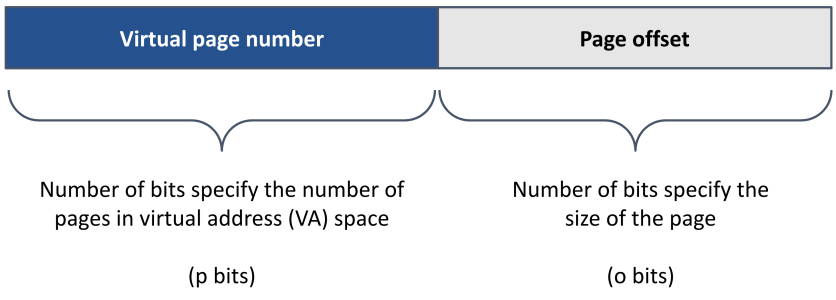
1. **Frame Allocation:** Logical pages are mapped to available physical frames based on the OS's allocation strategy.
2. **Page Table:** The OS maintains a data structure called the page table, which stores the mapping between logical pages and physical frames.
3. **Per-Process Management:** Each process has its own page table to manage its virtual address space.



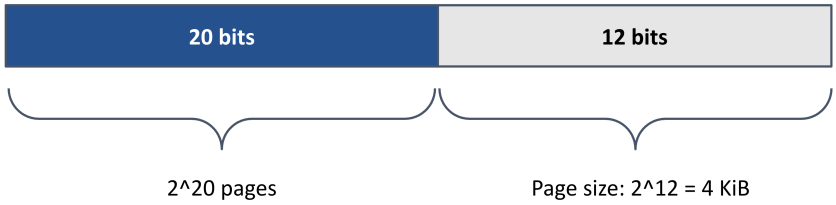
5.1.4 Address Representation

A virtual address is composed of two distinct components:

1. **Virtual Page Number (VPN):** The higher-order p bits of the address that identify the page in the virtual address space.
2. **Page Offset:** The lower-order o bits that specify the exact byte within the page.



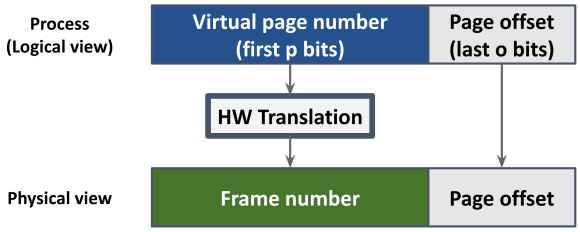
Example 5.1.4.1 (Virtual Address Example (32-bit Architecture))**.** *In a 32-bit system, the virtual address is divided into a virtual page number and a page offset, as illustrated below.*



5.1.5 Address Translation

Address translation is the process by which the Memory Management Unit (MMU) converts a virtual address into a physical address. The steps involved are:

1. **Extract the Virtual Page Number:** Take the first p bits of the virtual address.
2. **Map to Physical Frame:** Use the page table to find the corresponding physical frame number.
3. **Extract the Page Offset:** Take the remaining o bits.
4. **Compute the Physical Address:** Combine the frame number with the offset to access the specific byte in physical memory.



Accessing a Byte

To access a specific byte in memory, the MMU follows these steps:

1. Extract the virtual page number from the virtual address.
2. Map this virtual page number to the corresponding physical frame using the page table.
3. Extract the offset from the virtual address.
4. Access the byte at the calculated physical memory location.

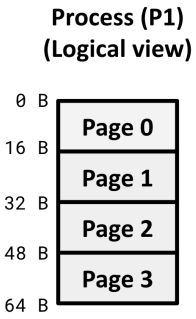
Personal Remark: This systematic approach to address translation is fundamental to the operation of virtual memory systems, ensuring efficient and secure memory access.

5.1.6 Virtual Address Space

Example 5.1.6.1 (Virtual Address Space)**.**

Consider a virtual address space consisting of 64 bytes, divided into 4 fixed-size pages of 16 bytes each. Assume all components of a program (code, stack, heap) comfortably fit into this address space.

Question: *What is the size of a pointer necessary to uniquely address any byte in this address space?*
Answer: *6 bits, since* $\log_2(64) = 6$ *bits are needed to uniquely represent each byte.*



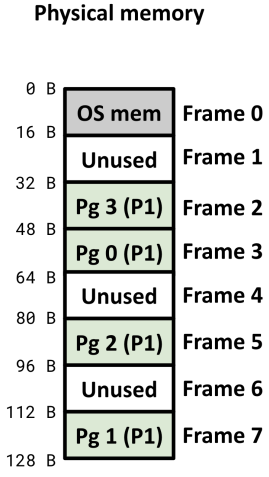
5.1.7 Physical Memory

Example 5.1.7.1 (Physical Memory)**.**

Physical memory is composed of fixed-size storage slots called page frames. Suppose there are 8 page frames, each 16 bytes, making the total physical memory 128 bytes. This setup requires at least 7 bits to uniquely represent any physical memory location ($\log_2(128) = 7$ *bits).*

The virtual pages of a process (e.g., process P1) map to physical memory frames as follows:

- *Virtual page 0 → Physical frame 3*
- *Virtual page 1 → Physical frame 7*
- *Virtual page 2 → Physical frame 5*
- *Virtual page 3 → Physical frame 2*



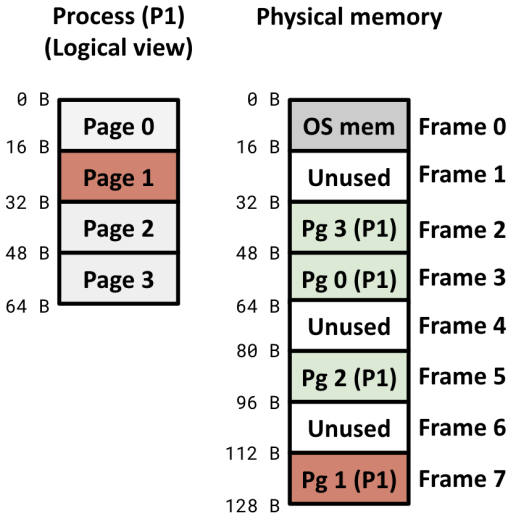
5.1.8 Virtual Address Translation

Example 5.1.8.1 (Virtual Address Translation)**.**

Consider that process P1 attempts to access a memory location using the following assembly instruction:

```
movl 21, %eax
```

This instruction moves 4 bytes starting from virtual address 21 into the %eax register. However, the data does not physically reside at virtual address 21; instead, it is stored in physical memory. Specifically, virtual address 21 belongs to virtual page 1, which maps to physical frame 7:



Example 5.1.8.2 (Computing Virtual Page Number and Offset)**.**

Given:

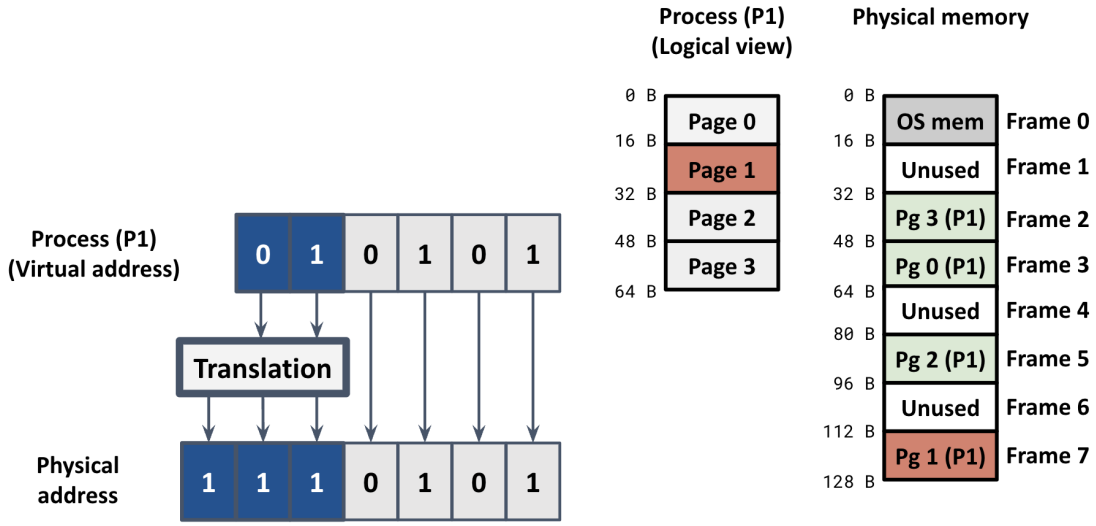
- *Virtual address space: 64 bytes (6 bits)*
- *Page size: 16 bytes (4 bits for offset)*

Thus, the remaining $6 - 4 = 2$ *bits represent the virtual page number.*

Question: *Determine the virtual page number and offset for the instruction* `movl 21, %eax`*.*
The binary representation of 21 is `010101`*. Thus:*

- *Virtual page number: first 2 bits (01)* $\rightarrow$ *Page 1*
- *Offset: last 4 bits (0101)*

Given the page table mapping, virtual page 1 corresponds to physical frame 7 (binary `111`*):*



5.2 The Page Table

Definition (Page Table).
A ***page table*** *is a data structure maintained by the operating system that stores the mapping between virtual addresses and their corresponding physical addresses. Each process has its own dedicated page table.*
The pointer to the currently active page table is stored in a special register known as the page-table base register (PTBR). On x86 architectures, this register is typically referred to as `%cr3`*. During context switches, the operating system saves and restores the PTBR value from the process control block (PCB).*

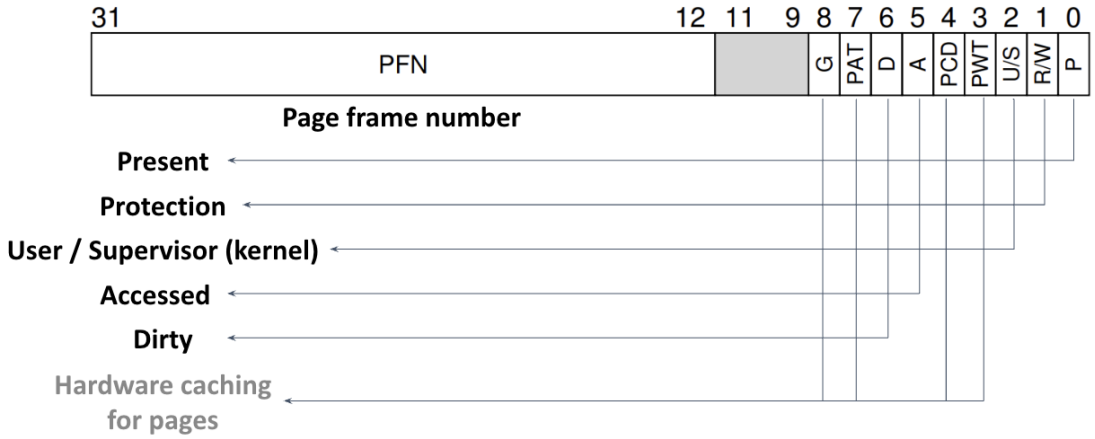
5.2.1 Structure of Page Table Entries

The page table consists of multiple **page table entries** (PTEs). Each entry stores not only the *page frame number* (PFN) that provides the mapping between virtual pages and physical frames but also additional status information. Important fields in a PTE typically include:

- **Present bit**: Indicates if the translation is valid and the page resides in physical memory.
- **Protection bits**: Define access permissions (read, write, execute).

- **User/Supervisor bit (U/S)**: Differentiates between user-mode and kernel-mode access permissions.
- **Dirty bit**: Indicates if the page has been modified (written to).
- **Access/Reference bit**: Used to track page usage patterns and inform page replacement algorithms.

Below, the 32-bit Intel PTE format



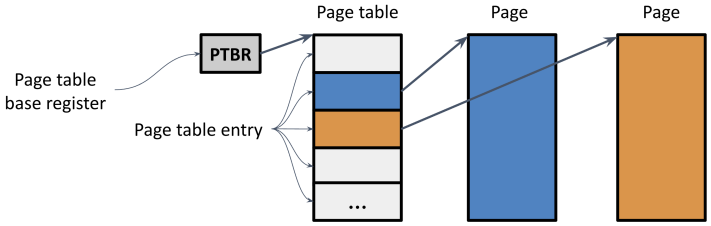
Additionally, each page table entry is always aligned to the page size for efficient access by the hardware.

5.3 Organizing the Page Table Structure

The simplest implementation is the **linear page table**. The Memory Management Unit (MMU) indexes directly into the page table using the *virtual page number*.
The process involves the following steps:

1. The MMU uses the virtual page number as an index into the linear array of page table entries.
2. It retrieves the corresponding *page table entry (PTE)* at this index.
3. From the PTE, the MMU obtains the associated *physical frame number*.

A linear page table requires the allocation of multiple contiguous memory pages to store the entire mapping structure.



Example 5.3.0.1 (Size of a Linear Page Table)**.** *Consider the following assumptions:*

- *Virtual address size: 32 bits*
- *Physical address size: 32 bits*
- *Page size: 4 KB (i.e., 12 bits offset)*
- *Each page table entry (PTE): 4 bytes*

The size of the linear page table is calculated as follows:

$$\textit{Number of entries} = 2^{\textit{Virtual Address Bits} - \textit{Offset Bits}} = 2^{32-12} = 2^{20}$$

Thus, the page table size is:

$$\textit{Page Table Size} = \textit{Number of Entries} \times \textit{Size per Entry} = 2^{20} \times 4 \textit{ bytes} = 4 \textit{ MiB}$$

5.3.1 Resolving addresses with a Linear Page Table (32-bit)

In a 32-bit linear paging system, the address translation process from virtual to physical addresses follows a straightforward mechanism, detailed step-by-step below and illustrated bellow.

1. **Access Address Breakdown**

   The virtual (logical) address consists of two parts
   (a) **Page number** identifies the page in virtual memory.
   (b) **Offset** identifies the byte within the page.
   For example, the address `20 983 809` can be split into:
   - Page number `5123`
   - Offset `1`

2. **Lookup in Page Table**

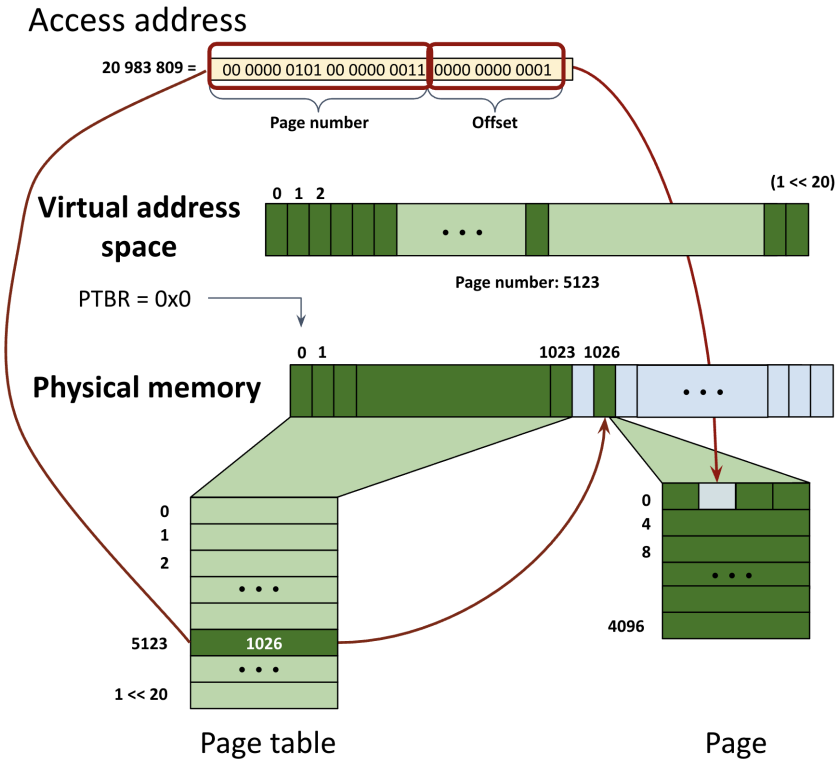
   - The Page Table Base Register (PTBR) points to the beginning of the page table in physical memory (in our example, `PTBR = 0x0`).
   - The page number extracted from the virtual address (`5123`) is used as an index into the linear page table.

3. **Page Frame Identification** The entry at index `5123` in the page table provides the frame number in physical memory (in the diagram example, frame number `1026`).

4. **Physical Address Computation**

   - The physical frame number (`1026`) obtained from the page table and the original offset (`1`) from the virtual address are combined to form the physical address.
   - This address directly maps to a unique location in physical memory.

5. **Accessing the Memory** Finally, the computed physical address is used to access the desired data in physical memory.



This linear mapping approach is simple and direct but may require substantial memory for the page table, especially when handling large virtual address spaces, let's see how much.

5.3.2 The Issue with Linear Page Tables (4 KB Pages)

Using a linear page table with a 32-bit architecture, a 4 KB (12-bit offset) page size, and 4-byte entries results in a considerable memory overhead:

$$\text{Number of entries} = 2^{32-12} = 2^{20}$$

$$\text{Page table size} = 2^{20} \times 4\,\text{bytes} = 4\,\text{MiB}$$

Expanding this to 64-bit architectures significantly increases memory usage:

Virtual Address Bits	Physical Address Bits	Entry Size	Page Table Size
32	48 bit	8 bytes	8 MiB
64	64 bit	8 bytes	$2^{52} \times 8$ bytes = 32 PiB

In 64-bit architectures with large, sparse address spaces, linear page tables quickly become impractical. Although increasing the page size (e.g., from 16 KiB pages) can reduce memory overhead, it introduces significant internal fragmentation. Let's look at a more effective approach

5.3.3 Multi-level Page Tables

Most processes use only a small fraction of the available address space. Multi-level page tables efficiently allocate metadata only for the used portion by organizing the page table in a hierarchy. Although each level adds an extra memory lookup during address translation, this method saves space overall.

Analogy Imagine locating a book in a library: first, you choose a section, then an aisle, then a shelf, and finally the book's position.

Analogy 2 - This made me understand how multi-level paging is more memory efficient
Imagine organizing a large library.

With a **linear (single-level) page table**, you'd have to create a catalog entry for *every shelf*—even if most shelves are completely empty. This wastes space by reserving entries for unused shelves.

A **two-level page table** solves this problem by using a hierarchical catalog: the *first-level catalog* only keeps track of sections of shelves that actually contain books. Detailed *second-level catalogs* are created solely for these used sections, avoiding unnecessary records for empty shelves and significantly reducing memory usage.

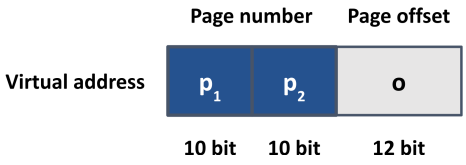
Example 5.3.3.1 (Two-Level Paging Example)**.** *A virtual address on a 32-bit machine with a 4 KiB page size is divided as follows:*

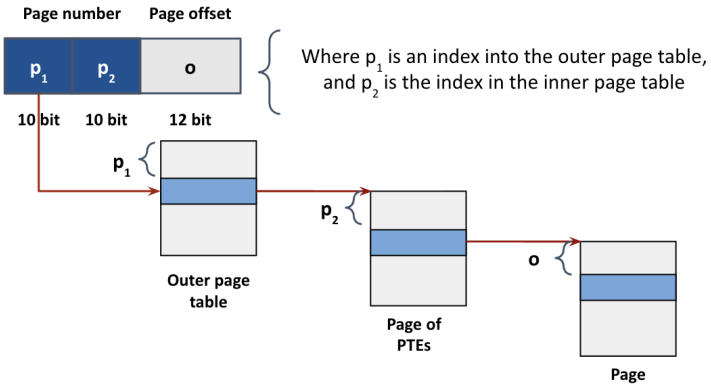
- ***Page offset:*** *12 bits.*
- ***Page number:*** *20 bits.*

Since the page table is paged, the 20-bit page number is split into two 10-bit parts:

- *The first 10 bits index the first-level page table.*
- *The next 10 bits index the second-level page table.*

With each page table entry occupying 4 bytes, a 4 KiB page can hold up to $\frac{4096}{4} = 1024 \quad (or\ 1 \ll 10) \quad entries.$





5.3.4 Resolving Addresses: Linear vs. Two-Level Paging (32-bit)

In a *linear* (single-level) paging scheme for a 32-bit address space, the virtual address is typically split into two parts:

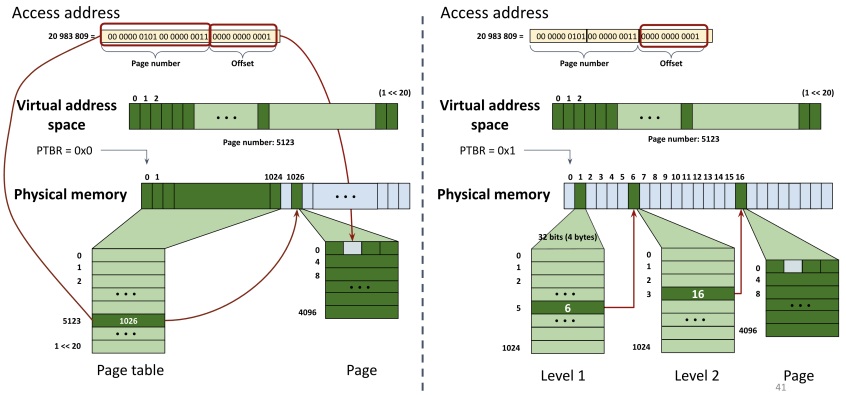
$$\underbrace{\text{Page Number}}_{20\text{ bits}} \quad \underbrace{\text{Offset}}_{12\text{ bits}}.$$

The *page number* serves as an index into a single page table, which holds the base address of the corresponding physical page. The *offset* is then added to this base address to obtain the final physical address.
By contrast, in a *two-level paging* scheme, the 20-bit page number is further subdivided into:

$$\underbrace{\text{Page Directory Index}}_{10\text{ bits}} \quad \underbrace{\text{Page Table Index}}_{10\text{ bits}} \quad \underbrace{\text{Offset}}_{12\text{ bits}}.$$

The top 10 bits (page directory index) point to an entry in the *page directory*, which in turn identifies the base address of a particular *page table*. The next 10 bits (page table index) select the entry in that page table, which gives the base address of the physical page. Finally, the offset is added to this base address to compute the physical address.

The main difference is that *linear paging* uses a single, large page table for the entire virtual address space, whereas *two-level paging* uses a hierarchy of smaller tables.



5.3.5 Multi-level Page Table for 64-bit Addressing

In systems with a 4 KiB page size (4096 bytes), each page holds 512 page table entries because

$$4096 \text{ bytes} \div 8 \text{ bytes/entry} = 512 \text{ entries} \quad (\text{requiring 9 bits since } 2^9 = 512).$$

For a full 64-bit address, subtracting the 12 bits used for the page offset leaves 52 bits to be mapped. A common scheme uses five levels of page tables for the first 45 bits (5 levels $\times$ 9 bits each) and a sixth level for the remaining 7 bits. If the virtual address space is reduced:

- To 57 bits: $57 - 12 = 45$ bits remain, which can be mapped with 5 levels.
- To 48 bits: $48 - 12 = 36$ bits remain, requiring 4 levels.

5.3.6 Paging: Advantages and Disadvantages

Advantages:

- *No external fragmentation:* Memory is allocated in fixed-size pages.
- *Fast allocation and deallocation:* Pages can be allocated or freed without searching for a contiguous memory block.

Disadvantages:

- *Memory overhead:* Additional space is required to store the page tables.
- *Increased memory accesses:* Each memory access may require extra references to the page tables.
- *Hardware complexity:* Efficient address translation demands specialized hardware (eg. we'll look at that hardware in the next section)

5.3.7 Logical Process of Memory Access in a Paging System

When the CPU requests a code or data value at a virtual address, the following steps occur:

1. The Memory Management Unit (MMU) begins a page table walk starting from the Page Table Base Register (PTBR).
2. Depending on the address space:
   - A 32-bit address may require 2 memory references for translation.
   - A 48-bit address may require up to 4 memory references.
3. After the page table lookup, the page offset is added to the translated address to access the actual data in physical memory.
4. To reduce the translation overhead, a cache called the Translation Lookaside Buffer (TLB) is used to store recent virtual-to-physical address mappings.

5.4 Translation Lookaside Buffer (TLB)

Seen in a comparch, I'll try to make this clear.
The Translation Lookaside Buffer (TLB) is a specialized, hardware-based cache that stores recent mappings from virtual addresses to physical addresses. When a process accesses memory, the Memory Management Unit (MMU) first checks the TLB:

- **TLB Hit:** If the mapping is present, the physical address is obtained directly, minimizing delay.
- **TLB Miss:** If the mapping is absent, the MMU must perform a page table walk, involving multiple memory accesses, which is significantly slower.

The effectiveness of the TLB is largely due to the principle of locality of reference, which ensures a high hit rate under typical workloads. However, TLB entries can become invalid after a context switch or when page tables are updated.

5.4.1 Memory Access Cost

Assume a 64-bit address space and that all page table levels are cached when the TLB is present (i.e., on a TLB hit). The following table illustrates the number of memory accesses required to read or write a memory location X for a process:

Page Table Level	**With TLB & TLB Hit**	**Without TLB**
No Paging	1	1
1 Level	1	2
2 Level	1	3
3 Level	1	4

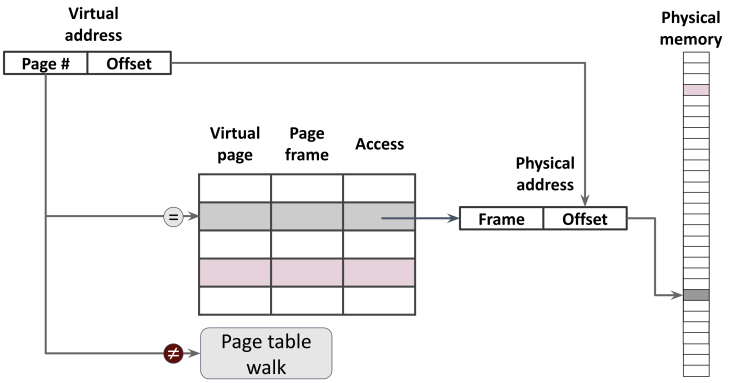
Key: The TLB is implemented as a dedicated circuit, separate from main memory, enabling rapid address translation.

5.4.2 TLB Lookup Process

A Translation Lookaside Buffer (TLB) is a small, fast cache that stores recent virtual-to-physical address translations to speed up memory accesses.

1. **Decompose the virtual address:** The processor splits the virtual address into two parts:
   - *Virtual Page Number (VPN)*: The high-order bits used for indexing or tagging in the TLB.
   - *Offset*: The low-order bits that remain unchanged when forming the physical address.
2. **Check the TLB:** The VPN is compared against the *tag* fields of all TLB entries (often in parallel, if the TLB is fully associative). If a matching tag is found, it indicates a potential translation match.
3. **Validate the match:** Along with the tag comparison, each TLB entry includes *valid* and possibly other *protection* bits. The processor checks these bits to ensure:
   - The entry is valid (i.e., not stale or invalidated).
   - The access permissions allow the requested operation (read, write, or execute).

   If these checks pass, the match is confirmed.
4. **Obtain the physical frame number (PFN):** Upon a valid match, the TLB entry provides the corresponding *Physical Frame Number* (PFN). This is combined with the original offset to form the final physical address.
5. **Handle TLB misses:** If no valid TLB entry matches the VPN (Virtual Page Number):
   - The hardware (or the operating system, depending on the architecture) performs a *page table walk* to locate the correct PFN in the page table.
   - The discovered translation may then be loaded into the TLB for future accesses.
   - The instruction or memory operation is retried with the updated TLB entry.



This process allows the CPU to translate virtual addresses into physical addresses quickly by leveraging the TLB's cached entries, significantly reducing the average memory access time.

5.4.3 CPU Execution of a Read/Write Operation

1. The CPU issues a load operation for a given virtual address (as part of a memory load/store).
2. The Memory Management Unit (MMU) checks the Translation Lookaside Buffer (TLB) for the virtual address.
3. **TLB Miss:**
   - The MMU performs a page walk through the page table.
   - If the Page Table Entry (PTE) is not present, a page fault occurs; the OS is invoked and a segmentation fault may be raised.
   - If the PTE is present, the TLB is updated and execution continues.
4. **TLB Hit:** The physical address is obtained from the TLB, the memory location is fetched, and the data is returned to the CPU.

5.4.4 Summary: Page Tables

- **Contents:** Page Table Entries (PTEs) that include permission bits.
- **Size:**
  - *Linear Page Table:* Can be very large.
  - *Multi-Level Page Table:* More memory efficient when sparsely populated.
- **Performance:** Paging overhead is mitigated by the use of the TLB.
- **Memory Exhaustion:** Appropriate measures must be taken when the system runs out of memory.

5.5 Swapping: Managing Memory Shortages

When the physical memory is insufficient to hold all the active processes, the operating system (OS) employs a mechanism called *swapping*. This process involves temporarily moving pages that are not actively used from main memory to disk storage. By doing so, the OS can reclaim memory for processes that require immediate attention and even over-provision memory beyond the available physical resources.

5.5.1 Concepts

- **Working Set:** The collection of pages a process actively uses at a given time. This set can change dynamically as the process executes.
- **Storing Unused Pages:** By transferring inactive pages to disk, the OS frees up main memory for active processes.
- **Over-Provisioning:** Swapping allows the system to allocate more virtual memory than is physically available.

5.5.2 Swapping In: Handling Page Faults

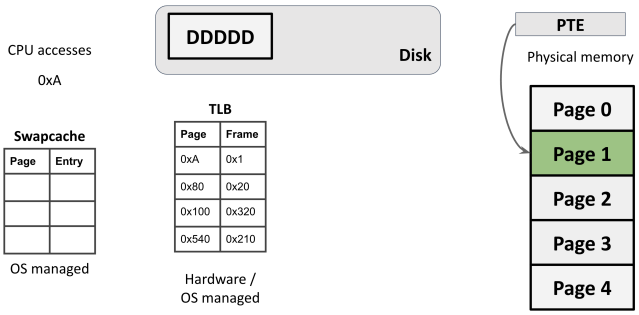
When a process accesses a page that is not present in main memory, the Memory Management Unit (MMU) cannot translate the virtual address because the corresponding page table entry indicates that the page is absent. This situation results in a *page fault*, prompting the OS to bring the page back from disk (swap-in).

Page Fault Handling Procedure

1. **Address Translation:** The MMU translates virtual addresses to physical addresses using page tables. Each page table entry has a *present bit* indicating if the page is in memory.
2. **Page Fault Occurrence:** If the present bit is unset, a page fault is triggered.
3. **Identifying the Fault:** The OS determines which process and address caused the fault by consulting its data structures.
4. **Determining Page Status:**
   - If the page is on disk, the OS issues a request to load it into memory.
   - If the page has not been swapped out, the OS creates the mapping and updates its data structures.
5. **Context Switching:** While waiting for disk I/O, the OS may switch to another process.
6. **Resuming Execution:** Once the page is loaded, the OS updates the page table entry and the Translation Lookaside Buffer (TLB), and then resumes the faulting process.

Swap-In Procedure

1. Locate the page in the swap cache.
2. Allocate a new page in memory.
3. Copy the content from disk to the allocated page.
4. Update the page table entry to indicate that the page is now in memory.
5. Load the corresponding TLB entry.

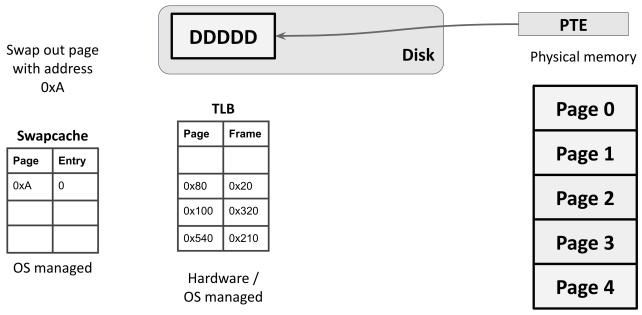


5.5.3 Swapping Out: Freeing Up Memory

Swapping out is the process of moving pages from main memory to disk, thereby freeing up physical memory for active processes. Although the steps involved are straightforward, choosing which pages to swap out is governed by complex OS policies to avoid performance degradation (e.g., thrashing).

Swap-Out Procedure

1. Invalidate the corresponding TLB entry.
2. Allocate an entry in the designated swap space.
3. Copy the page content from memory to disk.
4. Update the page table entry to reflect that the page is now stored on disk.
5. Release the physical memory page.



5.5.4 Conclusion

Swapping is a vital mechanism in memory management, enabling the OS to manage limited physical memory efficiently by transferring inactive pages to disk. This dynamic exchange between main memory and disk ensures that active processes have the resources they need while also allowing the system to support a larger number of processes. However, careful policy decisions are essential to minimize overhead and avoid issues such as thrashing.

Chapter 6

File System I

Definition (Persistence). *In computer science,* persistence *refers to the property of a system's state that remains available beyond the lifetime of the process that created it. In practical terms, persistent data is stored on non-volatile media—such as hard disks or solid-state drives—so that it is not lost when the system powers down.*
This concept is fundamental because, without persistence, all data would reside solely in volatile memory (RAM) and would be lost upon shutdown or power failure.

6.1 Purpose and Functionality of a File System

A file system is tasked with managing a set of persistent storage blocks provided by a storage device. Its design addresses several key objectives:

- **Efficient Data Management:** Organize and manage data on non-volatile storage.
- **File Manipulation:** Allow users and applications to create, name, and manipulate semi-permanent files.
- **Metadata Organization:** Maintain associated metadata (e.g., ownership, permissions, file types) to facilitate file management.
- **Resource Sharing and Access Control:** Enable file sharing among multiple users and processes while enforcing security restrictions.

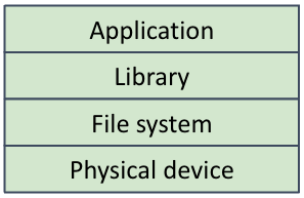
6.2 I/O Operations and File System Layers

File system operations are mediated by a layered architecture that abstracts the complexity of hardware interactions. This section outlines the main layers and their roles.

6.2.1 Layered Architecture Overview

The process of reading from or writing to a file involves multiple layers, each with distinct responsibilities:

- **Application Layer:**
  - Applications require reading and writing data.
  - They invoke standardized, operating system–independent library functions, such as `fopen`, `fread`, `fwrite`, and `fseek`.
  - These functions work with `FILE *` streams and offer buffering capabilities (e.g., via `setvbuf`) to optimize I/O operations.
- **System Call Interface:**
  - Between the high-level libraries and the file system lies the operating system's system call interface.
  - Functions such as `open`, `read`, `write`, and `lseek` are used at this level.
  - These calls operate on file descriptors and serve as the bridge to the file system.
- **File System Layer:**
  - The file system interprets the system calls and translates them into specific operations on the physical storage device.
  - This layer is responsible for managing the underlying persistent blocks efficiently.



Man Pages

We'll take a further look at this during the project's warmup
Man pages, short for manual pages, are built-in documentation for Unix-like operating systems, providing detailed information about commands, system calls, functions, and files. Each man page typically includes a synopsis, description, options, usage examples, and related commands.
To access a man page, use the command:

```
man [section] command
```

For instance, the command:

```
man fread
```

shows documentation for the `fread` function, typically in the default section. However, since the same name may exist in multiple sections, specifying a section number can be necessary:

```
man 3 fread
```

explicitly requests documentation from section 3, which covers library functions (part of `libc`). `fread` and other `FILE*` calls offer benefits such as portability across operating systems and higher-level abstractions like buffering.
On the other hand, system calls, such as:

```
man 2 read
```

provide lower-level interfaces. The `read` system call, documented in section 2, directly uses file descriptors, allowing the same code to interact uniformly with files, pipes, and sockets (covered further in networking lectures).

Example 6.2.1.1 (Example: Reading file contents in C (simplified `cat`))**.** This example illustrates how to implement a simple version of the `cat` command in C. It includes argument handling (`argv`), file I/O operations, writing to standard output (`stdout`), and briefly mentions how the tool `strace` can be used to trace system calls.

```c
/*cat -- simplified -- */
#include <stdio.h>
#include <stdlib.h>

#define BUFSIZE (32*1024) // 32 KB buffer

int main(int argc, char *argv[]) {
    FILE *finput;
    char buf[BUFSIZE];
    size_t bsize;

    if (argc > 1) {
        // Open file in read-only mode
        finput = fopen(argv[1], "r");
        if (!finput) {
            perror("fopen");
            return 1;
        }

        do {
            bsize = fread(buf, 1, BUFSIZE, finput);
            if (bsize > 0)
                fwrite(buf, 1, bsize, stdout);
        } while (bsize == BUFSIZE);

        fclose(finput);
    } else {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        return 1;
    }

    return 0;
}
```

Compilation and Redirection
To compile and run the simplified `cat` program:

```
cc cat.c -o cat
./cat myfile.txt > output.txt
```

This illustrates the use of the indirection operator (`>`), redirecting the program's standard output to the file `output.txt`.

System Call Tracing (`strace`)
You can trace system calls using `strace`:

```
strace ./cat myfile.txt > output.txt
```

This will display all the system calls (such as `open`, `read`, and `write`) that the simplified `cat` program makes.

6.3 File System Goals and Core Components

A file system provides reliable, long-term storage of information and is responsible for managing how data is stored and retrieved on secondary storage devices. Its primary goals are:

- **Persistent Storage**: Data remains stored across reboots and system shutdowns.
- **Concurrent Access**: Allows simultaneous reading and writing by multiple processes.
- **Human-Readable Naming**: Facilitates logical naming and organization of data for ease of access.

The file system comprises two fundamental components:

1. **Files**
2. **Directories**

6.3.1 Defining a File

Definition (File). *A file is a named, persistent collection of related information stored on secondary storage, represented as a linear array of bytes.*
A file consists of two distinct components:

- **Data**: The content provided by users or applications, structured as a linear array of bytes.
- **Metadata**: Auxiliary information managed by the operating system, including attributes such as:
  - Size
  - Ownership and permissions
  - Modification and access timestamps

6.3.2 Perspectives on Files

Files can be understood from three perspectives:

1. **User Perspective (Human-readable paths)**
2. **Operating System Perspective (Inode identification)**
3. **Process Perspective (File descriptors)**

6.3.3 User View: File Names

From a user's perspective, files are identified using meaningful, human-readable names. Files are organized hierarchically within directories, allowing intuitive and logical structuring.

Definition (File Path). *A file path describes the location of a file within the hierarchical directory structure, uniquely identifying it within the entire file system.*

- *Filenames are unique within their local directory.*
- *Full paths, however, provide globally unique identification.*

File Content and Types

Modern file systems primarily handle files as **untyped sequences of bytes**. The content interpretation is left entirely to the user applications; the OS neither understands nor manages content semantics.

6.3.4 Operating System View: Inodes

Definition (Inode). *An inode (Index Node) is a low-level persistent data structure maintained by the file system. It contains metadata about a file and pointers to the data blocks on disk.*

Characteristics of inodes

- Each file is associated with exactly one inode.
- Inode IDs are unique within the file system but not globally.
- After deletion, inode numbers can be recycled and reassigned.

Metadata stored in an inode

- File permissions and ownership
- File size
- Timestamps (creation, modification, and access times)
- Pointers to the actual data blocks and indirect blocks on storage media

Management of Inodes by File Systems

The file system dedicates a specific area of the disk known as the **inode table**, analogous to a linear page table, to manage inodes:

- The disk is partitioned into two distinct regions: one for inode tables and another for data blocks.
- Each inode has a fixed, unique location within this inode table.
- The inode number directly identifies the location of an inode on disk.

Typically, inode tables reside in a reserved area at the beginning of the storage medium.

6.3.5 Mapping Paths to Inodes

File systems store mappings from human-readable filenames to inodes within special files known as **directories**.

Definition (Directory).

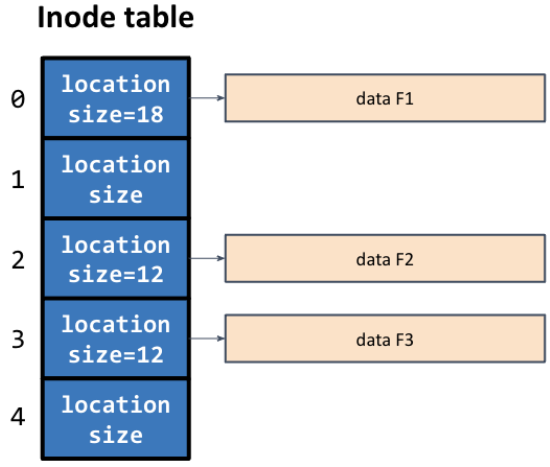
A directory is a special type of file containing an array of mappings between filenames and inode numbers.

Example 6.3.5.1 (Resolving a path)**.**

Resolving a path to its corresponding inode involves sequentially traversing directories:

Accessing the file `/tmp/test.txt` *involves three steps*

1. *Locate the inode of directory* `tmp` *in the root directory (*`/`*).*
2. *Within* `tmp`*, find the inode associated with* `test.txt`*.*
3. *Access the file content via the identified inode.*



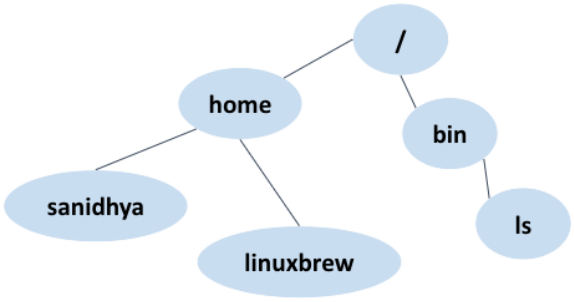
Relationship Between Inodes and Directories

Important distinctions regarding directories and inodes:

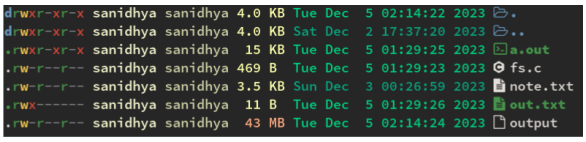
- Directories themselves are stored as regular files but with a special identifying flag.
- This special flag restricts operations (e.g., writing directly to directory files is prohibited).
- Directories contain arrays of pairs {`filename, inode number`}.
- An inode does **not** store its filename; filenames are only stored in directory entries.

6.3.6 Directory Organization

The file system organizes data into directories and files with a hierarchical structure.



- **Root Directory:** Denoted by "/" (typically associated with inode 1).
- **Navigation:**
  - "." refers to the current directory.
  - ".." refers to the parent directory.



- **Permission Bits:** Each file or directory has nine permission characters following a leading type indicator (e.g., "d" for directory or "-" for file):
  - **Owner:** Read, write, and execute (rwx).
  - **Group:** Typically read and execute (r-x).
  - **Others:** Typically read and execute (r-x).
  - For files, the execute bit ("x") indicates that the file is executable.
  - For directories, the execute bit allows users to change into the directory (i.e., using `cd`).

6.3.7 File Referencing via Links

Links provide a means to reference a file by its location or name without duplicating its data. There are two primary types of links:

Hard Links

- Associate an alternative file name directly with the same inode as the original file.
- Serve as mirror copies; both names refer to the same underlying data.
- Deleting one hard link does not remove the actual data as long as another hard link exists.

Symbolic (Soft) Links

- Create a reference by logically mapping a file path to a target file.
- Allocate a new inode for the link.
- If the target file is removed, the symbolic link becomes broken or invalid.

6.3.8 Process View: File Descriptors

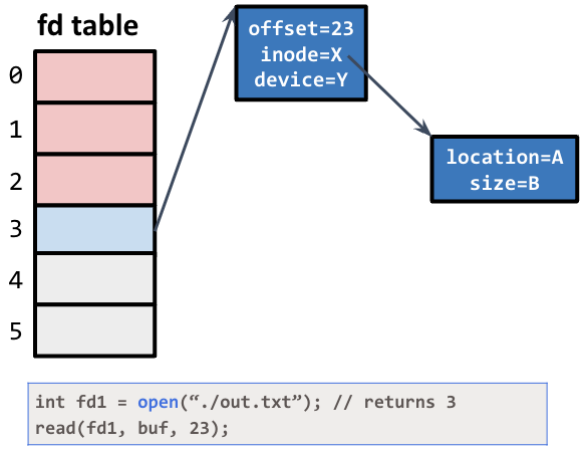
File system operations can be implemented using file names along with inode and device IDs. However, performing a lookup from a file name to its corresponding inode/device ID for every operation can be inefficient.
To overcome this, most systems perform the expensive directory traversal once and then store the resulting inode/device number in a per-process table known as the file descriptor (fd) table.
This table not only holds the inode/device number but also maintains additional information such as the current file offset. File descriptors are represented by small non-negative integers (typically 0, 1, 2, etc.) and are reused when they are freed.

Example 6.3.8.1 (Operations on a File)**.**

Each process maintains its own file descriptor table. The first three descriptors are reserved:

- ***0:*** *Standard Input (STDIN)*
- ***1:*** *Standard Output (STDOUT)*
- ***2:*** *Standard Error (STDERR)*

For example, a file opened by a process might receive file descriptor 3 (with an associated inode, say X). When a read operation is performed, the file offset is updated (e.g., from 0 to 23). If another descriptor (say, file descriptor 4) is assigned to the same file, it starts with its own independent offset (e.g., initialized to 0).



6.4 File System API

The File System API in UNIX-like operating systems provides a set of system calls to manage files programmatically. This section introduces essential operations such as creating, opening, closing, reading, writing, and manipulating file metadata.

Creating and Opening Files

To create or open a file, the `open()` system call is utilized:

```
int open(const char *pathname, int flags, mode_t mode);
```

- `pathname`: Path to the file.
- `flags`: Define the access mode and creation flags.
- `mode`: Set permissions for the file, effective when `O_CREAT` is specified.
- Returns a file descriptor (fd), a non-negative integer used to perform subsequent operations.

Example 6.4.0.1 (Creating and opening a file)**.** *The following code snippet demonstrates creating a file named `"out.txt"` with read, write, and execution permissions for the owner:*

```
int fd = open("./out.txt", O_CREAT | O_RDWR | O_TRUNC, S_IRWXU);
```

Closing Files

Files should be explicitly closed using the `close()` system call to release resources:

```
int close(int fd);
```

- `fd`: File descriptor.
- Returns 0 on success, or -1 on failure.

Reading and Writing Data

The File System API provides two primary system calls to perform I/O operations:

```
ssize_t read(int fd, void *buffer, size_t count);
ssize_t write(int fd, const void *buffer, size_t count);
```

- `fd`: File descriptor.
- `buffer`: Memory area for input/output data.
- `count`: Number of bytes to read/write.
- Both calls return the number of bytes successfully processed.

Managing File Offset

To manipulate the file offset explicitly, use the `lseek()` system call:

```
off_t lseek(int fd, off_t offset, int whence);
```

- `fd`: File descriptor.
- `offset`: Number of bytes to move.
- `whence`: Position from which offset is applied:
  - `SEEK_SET`: Beginning of file.
  - `SEEK_CUR`: Current offset.
  - `SEEK_END`: End of file.
- Returns the resulting offset location, or -1 on error.

Deleting Files

To remove a file, the `unlink()` system call is used:

```
int unlink(const char *pathname);
```

- Removes file entry from the filesystem, reducing its reference count.
- File is physically deleted when its reference count reaches zero.

Synchronizing Data

To ensure that all buffered data is written to disk, the `fsync()` system call is essential:

```
int fsync(int fd);
```

- Flushes file data and metadata from memory to disk.

Accessing File Metadata

File metadata such as permissions, inode number, and timestamps can be retrieved using the `fstat()` system call:

```
int fstat(int fd, struct stat *statbuf);
```

- Populates the provided `statbuf` structure with metadata.
- Information retrieved includes device ID, inode number, permission bits, user ID, etc.
- Returns 0 on success or -1 on failure.

Example 6.4.0.2 (Basic File Operations)**.**

The following program demonstrates a sequence of file operations:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd = open("./out.txt", O_CREAT | O_RDWR | O_TRUNC, S_IRWXU);

    char buffer[20] = {0};

    write(fd, "hello world", 11);      // Writes 11 bytes, offset at 11
    lseek(fd, 0, SEEK_SET);            // Resets offset to beginning

    read(fd, buffer, 5);               // Reads 5 bytes into buffer
    printf("Read data: %s\n", buffer); // Prints "hello"

    close(fd);                         // Closes the file descriptor
    return 0;
}
```

6.5 Mount Points

Mountpoints are directory locations in a filesystem where storage devices or partitions are attached, allowing access to their contents.

6.5.1 Multiple File Systems

A single operating system may contain multiple file systems coming from different sources.

- Different partitions on the same physical disk
- Multiple physical disks
- Removable media such as DVD or Blu-ray drives
- USB flash drives
- Network Attached Storage (NAS)
- Legacy devices such as floppy drives

This raises an important question: how do we organize, manage, and present these diverse file systems to users in a coherent manner?
The solution adopted by modern operating systems is to map all file systems into a single, unified hierarchy rooted at a common point:

- [Windows:] File systems are mapped using drive letters (e.g., `C:\`, `D:\`).
- [Unix/Linux:] File systems can be mounted into any directory, integrating seamlessly into a single hierarchical tree. For instance, the directory `/home` can itself represent a separate file system.

The act of **mounting** integrates multiple distinct file systems across different storage devices into one logical structure accessible via the standard directory tree.

6.5.2 Benefits of Using Mount Points

Using mount points provides several key advantages:

- A unified namespace offering a consistent view of all storage resources.
- Uniform access through the same file system interface and API.

Common commands related to mounting in Unix/Linux environments include:

- `mount <device> <directory>` (general syntax)
- `mount /dev/cdrom /media/cdrom` (example for mounting optical media)
- `mount -t ext4 /dev/sda5 /home` (mounting a specific file system type)
- `df` (reports disk space usage for all mounted file systems)
- `df .` (reports disk space usage for the current file system)

6.6 From File System Abstraction to Implementation

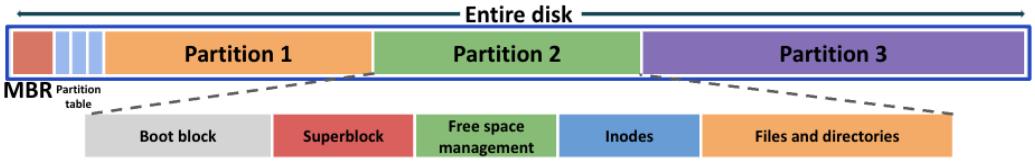
6.6.1 File System Implementation

A file system manages and organizes user data efficiently on storage media. To achieve this, the following elements must be considered:

- **Storage Structure:** Typically represented as a large sequence of N storage blocks.
- **Metadata Organization:** Data structures that clearly encode file hierarchies and individual file metadata.
- **Efficiency Criteria:**
  - Minimize metadata overhead compared to actual file data.
  - Minimize internal fragmentation (unused space within allocated blocks).
  - Provide efficient access to file contents, reducing external fragmentation and metadata access overhead.
- **Implementation of File System APIs:** Offers multiple design choices analogous to virtual memory implementations.

6.6.2 File System Layout on Disk

The file system is physically stored on disk drives, which are typically structured into partitions. The primary layout includes:

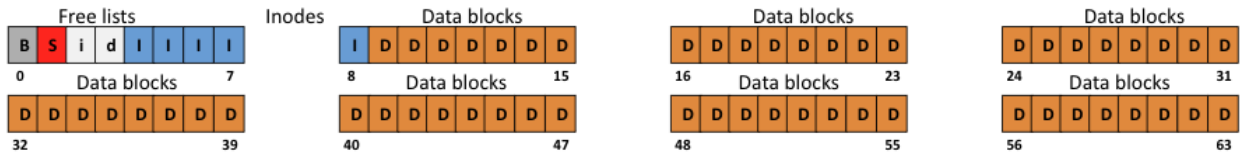


Disk Structure

- **Sector 0 (Master Boot Record - MBR):**
  - Contains bootstrap code executed by firmware during startup.
  - Stores a partition table indicating partition boundaries.
- **Boot Block:** Located at the start of each partition, it contains executable boot code loaded by the MBR.

6.6.3 Detailed View: Inside a Partition

Each partition is structured into a sequential collection of blocks:



Partition Structure

- **Block Organization:** Blocks numbered from 0 to $N - 1$ (e.g., 64 blocks, each of 4KB).
- **Block Types:**
  - *Data Blocks*: Contain actual file content.
  - *Metadata Blocks*: Manage file system structure, including:
    - An array of *inodes* (file descriptors).
      - Example: If an inode size is 256 bytes, each 4KB block can store 16 inodes. Thus, 5 blocks provide space for up to 80 files.
    - Bitmap structures or free lists that track available inodes and data blocks.
- **Boot and Superblock:** Typically placed at the beginning of each partition for initialization and file system configuration.

This layout ensures systematic management of storage, facilitating efficient data retrieval and minimizing fragmentation.

6.6.4 File System Superblock

The file system superblock stores critical metadata describing the structure and organization of the file system. Key characteristics include:

- Exists as one logical superblock per file system.
- Contains essential metadata, such as:
  - Number of inodes
  - Number of data blocks
  - Location of the inode table
  - Information to track free inodes and data blocks
- The first structure read when mounting the file system.

6.6.5 File Inode

An inode is a data structure used to store metadata about an individual file or directory. Typical inode attributes include:

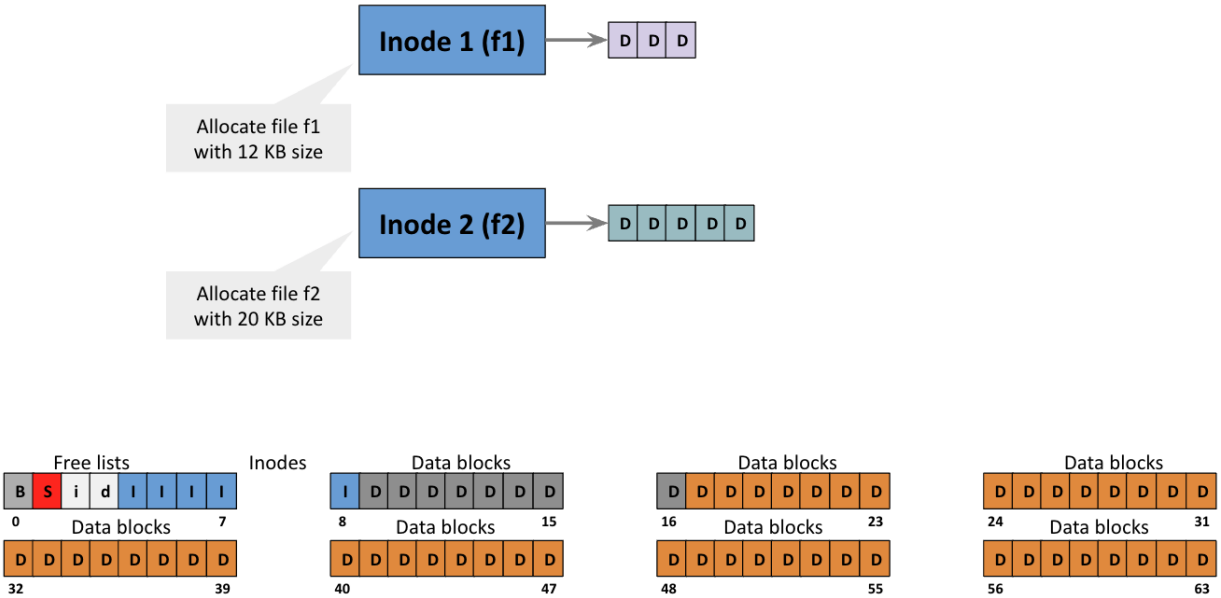
- File type (regular file, directory, symbolic link)
- User ID of the owner
- Permissions (Read/Write/Execute)
- File size in bytes
- Block addresses containing the file's data
- Creation timestamp
- Number of linked paths (hard links)
- Counts of direct and indirect data blocks

6.6.6 File Allocation Methods

File allocation methods determine how files are stored on disk blocks. The following approaches are commonly used:

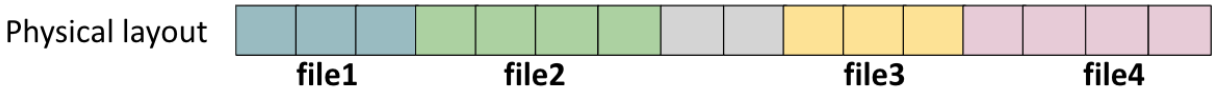
- **Contiguous Allocation**: Files stored in sequential blocks.
- **Linked Allocation**: Files stored as linked lists of blocks.
- **File Allocation Table (FAT)**: Uses a central table to manage file blocks.
- **Multi-level Indexed Allocation**: Files managed through hierarchical index pointers.

The choice of allocation method depends on considerations such as fragmentation, access patterns, metadata overhead, and file growth or shrinkage requirements.



6.6.7 Contiguous Allocation

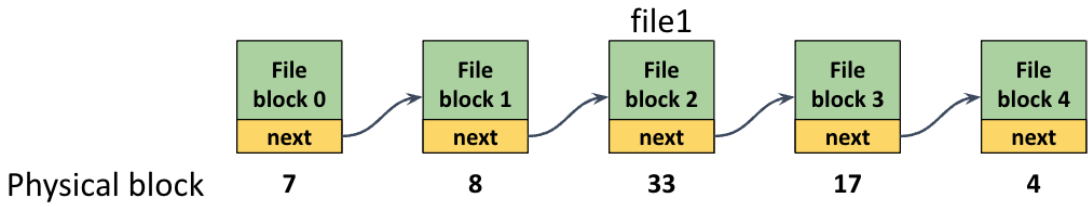
In contiguous allocation, all blocks of a file are stored consecutively on disk.



- **Simplicity**: Requires only the starting block and file length.
- **Efficiency**: Fast sequential and random access (one seek operation for entire file).
- **Fragmentation**: Susceptible to external fragmentation, especially when files are frequently created and deleted.
- **Limitations**: Difficult to resize files; file size must be known at creation.
- **Typical Use**: Ideal for read-only media (e.g., CD/DVD/Blu-ray).

6.6.8 Linked Allocation

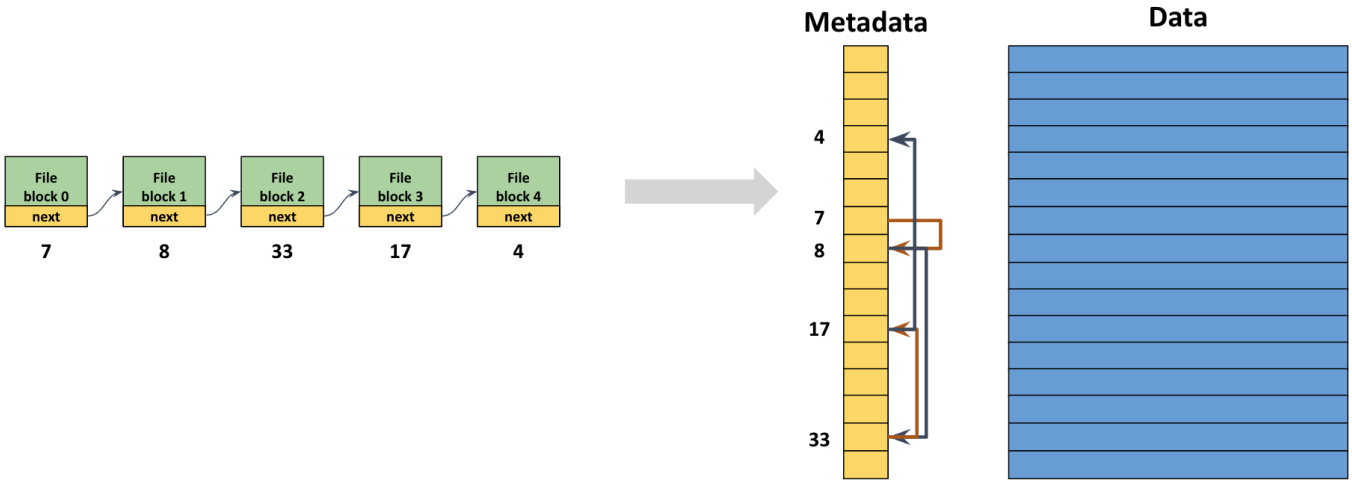
In linked allocation, each file is stored as a linked list of disk blocks. Each block contains data and a pointer to the next block.



- **Space Utilization**: Eliminates external fragmentation.
- **Simplicity**: Only starting block needed to access the file.
- **Performance**: Efficient sequential access, but poor random access.
- **Overhead**: Metadata overhead due to pointers in each block.
- **Implementation**: Data and metadata (pointers) are mixed in each block.

6.6.9 File Allocation Table (FAT)

The FAT system decouples data from metadata by using a centralized table containing block references. Each table entry points to the next block of the file.



- **Structure**: Centralized table separates pointers (metadata) from data blocks.
- **Fragmentation**: Avoids external fragmentation.
- **Simplicity**: File access requires only the starting block index.
- **Performance**: Good sequential access, but slower random access due to indirect lookups.
- **Memory Overhead**: FAT can consume significant memory if not fully cached (e.g., 1GB for a 1TB disk with 4KB blocks).

Chapter 7

File System II

7.1 Block Allocation Strategies

7.1.1 Limitations of Traditional Block Allocation

Files in modern operating systems typically occupy multiple blocks scattered across a disk. This creates several challenges for efficient file access and management:

- **Linked List Approach:** When blocks are linked together, accessing a file requires traversing all preceding blocks.
  - If each block access takes 100 μs, reading 5 blocks requires 500 μs.
- **File Allocation Table (FAT):** To improve performance, systems often cache the FAT in memory.
  - This approach consumes significant memory resources.
  - For each data block, metadata must be stored in the FAT.
  - Let's analyze the memory and performance implications for a large file:

Memory and Performance Analysis for FAT

Given:

- File size: 1 TB (2^{40} bytes)
- Block size: 4 KB (2^{12} bytes)
- FAT entry size: 4 bytes per block (typical)
- Block access time: 100 μs

Number of blocks needed to store the file:

$$\text{Blocks} = \frac{\text{File size}}{\text{Block size}} = \frac{1 \text{ TB}}{4 \text{ KB}} = \frac{2^{40} \text{ bytes}}{2^{12} \text{ bytes}} \tag{7.1}$$

$$= 2^{40-12} = 2^{28} \text{ blocks} \tag{7.2}$$

Memory required for FAT entries (metadata):

$$\text{FAT size} = \text{Number of blocks} \times \text{Entry size} \tag{7.3}$$

$$= 2^{28} \text{ blocks} \times 4 \text{ bytes/block} \tag{7.4}$$

$$= 2^{28+2} \text{ bytes} = 2^{30} \text{ bytes} = 1 \text{ GB} \tag{7.5}$$

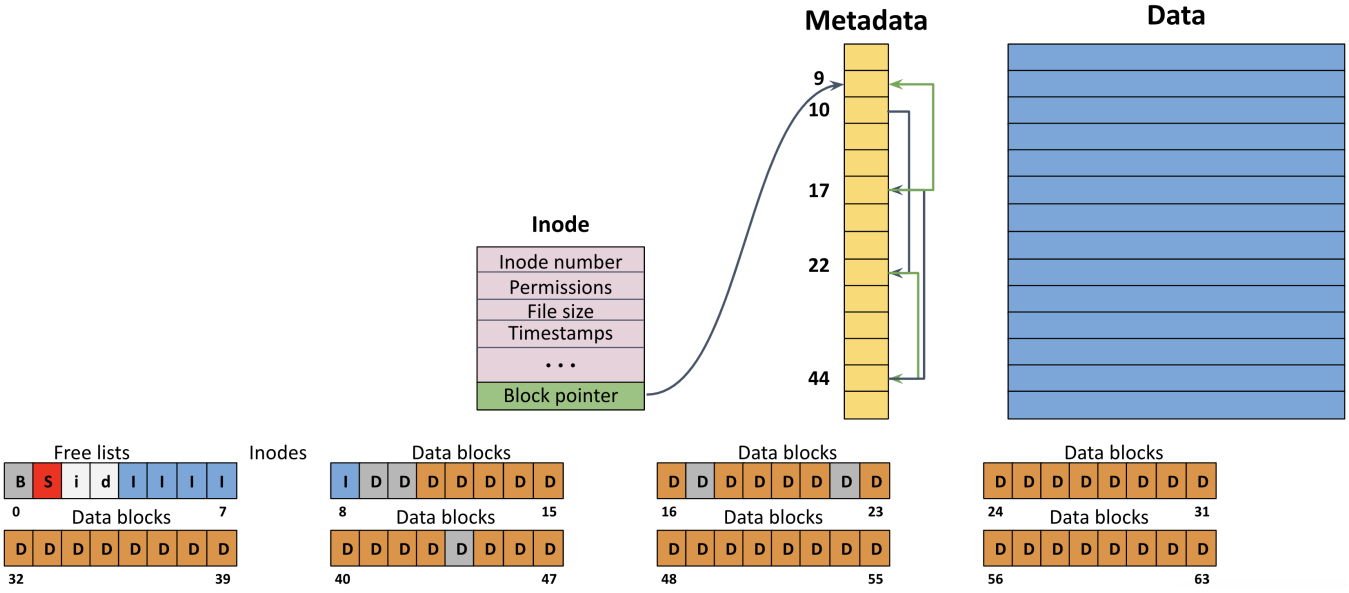
Time to access all metadata (worst case):

$$\text{Access time} = \text{Number of metadata blocks} \times \text{Block access time} \tag{7.6}$$

$$= \frac{1 \text{ GB}}{4 \text{ KB}} \times 100 \ \mu\text{s} = \frac{2^{30}}{2^{12}} \times 100 \ \mu\text{s} \tag{7.7}$$

$$= 2^{18} \times 100 \ \mu\text{s} \approx 26.2 \text{ seconds} \tag{7.8}$$

Implications: For a 1 TB file, the FAT approach requires 1 GB of memory just to store metadata. Reading all this metadata would take approximately 26 seconds, making file operations extremely slow.



7.1.2 Design Goals for Efficient Block Allocation

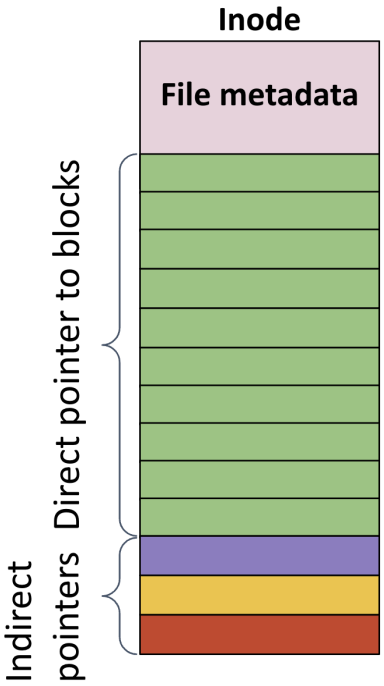
A well-designed block allocation strategy should balance several competing requirements:

- Minimize memory overhead for metadata
- Provide fast access to all parts of a file
- Support both small and large files efficiently
- Scale gracefully as file size increases

7.1.3 The Inode Approach

Key Observation: File systems must efficiently handle two common types of files:

1. **Small files** (< 50 KB)
   - Can be accessed directly with a small set of pointers
   - Direct inode pointers point to data blocks
2. **Large files**
   - Metadata blocks are allocated as the file grows
   - Similar to multi-level page tables
   - Minimizes memory waste through indirection



Inode Pointer Structure

An inode contains a fixed set of pointers that provide access to data blocks using a hierarchical addressing scheme:

Pointer Type	Description	File Size Range
Direct	First 12 pointers point directly to data blocks, providing immediate, single-step access with no indirection overhead.	Small files ($\leq$ 48 KB)
Single-Indirect	Pointer #13 points to a block of pointers where each entry points to a data block (one level of indirection).	Medium files (up to several MB)
Double-Indirect	Pointer #14 points to a block of pointers; each entry in that block points to another block, which in turn contains pointers to data blocks (two levels of indirection).	Large files (up to several GB)
Triple-Indirect	Pointer #15 points to a block of pointers; each entry points to another block of pointers, then to yet another block before finally reaching data blocks (three levels of indirection).	Very large files (up to TB range)

7.1.4 Benefits of the Inode Structure

- **Space Efficiency:** Metadata grows only as needed for larger files
- **Access Speed:** Small files can be accessed with minimal indirection
- **Scalability:** Can address extremely large files with limited overhead
- **Balanced Approach:** Optimizes for both small and large file access patterns

7.2 File Allocation Approach: Multi-level Indexing

The multi-level indexing scheme employs a tree-like structure to organize file data blocks, enhancing the efficiency of block retrieval. This approach uses a combination of direct, single indirect, double indirect, and triple indirect pointers to reference data blocks, thereby adapting the indexing depth to the file size.

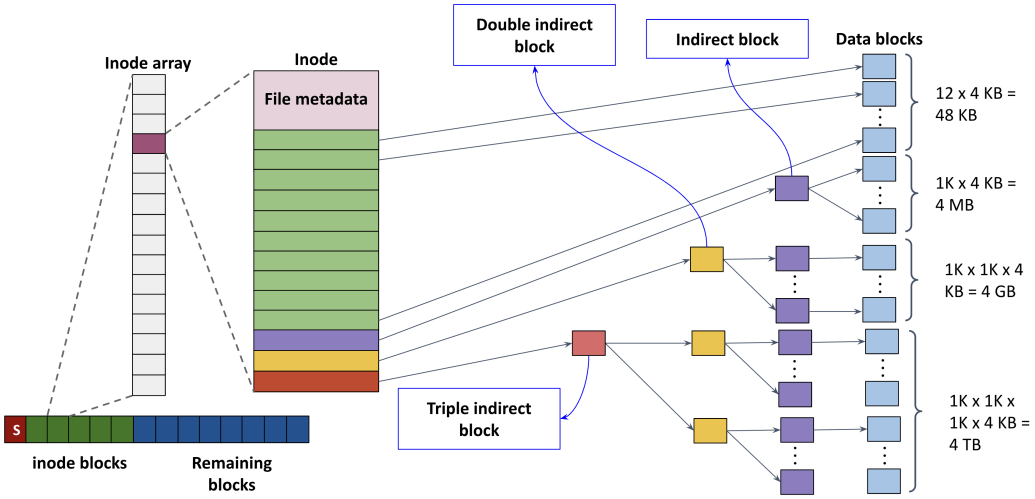
Key Features and Advantages

- **Efficient Block Location:** The tree structure allows rapid location of data blocks. Once an indirect block is read, it can reference hundreds of data blocks, making sequential read operations highly efficient.
- **Asymmetric Overhead:** The design is asymmetric, meaning that small files benefit from minimal overhead by primarily using direct pointers, while larger files leverage additional levels of indirection without incurring a prohibitive metadata cost.
- **Fixed Structure and Simplicity:** The fixed, hierarchical layout simplifies implementation. Metadata is stored separately from data, ensuring there is no conflation between file data and file system metadata.
- **No External Fragmentation:** Since data blocks are allocated without external fragmentation, the overall space utilization is improved.
- **Performance:** The structure provides reasonable read performance with low seek times, balancing the extra reads required for indirect accesses with the overall efficiency of accessing multiple blocks once an indirect block is in memory.

Dynamic Allocation and Practical Considerations

The allocation dynamics are designed to be adaptive:

- **Small Files:** For a file that contains only a few kilobytes of data, direct pointers are used. For example, reading a 4 KB block from a file accessed via a direct pointer incurs minimal overhead.
- **Large Files:** As the file size grows, additional levels of indexing are activated. With a three-level (triple indirect) indexing, even a file requiring 16 KB of data can be managed efficiently. The extra levels allow the file system to scale, enabling support for very large files without a linear increase in metadata.
- **Mixed Access Patterns:** The tree-like indexing provides a good balance between random access (via direct pointers) and sequential reads (via high-degree indirect blocks), which is beneficial for different file access patterns.



The multi-level indexing file allocation method enhances both performance and scalability by adapting the index structure to the file size, ensuring low overhead for small files while supporting efficient access for large files.

7.3 File Operations in a Filesystem

Reading and writing files in a filesystem involve complex sequences of operations that extend beyond simply accessing data. These operations require traversing directory structures, accessing metadata, and managing disk blocks. This section explores the mechanics of these fundamental operations.

7.3.1 Reading from a File

When an application reads data from a file, the operating system performs multiple disk operations to locate and retrieve the requested data. The process begins with opening the file and continues with reading data blocks as needed.

Opening a File for Reading

Before data can be read, the file must be opened:

Example 7.3.1.1 (Opening a file)**.** *`open("/cs202/w07", O_RDONLY)`*

This system call initiates a sequence of operations:

- The filesystem traverses the directory tree to locate the inode for "w07"
- It reads the inode to verify access permissions
- Upon successful verification, it returns a file descriptor that serves as a reference for subsequent operations

Reading Data

Each `read()` operation requires multiple steps:

- The filesystem reads the file's inode to locate the appropriate data blocks
- It reads the data block(s) corresponding to the current file offset
- It updates the last access time in the inode
- It updates the file offset in the in-memory open file table for the file descriptor

Example 7.3.1.2 (Reading the First Two Data Blocks from "/cs202/w07"). Let's look at the complete sequence of operations required to open a file and read its first two data blocks.
Step 1: Opening the File

1. **Root inode access:** The system reads the inode of the root directory (/) to locate its data blocks.
2. **Root directory data:** The filesystem reads the root directory's data blocks to find the entry for "cs202".
3. **cs202 inode access:** Using information from the root directory, it reads the inode for the "cs202" subdirectory.
4. **cs202 directory data:** It reads the data blocks of the "cs202" directory to locate the entry for "w07".
5. **w07 inode access:** Finally, it reads the inode associated with "w07", which contains the metadata and pointers to the file's data blocks.

At this point, the file is open and the system has established the necessary references to access its data.
Step 2: First read() Call

1. **w07 inode read:** The system reads the inode again to retrieve the pointer to the first data block and verify metadata.
2. **Data block access:** It reads the actual first data block of file "w07".
3. **Inode update:** It writes to the inode to update the last access timestamp.

Step 3: Second read() Call

1. **w07 inode read:** The system reads the inode again to retrieve the pointer to the second data block.
2. **Data block access:** It reads the second data block of file "w07".
3. **Inode update:** It writes to the inode to update the last access timestamp again.

The sequence of operations for file reads can be visualized as follows:

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs202 data	w07 data[0]	w07 data[1]
open("cs202/w07")			read()						
						read()			
				read()					
							read()		
					read()				
read()					read()				
								read()	
					write()				
read()					read()				
									read()
					write()				

7.3.2 Writing to a File

Writing to a file involves more complex operations than reading, particularly when new data blocks need to be allocated.

Opening a File for Writing

Similar to reading, writing begins with opening the file:

Example 7.3.2.1 (Opening a file for writing). *`open("/cs202/w07", O_WRONLY)`*

This assumes the file already exists. If it doesn't, additional operations would be required to create it.

Writing Data

Each logical write operation can generate multiple physical I/O operations:

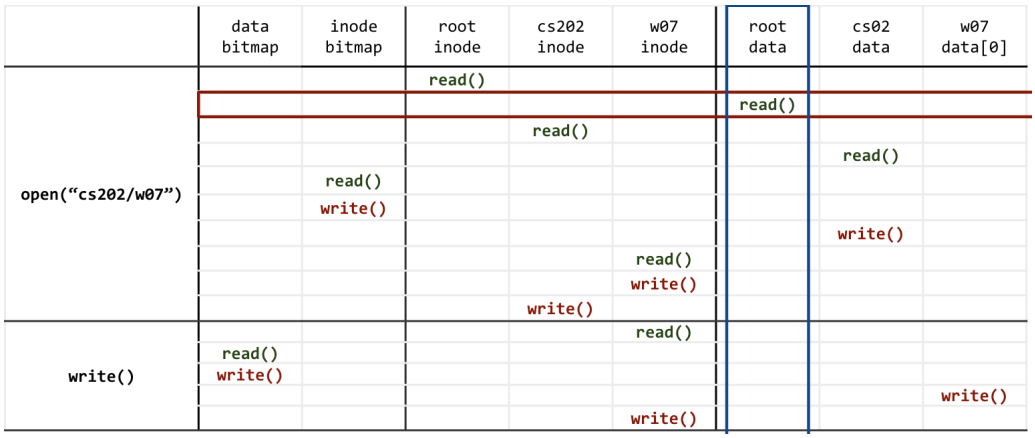
1. Read the free data block bitmap to locate available space
2. Write to the data block bitmap to mark the block as allocated
3. Read the file's inode to access its metadata
4. Write to the file's inode to update its block pointers
5. Write the actual data to the newly allocated block

File Creation and Additional Complexity

Creating a new file involves even more operations:

- Reading and writing the free inode bitmap to allocate an inode
- Writing the new inode with initial metadata
- Reading and updating the parent directory's data blocks
- If the parent directory is full, allocating new blocks for it

Example 7.3.2.2 (Creating and Writing to a New File "/cs202/w07"). Now, let's look at the complete sequence of operations required to create a new file and write its first data block.



Step 1: Creating the File

1. **Root inode access:** Reads the root directory's inode to locate its data blocks.
2. **Root directory data:** Reads the root directory's data to find the entry for "cs202".
3. **cs202 inode access:** Reads the inode for the "cs202" directory.
4. **cs202 directory data:** Reads "cs202" directory data to verify "w07" doesn't already exist.
5. **Inode bitmap operations:** Reads the inode bitmap to find a free inode, then writes to mark it as allocated.
6. **Directory update:** Updates the "cs202" directory data to include an entry for "w07" linked to the new inode.
7. **New inode initialization:** Writes initial metadata to the new inode (permissions, owner, timestamps).
8. **Parent directory update:** Updates the metadata for "cs202" (modification time, entry count).

Step 2: Writing Data to the New File

1. **w07 inode access:** Reads the new file's inode to access its metadata.
2. **Data bitmap operations:** Reads the data bitmap to find a free data block, then writes to mark it as allocated.
3. **Data write:** Writes the actual file content to the newly allocated data block.
4. **Inode update:** Updates the "w07" inode with the new file size, data block pointers, and timestamps.

7.4 File System Performance

File system performance is a critical aspect of operating system design that directly impacts user experience and application efficiency. This section explores how performance is defined, measured, and optimized in file systems.

7.4.1 Performance Metrics and Evaluation

Performance in file systems can be evaluated from multiple perspectives, each focusing on different aspects of system behavior:

Definition (File System Performance). *The measure of how efficiently a file system can execute operations such as reading, writing, and metadata manipulation, typically expressed in terms of latency, throughput, and resource utilization.*

When evaluating file system performance, several factors must be considered:

- **Operation count:** The number of I/O operations required to complete a task
- **Operation speed:** The time required to complete individual I/O operations
- **Program-level impact:** Effect on the performance of a single program
- **System-level impact:** Effect on overall system performance across all programs

These factors can be quantified using the following key metrics:

- **Latency:** The time delay between initiating and completing an operation
- **Throughput:** The amount of data processed per unit time (e.g., MB/s)
- **IOPS (I/O Operations Per Second):** The number of read/write operations a storage system can perform in one second

7.4.2 Performance Optimization Strategies

File systems employ various strategies to optimize performance, each addressing different performance bottlenecks

Definition (Block Cache). *A memory area that temporarily stores recently accessed disk blocks to reduce the need for physical disk operations when the same data is requested again.*
Caching significantly improves performance by reducing the need for slow disk operations:

- Frequently accessed blocks remain in memory, allowing `read()` operations to complete without disk I/O
- Modern systems often dedicate all unused memory to the file system buffer cache
- The cache maps file identifiers (inode, block offset) to physical memory locations (page frame numbers)

Operation Batching

Grouping multiple operations together can significantly improve overall system throughput:

Example 7.4.2.1 (Write Batching)**.** *Instead of writing data to disk immediately after each user interaction, an application can queue multiple write operations for 5 seconds and then perform them as a batch. This reduces the total number of disk accesses, improving throughput at the cost of slightly increased latency for individual operations.*

The benefits of operation batching include:

- Reduced disk seek time by grouping operations on physically proximate disk sectors
- Amortized per-operation overhead across multiple operations
- Opportunity for operation optimization and reordering

Delayed Idempotent Operations

Definition (Idempotent Operation). *An operation that can be performed multiple times without changing the final outcome beyond the initial application.*
Delaying or batching idempotent operations provides performance benefits without compromising correctness:

Example 7.4.2.2 (Timestamp Updates)**.** *Updating a file's "last accessed" timestamp can be delayed or batched because only the most recent timestamp is relevant. Multiple updates within a short time window can be coalesced into a single disk write.*

Strategic Indirection

Adding levels of indirection enables optimization opportunities:

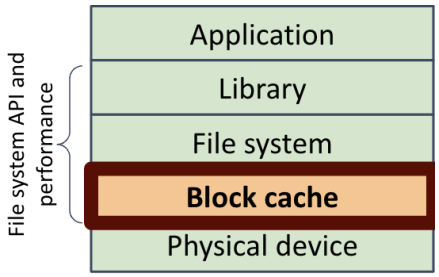
- Maintaining abstractions that decouple logical operations from physical ones
- Allowing the system to reorder or coalesce operations
- Providing flexibility in how and when operations are physically executed

7.4.3 The Block Cache Architecture

The block cache serves as a critical performance optimization layer in file systems

Example 7.4.3.1 (Block Cache Operation)**.** When an application repeatedly reads the same inode block

1. **First read:** The block is loaded from disk into the block cache
2. **Subsequent reads:** The system checks if the block is in the cache using the mapping: $\{inode, block\_offset\} \rightarrow page\_frame\_number$
3. If found, the data is returned directly from memory without disk I/O
4. The block remains in cache until memory pressure forces eviction



Key Block Cache Characteristics

- Dynamically adjusts size based on system memory availability
- Implements replacement policies to maximize cache hit rates
- Manages consistency between cached blocks and their disk versions
- May implement read-ahead or prefetching to anticipate future access patterns

These performance optimization strategies collectively ensure that file systems can deliver high throughput and low latency despite the inherent performance limitations of physical storage devices.

7.4.4 Optimizing I/O Operations Through Batching

Definition (I/O Batching). *The process of combining multiple I/O operations into larger, more efficient transfers to minimize overall system overhead and maximize throughput.*
Modern file systems employ batching strategies to address two key performance limitations:

- High latency cost per individual I/O operation
- Limited I/O operations per second (IOPS) capacity

Storage-Specific Optimizations

Different storage technologies benefit from distinct batching strategies:

- **Hard Disk Drives (HDD):**
  - Optimizes for sequential access by grouping operations on consecutive disk blocks
  - Minimizes seek time by processing physically proximate blocks together
  - Performance heavily influenced by disk fragmentation - the degree to which an inode's blocks are non-contiguous
- **Solid State Drives (SSD):**
  - Leverages internal parallelism for concurrent operations
  - Benefits from larger transfer sizes due to internal architecture
  - Less sensitive to physical block placement

7.4.5 Asynchronous Operations and Write Delays

While read operations typically require immediate process blocking, write operations present opportunities for optimization through delayed execution:

Example 7.4.5.1 (Asynchronous Write Operations)**.** *When an application writes data:*

1. *Data is initially stored in memory buffers*
2. *Write operations are queued for asynchronous processing*
3. *System performs actual disk writes within a defined interval (typically 30 seconds)*
4. *Operations may be reordered to optimize throughput*

Definition (Write Delay). *A performance optimization technique where write operations are temporarily held in memory and executed asynchronously to improve system throughput.*

Important Note: While write delays improve performance, they introduce a risk of data loss in case of system crashes before cached data is written to disk.

7.4.6 Cache Impact on Data Persistence

File systems cache multiple critical data structures to enhance performance:

- Free block and inode bitmaps
- Directory entries
- Inode metadata
- Data blocks

While caching significantly improves read performance, it introduces complexity for write operations due to the need to maintain data consistency between memory and disk.

7.4.7 Write Caching Policies

File systems implement different caching strategies to balance performance and data consistency:

Definition (Write-Back Cache). *A caching policy where modifications are initially made to the cache and later written to disk, prioritizing performance over immediate consistency.*

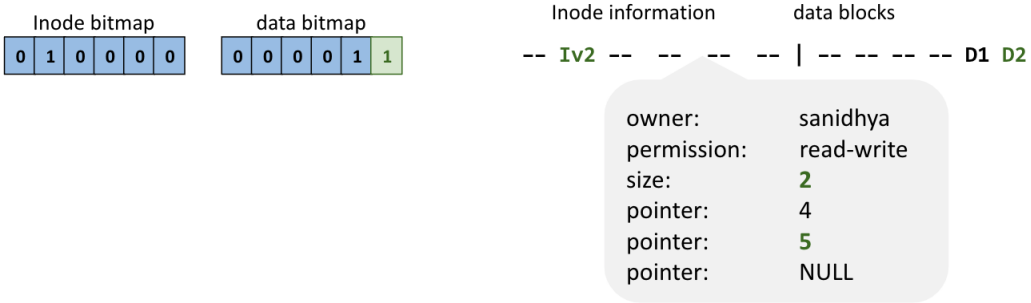
Definition (Write-Through Cache). *A caching policy where modifications are immediately written to both cache and disk, ensuring consistency at the cost of performance.*

Cache Policy	Advantages	Disadvantages
Write-Back	Higher performance Better I/O optimization	Risk of data loss during system crashes
Write-Through	Guaranteed consistency Immediate persistence	Lower performance Higher I/O overhead

Applications can force immediate disk writes using the `fsync` system call when data consistency is critical.

7.5 Crash Consistency

Suppose we are appending a data block to a file. This operation involves several steps: adding a new data block $D2$, updating the inode, and updating the data bitmap.

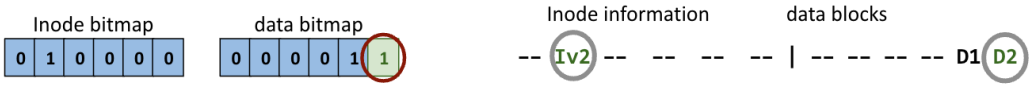


What happens if a crash or power outage occurs during these writes? The key issue is that file system operations often involve multiple write operations, and a failure between these operations can lead to an inconsistent state.

7.5.1 Single Write Scenario

Consider the case where only one write operation is successfully written to disk before a crash. Let's examine a few possibilities:

- **Data Block $D2$ is written**: The data is written, but there is no valid inode pointing to it. $D2$ appears as a free block in the metadata. The write is essentially lost, but the file system metadata structures remain consistent.
- **Inode ($Iv2$) is written**: If only the updated inode $Iv2$ is written, following the block pointer will lead to reading garbage data. This results in an inconsistent file system because the data bitmap indicates that the block is free, while the inode claims it is in use.
- **Updated Data Bitmap is written**: If only the updated data bitmap is written, the file system becomes inconsistent because the data bitmap indicates that a data block is in use, but no inode points to it.



7.5.2 Multiple Writes Scenario

Now, let's consider scenarios where two write operations succeed before a crash:

- **Inode and Data Bitmap updates succeed**: The file system remains consistent from a metadata perspective. However, reading the new block will return garbage data because the actual data block $D2$ was not successfully written.
- **Inode and Data Block updates succeed**: This leads to an inconsistent file system because the inode points to the new data block, but the data bitmap might not reflect that the block is in use.
- **Data Bitmap and Data Block updates succeed**: This also results in an inconsistent file system because the data bitmap marks the data block as used, but no inode points to it.

Caching exacerbates these issues because data can be written asynchronously, making it harder to predict the order of writes.
If the file system is interrupted between these writes, it can lead to an inconsistent state due to:

- Power loss and hard reboot
- Kernel panic
- File system bugs

Therefore, a mechanism is needed to recover from or fix these inconsistent states.

7.5.3 The Consistent Update Problem

The fundamental problem is that several file system operations update multiple data structures. Caching can worsen the issue because data can be written asynchronously. If a file system operation is interrupted between writes, it may leave the data in an inconsistent state. This can occur due to power loss, hard reboots, kernel panics, or file system bugs.
Therefore, the goal is to have a mechanism to recover from (or fix) an inconsistent state.

7.5.4 Consistency Solution #1: File System Checker (FSCK)

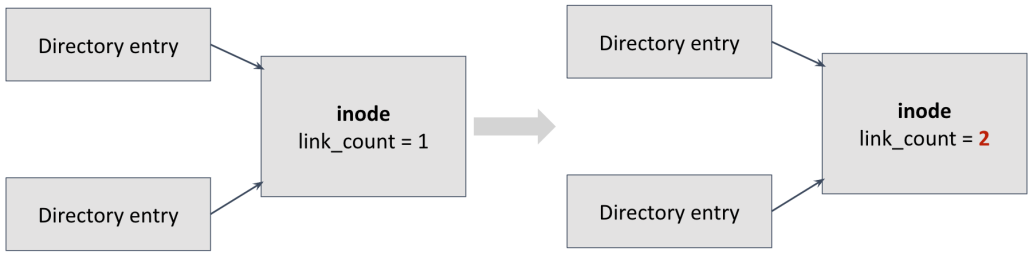
One approach to address file system inconsistencies is using a file system checker (FSCK). FSCK is a utility that checks the consistency of the file system after a certain number of mount operations or after a crash. It performs hundreds of consistency checks across different fields, such as:

- Do superblocks match?
- Is the file system size reasonable?
- Are link counts equal to the number of directory entries?

7.5.5 The File System Checker

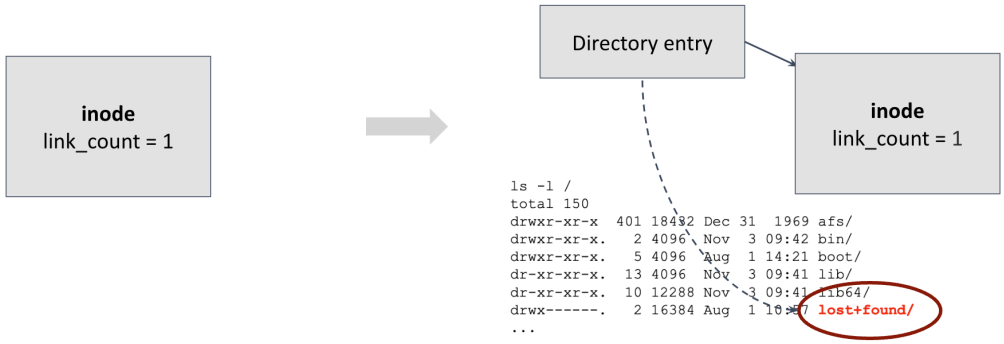
The file system checker (`fsck`) is a crucial utility for maintaining file system integrity. It is automatically invoked after a specific number of mount operations or following a system crash to ensure the file system's consistency. The `fsck` performs numerous checks across various file system components, addressing potential issues like incorrect link counts, data bitmap errors, duplicate pointers, and invalid pointers.
The following are examples of the consistency checks `fsck` performs:

- **Link Count Inconsistencies:** The `fsck` verifies that the number of directory entries pointing to an inode matches the inode's link count. For instance, if two directory entries point to the same inode but the inode's link count is set to 1, `fsck` detects this inconsistency.



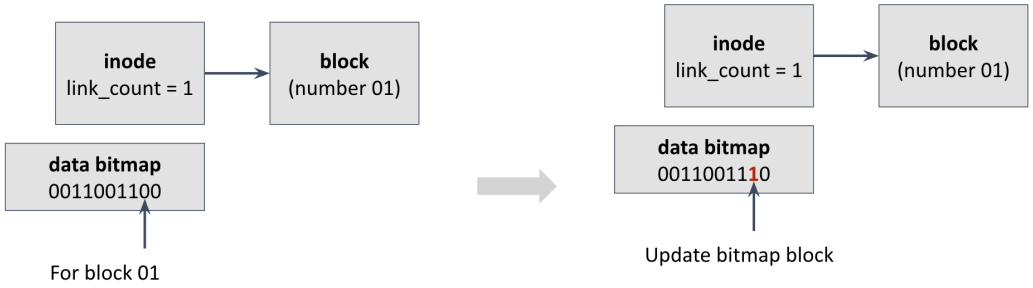
Fix the link count by increasing to 2

- **Lost Inodes:** If an inode has a link count greater than zero but no directory entries point to it, the `fsck` moves the corresponding file to the `lost+found` directory, allowing for potential recovery by the system administrator.



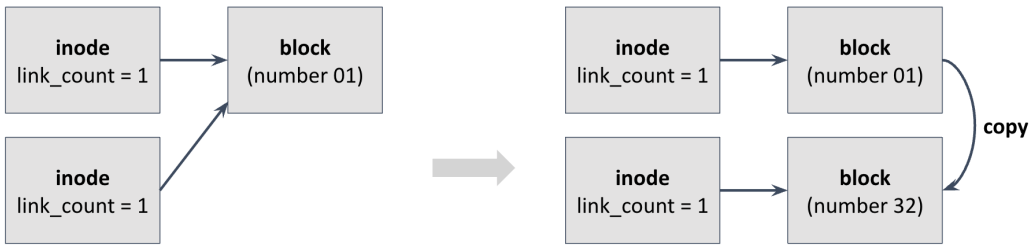
Link the file in lost+found directory

- **Data Bitmap Errors:** The `fsck` ensures the data bitmap accurately reflects the allocation status of blocks. If an inode points to a block, but the corresponding bit in the data bitmap is 0 (indicating the block is free), `fsck` corrects the bitmap to reflect the block's usage.



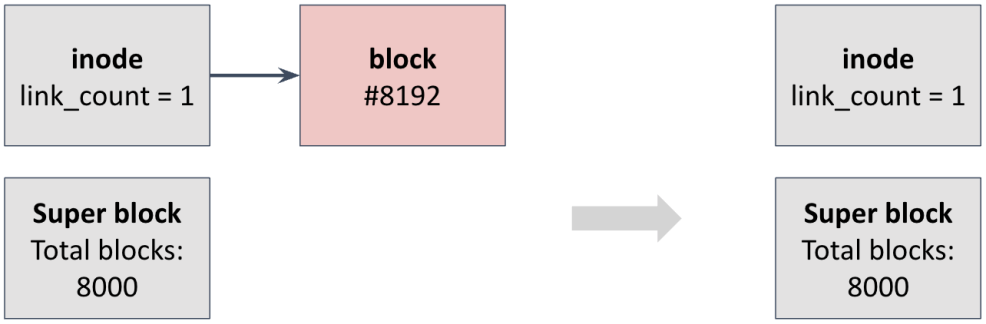
Update a reference block in the bitmap

- **Duplicate Pointers:** The `fsck` identifies and resolves situations where multiple inodes point to the same data block, which can lead to data corruption. In such cases, `fsck` may create a duplicate of the block, updating one of the inodes to point to the new duplicate, thus preserving data integrity.



Make a copy of the data block

- **Invalid Pointers:** The `fsck` checks for inodes pointing to blocks with numbers exceeding the total number of blocks in the file system. Such pointers are invalid and can cause crashes or data corruption. The `fsck` removes these invalid pointers to prevent further issues.



Remove the reference of the data block

7.5.6 Problems with FSCK

While `fsck` is essential, it has limitations:

- **Functionality:** `fsck` aims to bring the file system to a consistent state, which is not always the "correct" state. Determining the appropriate corrections can be challenging, and in severe cases, reformatting the disk may seem like the easiest solution, albeit with significant data loss.

- **Performance:** `fsck` can be slow, sometimes taking hours to complete, especially on large file systems. This prolonged downtime can be disruptive.

7.6 Consistency Solution #2: Journaling

To address the limitations of `fsck`, journaling offers an alternative approach to maintaining file system consistency with the following goals:

- Minimize the amount of work required for recovery after a crash.

- Achieve the *correct* state of the file system, not just a consistent one.

The core idea behind journaling is to record changes in a journal (or log) before applying them to the actual file system. This journal serves as a historical record of operations, allowing the system to recover to a known good state in the event of a crash. No need to scan the entire disk

Definition (Journaling). *Journaling is a technique used in file systems where all intended modifications are first recorded in a sequential log (the "journal") before being committed to the main file system. This write-ahead logging ensures atomicity and durability of operations, facilitating recovery after a crash.*

Before modifying any data (read, write, delete, etc.), the changes are first recorded in the journal. The journal is a special area on the disk that stores data in a write-ahead fashion. Journaling leverages the atomicity of transactions to provide crash consistency.

7.6.1 A Principled Approach: Transactions

Journaling relies on the concept of transactions to ensure data integrity.
Definition (Transaction). *A transaction is a group of operations treated as a single logical unit of work. It must adhere to the ACID properties: Atomicity, Consistency, Isolation, and Durability.*

- **Atomic:** A transaction is indivisible; either all operations within it are executed, or none are.
- **Consistent:** A transaction must maintain the integrity of the data. It moves the system from one valid state to another.
- **Isolated:** Concurrent transactions should not interfere with each other. The effects of one transaction should not be visible to others until it is complete.
- **Durable:** Once a transaction is committed, its effects are permanent and survive system failures.

Transactions can have two outcomes:

- **Commit:** The transaction is successfully completed, and its changes are made permanent.
- **Abort:** The transaction is terminated, and any changes made during the transaction are rolled back, restoring the system to its previous state.

7.6.2 How Journaling Works

Journaling groups file system operations into atomic and consistent units using transactions. `TxBeg` and `TxEnd` markers denote the start and end of a transaction. The process involves writing to the journal first and then writing the actual file system blocks (checkpoint) in a specific order. Journaling can be applied to both data and metadata blocks.

7.6.3 Data Journaling: An Example

Consider adding a new block `D2` to a file. This can be viewed as similar to operations in `git`. The steps involved in data journaling are:

1. Write the following blocks to the journal: `TxBeg | Iv2 | Bv2 | D2 | TxEnd`. Here, `Iv2` represents the inode update, `Bv2` represents the bitmap information, and `D2` is the new data block. Writing each record to a separate block ensures atomicity.
2. Write the blocks `Iv2`, `Bv2`, and `D2` to their respective locations in the file system (checkpoint).
3. Mark the transaction as free in the journal (i.e., remove it).

In case of a crash:

- If the crash occurs before the log is updated, the changes are ignored as if the transaction never happened.
- If the crash occurs after the log is updated but before the checkpoint, the changes are replayed from the log back to the disk during recovery.

7.6.4 Simplified Journaling Example

Consider the goal of atomically writing the value 10 to block 0 and the value 5 to block 1.

Time	Block 0	Block 1	Extra	Extra	Extra
0	12	3	0	0	0
1	10	3	0	0	0
2	10	5	0	0	0

A naive approach of directly writing to the blocks is problematic because a crash between the two writes could leave the file system in an inconsistent state.
Journaling solves this by first writing the changes to the journal within a transaction.

Time	Block 0	Block 1	Block 0'	Block 1'	Valid
0	12	3	0	0	0
1	12	3	10	0	0
2	12	3	10	5	0
3	12	3	10	5	1
4	10	3	10	5	1
5	10	5	10	5	1
6	12	3	10	5	0

The steps are as follows:

1. Write the transaction begin marker (`TxBeg`) to the journal.
2. Write the data for block 0 (value 10) to the journal.
3. Write the data for block 1 (value 5) to the journal.
4. Write a valid block indicator to the journal, signifying that the transaction completed successfully in the journal.
5. Write the data for block 0 (value 10) to block 0 in the main file system.
6. Write the data for block 1 (value 5) to block 1 in the main file system.

Time	Block 0	Block 1	Block 0'	Block 1'	Valid
0	12	3	0	0	0
1	12	3	10	0	0
2	12	3	10	5	0
3	12	3	10	5	1
4	10	3	10	5	1
5	10	5	10	5	1
6	12	3	10	5	0

Crash Scenarios:

- **Crash before time unit 3:** The file system retains the old data, as the transaction was not fully written to the journal.
- **Crash after time unit 3 but before time unit 6:** Upon recovery, the system detects the incomplete transaction in the journal and replays the changes to blocks 0 and 1, ensuring the new data is present.
- **Crash after time unit 6:** The new data is already present in the file system, and no recovery is needed.

Chapter 8

Input/Output Systems

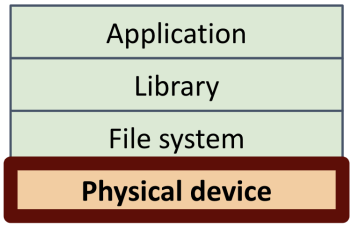
In this lecture, we will discuss how computers communicate with I/O devices, the layered structure of I/O operations, and the various mechanisms used to optimize I/O performance.
This was mostly covered in Computer Architecture.

8.1 I/O System Architecture

8.1.1 Layered Approach to I/O Operations

I/O operations in modern computing systems follow a layered approach, with each layer providing a specific set of responsibilities:

- **Application Layer:** Programs that need to read or write data
- **Library Layer:** Standard libraries that provide I/O functions to applications
- **Operating System Layer:** File systems and device drivers that translate generic operations into device-specific commands
- **Hardware Layer:** Physical devices that perform the actual I/O operations



This layered architecture provides abstraction, allowing applications to perform I/O operations without understanding the underlying hardware details.

8.1.2 Core I/O Services in Operating Systems

Operating systems provide several essential I/O services:

- Loading programs and data from storage devices
- Writing data to display devices (terminals, screens)
- Reading and writing network packets
- Capturing input from various input devices (keyboard, mouse, sensors)

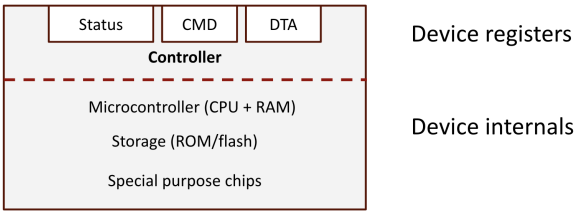
These services form the foundation of how users and applications interact with the computer system.

8.2 Device Interaction Models

8.2.1 Canonical Device Structure

Device communication follows a standardized model that abstracts hardware complexity:

Definition (Canonical Device). *A **canonical device** refers to a standardized model of I/O device interaction where the CPU communicates with a device controller through designated registers, while the internal workings of the device remain hidden from the system.*



In this interaction, we have the following components:

- **Device Controller:** Hardware that interfaces between the device and the system
- **Device Registers:** Control, status, and data registers used for communication
- **Device Driver:** Software component that knows how to communicate with specific devices
- **Signaling Mechanisms:** Methods (polling or interrupts) for the device to signal its status to the CPU

Question: Why do we setup data before setting up the CMD register?

Answer: Race condition if data is not present

Definition (Race Condition).
*A **race condition** occurs when the timing or ordering of events affects the correctness of a program. In device interactions, setting up data before the command prevents race conditions where the device might begin execution before all necessary data is available.*

Thus the following steps are taken to communicate with the device:

1. Wait until the device is ready (check status register)
2. Set data in the data register
3. Issue command by writing to the command register
4. Wait for command completion (check status register)

```c
// 1. Wait until device is ready
while (STATUS == BUSY) ;

// 2. Write data to data register
*dtaRegister = DATA;

// 3. Write command to command register
*cmdRegister = COMMAND;

// 4. Wait until command completes
while (STATUS == BUSY) ;
```

8.3 Parameters of I/O Operations

When designing or analyzing I/O systems, five fundamental parameters must be considered:

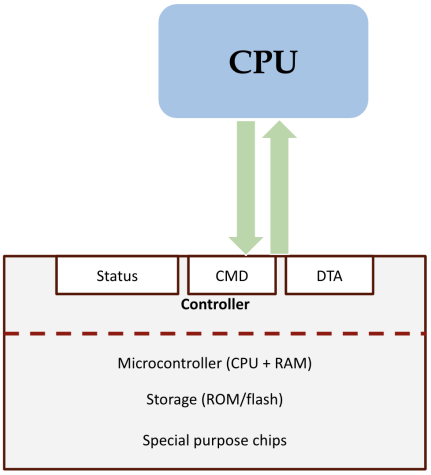
1. **Communication Method:** How does the CPU communicate with the device?
2. **Data Granularity:** What is the size of data transfers (byte, block, etc.)?
3. **Access Pattern:** How is data accessed (sequentially or randomly)?
4. **Notification Mechanism:** How does the device signal the CPU?
5. **Transfer Mechanism:** How is data moved between memory and device?

8.3.1 CPU-Device Communication: Memory-Mapped I/O

Definition (Memory-Mapped I/O (MMIO)). ***Memory-Mapped I/O (MMIO)*** *is a method where device control registers are mapped into the physical address space of the processor, allowing the CPU to access devices using standard memory access instructions (load/store).*

In Memory-Mapped I/O, we have the following characteristics:

- Device registers appear as memory locations to the CPU
- Standard load/store instructions are used for device communication
- Applicable to a wide range of devices
- Particularly effective for high-performance devices (fast storage, network interfaces, displays)



8.3.2 Data Granularity and Access Patterns

Different I/O devices operate with different data sizes and access patterns:

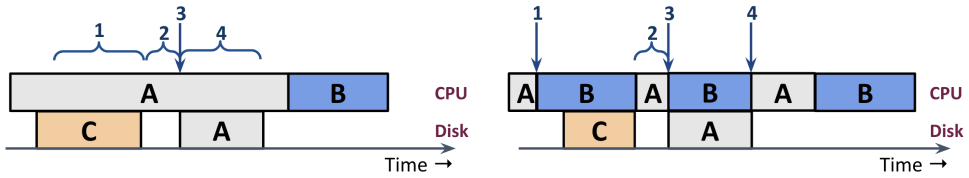
- **Data Granularity:**
  - *Byte-oriented devices:* Transfer one byte at a time (e.g., keyboards, serial ports)
  - *Block-oriented devices:* Transfer blocks of data (e.g., disk drives, network cards)
- **Access Patterns:**
  - *Sequential access:* Data must be accessed in order (e.g., tape drives)
  - *Random access:* Data can be accessed in any order (e.g., disk drives, SSDs)

These characteristics significantly impact the design of device drivers and the overhead involved in data transfers.

8.4 Notification Mechanisms

8.4.1 From Polling to Interrupts

Waiting for device operations to complete poses a challenge for system efficiency. Two main approaches address this:



Definition (Polling). ***Polling*** *(or busy-waiting) is a notification technique where the CPU periodically checks a device's status register to determine if an operation has completed.*

Definition (Interrupt). *An* ***interrupt*** *is a hardware signal sent from a device to the CPU that causes the CPU to temporarily suspend its current execution, save its state, and execute an interrupt handler routine.*

8.4.2 Comparing Polling and Interrupts

Aspect	**Polling**	**Interrupts**
Mechanism	CPU periodically checks device status	Device signals CPU when attention needed
CPU Utilization	Wastes CPU cycles when events are infrequent	Efficient for unpredictable or infrequent events
Overhead	Low per-check overhead	Higher overhead for context switching
Predictability	Predictable timing	Less predictable
Best Use Cases	High-frequency events, short wait times	Unpredictable events, long wait times

Definition (Livelock). ***Livelock*** *is a situation where a system is continuously responding to interrupts and cannot make progress on its main tasks, similar to deadlock but with processes actively running rather than blocked.*

8.4.3 Optimizing Notification Mechanisms

Modern systems use sophisticated approaches to balance efficiency:

- **Hybrid Approaches:** Using both polling and interrupts depending on workload characteristics
- **Interrupt Coalescing:** Delaying and batching multiple interrupts to reduce overhead
- **Adaptive Strategies:** Dynamically switching between polling and interrupts based on system load and device activity

8.4.4 Data Transfer Mechanisms

The final aspect of I/O operations concerns how data is transferred between main memory and device controllers.

Definition (Programmed I/O (PIO)). ***Programmed I/O (PIO)*** *is a data transfer technique where the CPU directly controls data movement between memory and a device, executing an instruction for each data unit transferred.*

Definition (Direct Memory Access (DMA)). ***Direct Memory Access (DMA)*** *is a data transfer technique that allows a device controller to directly transfer data to or from main memory without CPU intervention, after initial setup by the CPU.*

Aspect	Programmed I/O	Direct Memory Access
CPU Involvement	CPU handles each data transfer	CPU only sets up transfer, then free for other tasks
Efficiency	Efficient for small transfers	Efficient for large transfers
Complexity	Simpler implementation	More complex hardware needed
CPU Overhead	High, proportional to data size	Low, independent of data size
Best Use Cases	Small data transfers, simple devices	Large data transfers, high-performance devices

- **Programmed I/O (PIO):**
  - CPU directly controls data transfer, telling the device what data is
  - Requires one instruction for each byte/word transferred
  - Efficient for small transfers but consumes CPU cycles proportional to data size
- **Direct Memory Access (DMA):**
  - CPU only tells the device where data is located in memory
  - Controller is granted access to the memory bus
  - Device transfers data to/from memory without CPU intervention
  - Highly efficient for large data transfers

8.5 Direct Memory Access (DMA)

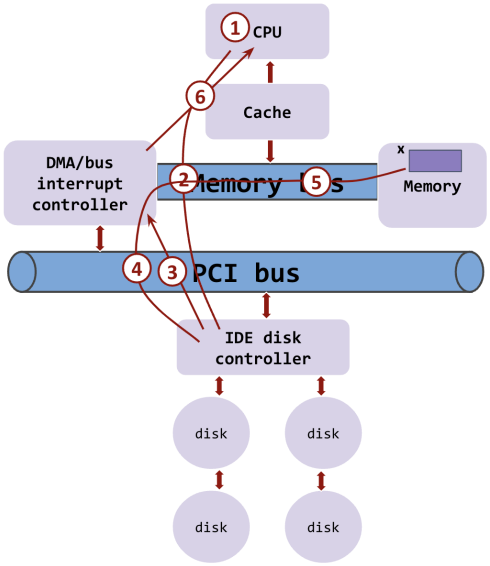
Definition (Direct Memory Access (DMA)). *Direct Memory Access (DMA) is a method that allows hardware subsystems to access main system memory independently of the CPU, enabling efficient data transfer between I/O devices and memory.*

8.5.1 DMA Controller Operation

The DMA controller facilitates direct data transfer between peripheral devices and memory without constant CPU intervention, significantly improving system efficiency for data-intensive operations.

The following steps illustrate a typical DMA transfer sequence:

1. The device driver receives an instruction to transfer disk data to a buffer at address X.
2. The driver commands the disk controller to transfer C bytes from disk to the buffer at address X.
3. The disk controller initiates the DMA transfer operation.
4. The disk controller sends each byte to the DMA controller.
5. The DMA controller transfers bytes to buffer X, incrementing the memory address and decrementing C until C = 0.
6. When C = 0, the DMA controller interrupts the CPU to signal completion of the transfer.



8.6 Device Management in Operating Systems

8.6.1 The Device Driver Challenge

Modern computing systems must interface with numerous devices, each with unique protocols and requirements. This diversity creates significant challenges for operating system design.

Definition (Device Driver). *A device driver is a specialized component of the operating system kernel that directly interfaces with hardware devices, translating between the standardized OS interfaces and device-specific protocols.*

Device drivers solve the challenge of hardware diversity by:

- Supporting a standard, internal interface within the OS
- Enabling the same kernel I/O system calls to interact with different physical devices
- Providing device-specific implementations of standard operations

8.6.2 Principles of Device Driver Design

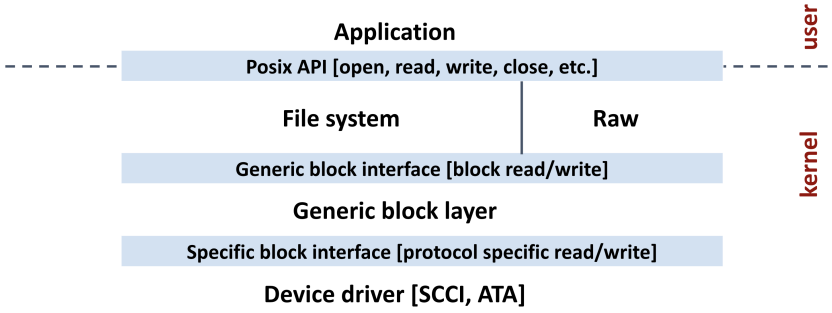
Device drivers demonstrate encapsulation in system design. Different drivers adhere to the same API, allowing the OS to interact with diverse hardware through consistent interfaces. The OS implements support for APIs based on device class rather than specific hardware models.

This approach requires:

- Well-designed interfaces and APIs
- Careful balance between versatility and specialization
- Layered API structure to manage complexity

8.6.3 Complexity of API Layers

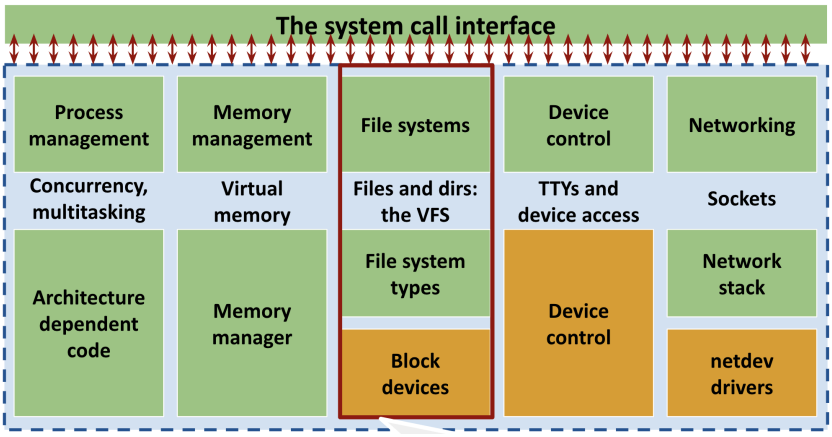
The file system stack exemplifies the layered approach to I/O management:



This layering allows for abstraction and modularity while providing necessary functionality at each level.

8.6.4 OS Device Structure

The overall device structure in an operating system organizes components into logical layers:



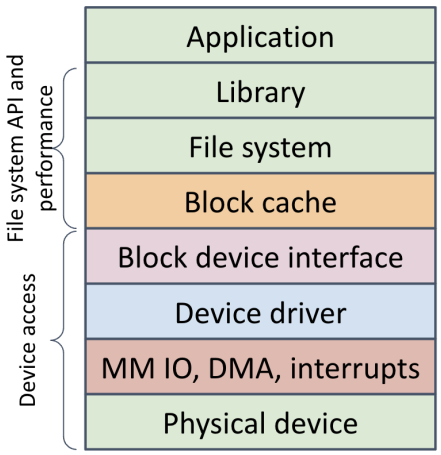
- **Process Management:** Handles process creation, scheduling, synchronization, and termination, allowing multiple programs to run concurrently.
- **Memory Management:** Controls allocation and deallocation of memory resources, implements virtual memory, and manages address translation.
- **File Systems:** Provides abstractions for persistent data storage, organizing files and directories while managing access permissions.
- **Device Control:** Interfaces with hardware peripherals through device drivers, translating generic commands into device-specific operations.
- **Networking:** Implements network protocols and provides interfaces for network communication, enabling data exchange between systems.

The system call interface serves as the boundary between user applications and these kernel components, providing a standardized way for programs to request services from the operating system. Through system calls, applications can interact with all these subsystems without needing to understand their internal implementation details.

8.6.5 General I/O Abstraction Stack

I/O systems are accessed through a series of layered abstractions that progressively translate between user-level operations and hardware-specific details:
These layers provide:

- Caching of recently read disk blocks
- Buffering of recently read blocks
- A unified interface to diverse devices
- Fixed-size block data operations
- Translation between OS abstractions and hardware-specific details
- Control of hardware registers, bulk data transfers, and OS notifications



8.7 Storage Systems

8.7.1 Storage Media Evolution

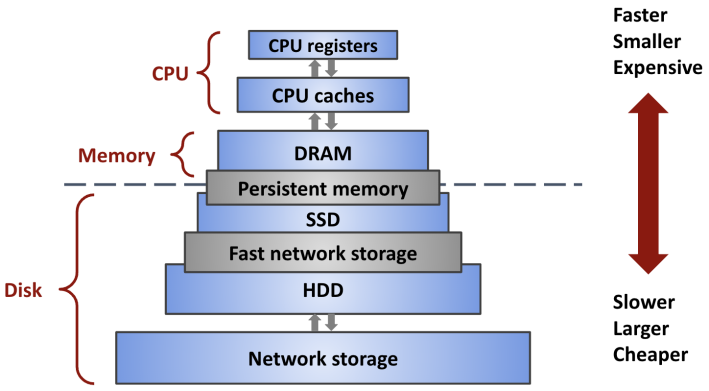
Definition (Persistent Storage). *Persistent storage refers to data storage technologies that retain information even when power is removed from the system.*

Computer systems have employed various storage media for persistent data:

- Punched paper cards
- Magnetic tapes
- Floppy disks
- Magnetic hard disks
- Optical disks
- USB flash drives
- Solid-state drives

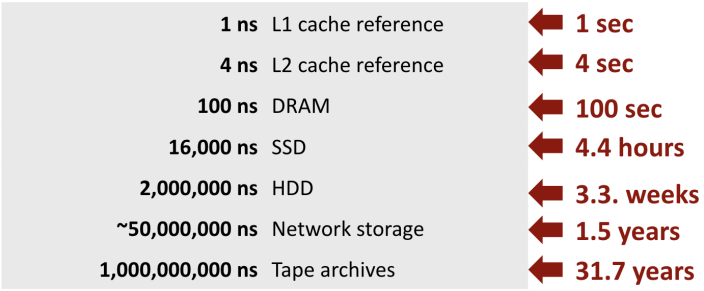
8.7.2 The Storage Hierarchy

Modern computing systems organize storage into a hierarchy based on speed, capacity, and cost: *Main memory for currently used data* and *Disk for storing application data*



8.7.3 Performance Considerations: Latency

Understanding storage performance requires awareness of access latency across different technologies:



These latency figures are critical knowledge for system designers and software engineers when optimizing data access patterns.

8.7.4 Disk Storage Characteristics

Magnetic and solid-state disks serve as the predominant secondary storage devices in modern systems:

Definition (Disk Block). *A disk block (or page) is the fundamental unit of data storage and retrieval for disk-based storage systems.*

Key characteristics of disk storage include:

- Non-uniform access times, unlike RAM
- Access time variations based on disk technology (magnetic vs. flash)
- Performance differences between sequential and random access patterns

The relative placement of data on physical disks significantly impacts application performance, making storage optimization a critical consideration in system design.

8.7.5 Data Integrity in Storage Systems

Modern storage systems employ various techniques to ensure data integrity:

- Error detection and correction codes to handle bit errors
- Capabilities to detect data corruption
- Error handling at both controller and OS levels

Despite these safeguards, storage devices remain a potential single point of failure in computing systems, with limitations in performance, capacity, and reliability.

8.8 Redundant Array of Inexpensive Disks (RAID)

Definition (Redundant Array of Inexpensive Disks (RAID)). *RAID (Redundant Array of Inexpensive Disks) is a storage virtualization technology that combines multiple physical disk drives into a single logical unit for improved performance, capacity, or reliability.*

8.8.1 Storage System Requirements

Effective storage systems must satisfy three key requirements:

- Speed: Data must be accessible with minimal latency
- Reliability: Retrieved data must match what was originally stored
- Affordability: Cost must be reasonable relative to system requirements

RAID technology addresses these requirements by building logical storage volumes from multiple physical disks, providing:

- Higher throughput through data striping
- Improved reliability through redundancy
- Cost-effective scaling of storage capacity

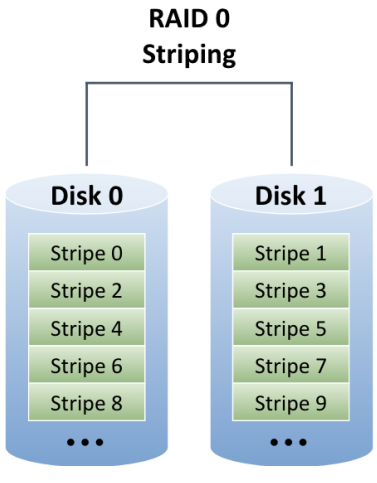
8.8.2 RAID 0: Striping

RAID 0 focuses on performance optimization by distributing data across multiple disks, allowing parallel access to improve throughput and reduce access times.

Definition (RAID 0). *RAID 0 implements block-level striping without parity or mirroring, distributing data evenly across multiple disks without redundancy.*

Characteristics of RAID 0:

- Files are striped across multiple disks
- No data redundancy or fault tolerance
- Provides maximum performance and full storage capacity
- Cumulative bandwidth across all member disks
- Total storage capacity equals the sum of all disk capacities
- Reduced reliability as disk count increases (higher probability of failure)



Example 8.8.2.1. *In a four-disk RAID 0 array, each with a mean time between failures (MTBF) of 100,000 hours, the expected MTBF for the entire array would be approximately 25,000 hours (one-fourth of a single disk's MTBF), since failure of any single disk results in data loss for the entire array.*

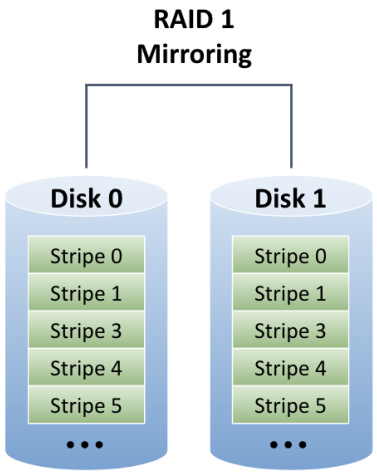
8.8.3 RAID 1: Mirroring

RAID 1 focuses on data reliability through complete redundancy, creating exact copies of data across multiple disks to protect against hardware failures.

Definition (RAID 1). *RAID 1 implements disk mirroring, where identical data is written to two or more drives, providing complete data redundancy.*

Characteristics of RAID 1:

- Data blocks are duplicated across multiple drives
- Excellent protection against disk failure
- Does not protect against data corruption
- Usable storage capacity equals that of a single disk
- Read operations can be parallelized
- Write performance equivalent to a single disk
- Higher cost per usable gigabyte
- Commonly used for critical systems and sensitive information

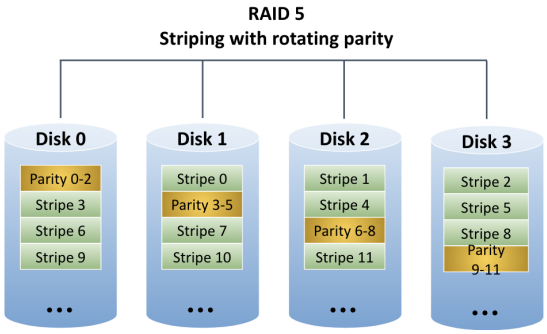


8.8.4 RAID 5: Distributed Parity

RAID 5 balances performance and reliability by distributing both data and parity information across all disks in the array, providing fault tolerance with better storage efficiency than mirroring.

RAID 5 has the following features:

- Distributed parity blocks across all disks
- Can survive failure of any single disk
- Data can be reconstructed using XOR operations on remaining drives
- Good read performance: approximately (N-1) times that of a single disk
- Write operations are more complex due to parity calculations
- Storage efficiency: usable capacity is (N-1) disks in an N-disk array
- Commonly used in data center environments



Definition (RAID 5). *RAID 5 implements block-level striping with distributed parity, providing fault tolerance while using less storage for redundancy than mirroring.*

Definition (Parity). *Parity is a fault tolerance mechanism where redundant data is calculated and stored to enable recovery from single-disk failures. For a set of blocks S_i through S_j, the parity P_{i-j} is calculated as: $P\_i - j = S\_i \oplus S\_i + 1 \oplus ... \oplus S\_j$, where $\oplus$ represents the XOR operation.*

Example 8.8.4.1. *In a 5-disk RAID 5 array, if disk 3 fails, its data can be reconstructed by performing XOR operations on the corresponding blocks from the other 4 disks. This allows the system to continue operation despite the failure, though with degraded performance until the failed disk is replaced and rebuilt.*

Chapter 9

Introduction to CPU Scheduling

9.1 The Need for Scheduling

In modern computing systems, physical resources are limited, necessitating efficient methods to share these resources among multiple processes and threads. CPU scheduling is a fundamental concept in operating systems that addresses how to allocate processor time among competing tasks.

9.1.1 Resource Sharing Approaches

There are two main approaches to achieve resource sharing in computing systems:

- **Time Sharing** — Running one task at a time and rapidly switching among multiple tasks. In this approach, each task gets exclusive access to the resource for a limited time period.
- **Space Sharing** — Dividing the available resource so that each task receives a portion of the total space simultaneously.

9.2 Fundamentals of CPU Scheduling

The primary goal of CPU scheduling is to create the illusion that each thread has exclusive use of the processor, while in reality, the CPU is being shared among multiple threads. This illusion is maintained through efficient time sharing of the CPU resource.

9.2.1 Thread Types and Scheduling

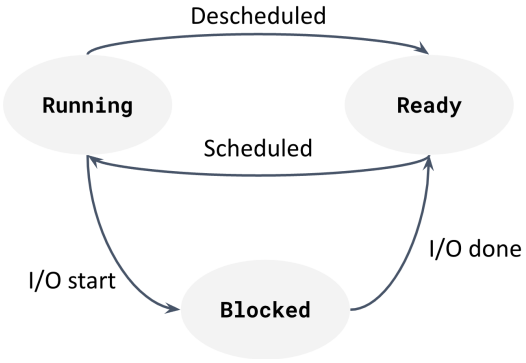
Threads can be categorized based on their operation patterns:

- **CPU-bound threads**: Perform computations with minimal I/O calls (e.g., calculating Fibonacci numbers)
- **I/O-bound threads**: Frequently make I/O calls (e.g., reading from disk, waiting for network)

9.2.2 Thread States

Thread states represent the different operational conditions a thread can be in during its lifecycle.

- **Running**: The thread is currently being executed by the CPU.
- **Ready**: The thread is waiting to be executed by the CPU.
- **Blocked**: The thread is waiting for an event to occur (e.g., I/O completion, message arrival).



9.2.3 Role of the Operating System Scheduler

The operating system's scheduler is responsible for:

- Maintaining a list of all threads in the system
- Tracking each thread's state (running, ready, blocked, etc.)
- Selecting which thread to run next according to a defined scheduling policy
- Managing context switches to give each thread its turn on the CPU

9.2.4 Reasons for Thread Scheduling

The operating system may need to schedule a new thread for various reasons:

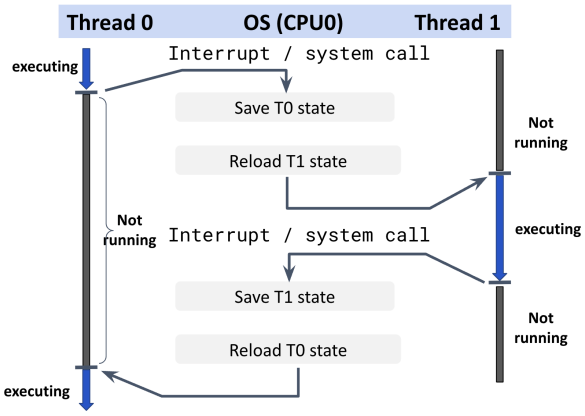
- The current thread has completed execution or terminated (e.g., due to invalid operations)
- The thread has made a system call (e.g., I/O operation) and must wait for its completion
- The OS scheduler has determined another thread should run (e.g., time slice expired)
- Other threads with higher priority are present in the ready queue

9.2.5 Context Switching

Context switching is the process of saving the state of a currently running thread and loading the state of another thread. This mechanism enables time-sharing of the CPU resource.

The OS performs the following operations during a context switch:

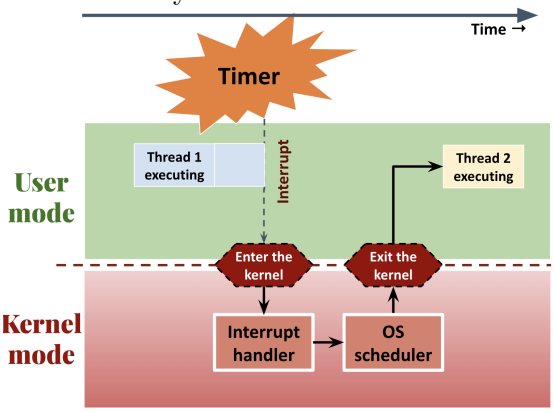
1. Saves the running thread's execution state (registers, program counter, etc.) in memory
2. Selects the next thread to run according to the scheduling policy
3. Restores the execution state of the selected thread
4. Passes control to the thread using a return-from-trap mechanism



9.2.6 Handling Misbehaving Threads

A critical challenge in operating systems is managing threads that may:

- Refuse to give up CPU control on their own
- Run in infinite loops without performing I/O operations
- Attempt to monopolize system resources



To address this challenge, operating systems implement preemptive scheduling using hardware timer interrupts. The process works as follows:

1. The OS sets a hardware timer before scheduling a thread
2. When the timer expires, the CPU is interrupted
3. The current thread is suspended and the system switches to kernel mode
4. The interrupt handler invokes the OS scheduler
5. The scheduler selects the next thread and performs a context switch

This mechanism ensures that no single thread can monopolize the CPU indefinitely, maintaining fairness in resource allocation.

9.3 Scheduling Policies

The scheduling policy is a key component of the operating system that determines which thread should run next. When a system has multiple threads competing for CPU time, the policy establishes the order in which threads execute.

9.3.1 Scheduling Metrics

To evaluate and compare different scheduling policies, we use performance metrics that quantify system behavior. Two fundamental metrics are:

1. **CPU Utilization**: The fraction of time the CPU is executing thread code.
   - **Goal**: Maximize CPU utilization (keep the CPU as busy as possible)
   - Measured as a percentage from 0% (idle) to 100% (fully utilized)
2. **Turnaround Time**: The total time from a thread's arrival to its completion.
   - **Goal**: Minimize turnaround time for better system responsiveness
   - Mathematical definition: $T_{turnaround} = T_{completion} - T_{arrival}$

9.3.2 First In, First Out (FIFO)

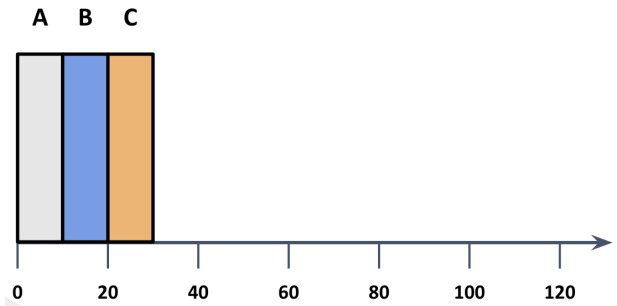
FIFO (also known as First Come, First Served) is the simplest scheduling algorithm where threads are executed in the order they arrive in the ready queue.

FIFO with Equal Run Times

Consider a scenario with three threads (A, B, C) that arrive simultaneously and each requires 10 seconds of CPU time

Assumptions:

- Each thread runs for the same time (10s)
- All threads arrive at the same time ($T_{arrival} = 0$)
- Each thread runs to completion
- Run-time of threads is known in advance



Calculations:

$$T_{arrival} = 0$$
$$T_{completion}(A) = 10$$
$$T_{completion}(B) = 20$$
$$T_{completion}(C) = 30$$

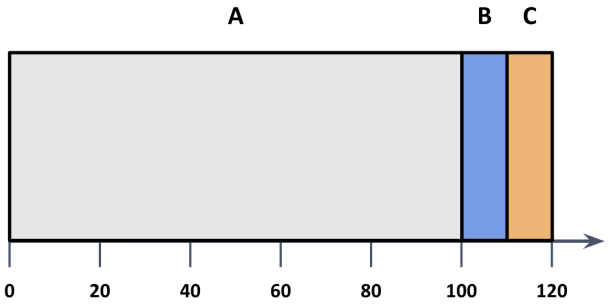
Average turnaround time $= \frac{10+20+30}{3} = 20$ seconds

FIFO with Varied Run Times

Now consider a scenario where thread A requires much more CPU time than the others:

Assumptions:

- Threads have different run times
(A: 100s, B: 10s, C: 10s)
- All threads arrive at the same time ($T_{arrival} = 0$)
- Each thread runs to completion
- Run-time of threads is known in advance



Calculations

$$T_{arrival} = 0$$
$$T_{completion}(A) = 100$$
$$T_{completion}(B) = 110$$
$$T_{completion}(C) = 120$$

Average turnaround time $= \frac{100+110+120}{3} = 110$ seconds

In FIFO scheduling, long-running threads can significantly delay shorter threads, causing poor average turnaround times. This is known as the "convoy effect" and represents a major limitation of the FIFO scheduling policy

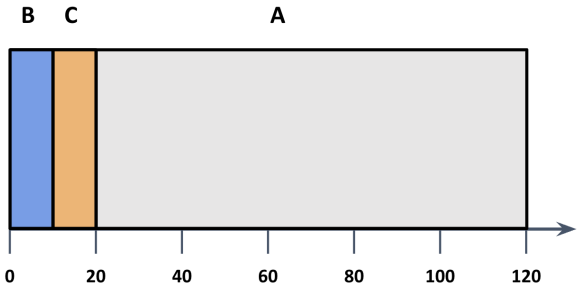
basically, the classic - someone in a supermarket queue is buying a lot of stuff and you're waiting for them to finish even though you have less items than them

9.3.3 Shortest Job First (SJF)

Choose ready threads with shortest running time

Assumptions:

Assume 3 threads (A, B, C): A runs for 100 seconds, while B and C run 10 seconds



Calculations:

$$T_{arrival} = 0$$
$$T_{completion}(A) = 120$$
$$T_{completion}(B) = 10$$
$$T_{completion}(C) = 20$$

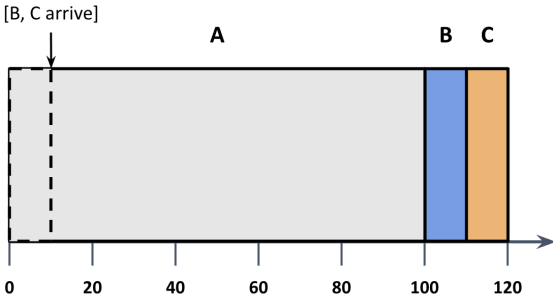
Average turnaround time $= \frac{120+10+20}{3} = 50$ seconds

Turnaround time improves by almost 50%

Issue with SJF

A runs for 100 seconds, while B and C run 10 seconds **Assumptions:**

- Threads do not need to run for same time
- Threads do not need to arrive at same time
- Each thread runs to completion
- Run-time of threads is known



Calculations:

$$T_{arrival}(A) = 0$$
$$T_{arrival}(B) = T_{arrival}(C) = 10$$
$$T_{completion}(A) = 100$$
$$T_{completion}(B) = 110$$
$$T_{completion}(C) = 120$$

Average turnaround time $= \frac{100+(110-10)+(120-10)}{3} = 103.3$

Remark: Long running threads cannot be interrupted, leading to convoy effect

9.3.4 Polite vs. forced scheduling

Non-preemptive Scheduling:

- Previous schedulers (FIFO, SJF) are non-preemptive
- Only switch to other threads once the current thread finishes its whole execution (run-to-completion)
- OS has no control on a thread's completion time

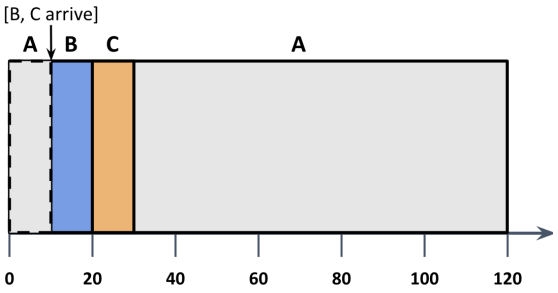
Preemptive Scheduling:

- Stops the execution of the current thread and switches to other ready thread forcibly
- OS avoids CPU monopolization and maintains control (thread create/destroy, timer interrupts)
- **This removes the assumption that each job runs to completion**

9.3.5 Shortest Time to Completion First (STCF)

STCF extends the SJF by adding preemption
Assumptions:

- Any time a new thread is created:
- STCF scheduler determines which of the remaining jobs (including new job) has the least time left
- STCF then schedules the shortest job first



Calculations:

$$\begin{aligned} T_{arrival}(A) &= 0 \\ T_{arrival}(B) &= T_{arrival}(C) = 10 \\ T_{turnaround}(A) &= 120 \\ T_{turnaround}(B) &= (20 - 10) = 10 \\ T_{turnaround}(C) &= (30 - 10) = 20 \end{aligned}$$

Average turnaround time $= \frac{120+10+20}{3} = 50$ seconds
Remark: Reschedule when new threads arrive, prioritize short running threads

9.3.6 New Metric - Response Time

Previous metrics:

- Focused only on turnaround time (i.e., completing the threads' execution as fast as possible)
- Turnaround time is important for batch jobs (non-interactive tasks)

New metric:

- Response time became equally important
- Defined as how long it takes until a thread is scheduled for the first time

STCF with Response Time

For example, for the STCF
Response time: Time from when the job arrives in the system to the first time it is scheduled:
$T_{response} = T_{firstrun} - T_{arrival}$

Calculations:

$$\begin{aligned} T_{arrival}(A) &= 0 \\ T_{arrival}(B) &= T_{arrival}(C) = 10 \\ T_{response}(A) &= (0 - 0) = 0 \\ T_{response}(B) &= (10 - 10) = 0 \\ T_{response}(C) &= (20 - 10) = 10 \end{aligned}$$

Average response time $= \frac{0+0+10}{3} = 3.3$ seconds

STCF is still not perfect

Prior scheduling policies are not good for response time:

Consider 3 jobs arrive at T=0 with same running time, the third job has to wait for the previous two jobs before getting scheduled!
This is great for turnaround time, but bad for interactivity.
Another way to think: typing on a keyboard and waiting **seconds** for the character to show up on the screen...

9.3.7 Round Robin Scheduling

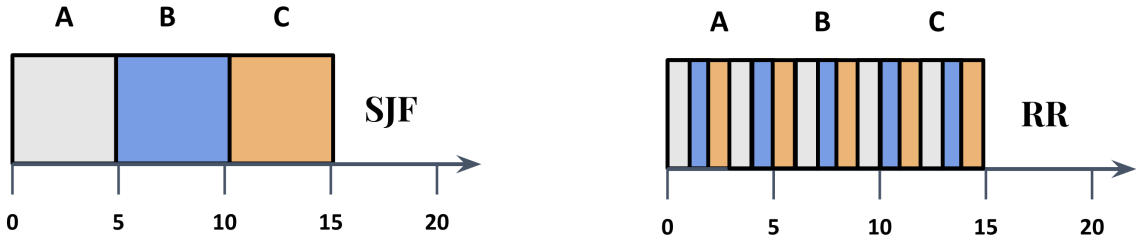
Instead of running threads to completion, RR schedules a thread for a fixed interval (or a time-slice) and then switches to the next thread Alternate ready threads every fixed-length time slice.

Round Robin vs STCF

Threads A, B, and C run for 5 seconds each and arrive at time 0, $T_{arrival} = 0$

- In SJF, each thread runs to completion before running another
- Average response time = (0 + 5 + 10) / 3 = 5
- Average turnaround time = (5 + 10 + 15) / 3 = 10

- In RR, time slice is 1 second and it will run each thread every second
- Average response time = (0 + 1 + 2) / 3 = 1
- Average turnaround time = (13 + 14 + 15) / 3 = 14



Remark: Responsiveness increases turnaround (for equally long running threads)

9.3.8 IO Request Scheduling

In operating systems, we need to consider how IO operations affect CPU scheduling. Let's examine a simple example:
Assume we have two threads (A and B):

- Thread A needs 40 ms of CPU time and makes 3 IO requests of 10 ms each
- Thread B needs 40 ms of CPU time and makes no IO requests
- Thread A issues an IO request every 10 ms of CPU time

When thread A issues IO requests, the CPU is not utilized efficiently if the scheduler doesn't account for this behavior.

Scheduling with IO Awareness

Question: For a Shortest Time-to-Completion First (STCF) scheduler, how should we handle thread A's 4 sub-jobs (10 ms each) versus thread B's single 40 ms job?

Answer: A better approach is for the scheduler to account for both IO and CPU time to improve resource utilization:

- Treat each of A's 10 ms segments as independent sub-jobs
- Consider B as a whole 40 ms job
- STCF chooses A's first sub-job (10 ms) and then schedules B
- When A's next sub-job becomes ready, the scheduler preempts B
- While A waits for IO completion, B can run on the CPU
- This leads to better CPU utilization through overlapping CPU and IO operations

9.3.9 Multi-level Queue Scheduling (MLFQ)

The Scheduling Challenge

A general-purpose scheduler must support different types of threads:

- **Batch-processing threads:** Long-running background processes that need lots of CPU time but where response time is not critical
- **Interactive threads:** Foreground processes that require low latency and run in short bursts (need frequent but small amounts of CPU time)

MLFQ Approach

MLFQ aims to optimize for both types of threads:

- First, it tries to optimize turnaround time (important for batch threads)
- Then, it minimizes response time for better interactivity

The challenge is that the scheduler doesn't know the total runtime of a thread in advance (which would be needed for SJF or STCF). The insight of MLFQ is to use past behavior as a predictor for future behavior.

MLFQ Implementation

MLFQ uses multiple levels of Round Robin queues:

- Each level has a different priority
- Higher levels preempt lower levels
- Higher levels have shorter time slices
- Lower levels have longer time slices

MLFQ Rules

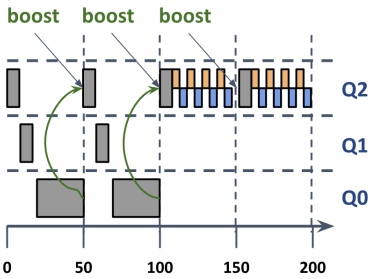
The scheduler follows these rules to adjust priorities dynamically:

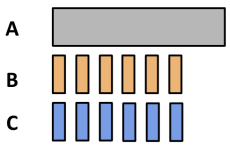
1. If priority(A) > priority(B), then A runs
2. If priority(A) = priority(B), then A and B run in Round Robin
3. All threads start at the highest priority level
4. If a thread uses its entire time slice, the scheduler lowers its priority
5. Periodically, all threads are moved back to the highest priority queue (called "priority boosting")

The periodic priority boosting (rule 5) prevents starvation, which could happen when multiple IO-bound threads at high priority might prevent a CPU-bound thread at low priority from ever running.

Example 9.3.9.1 (MLFQ in Action)**.** Let's see how MLFQ works with a concrete example. Assume we have a system with three priority queues (Q2, Q1, Q0), where Q2 is the highest priority:

- Each queue uses Round Robin scheduling
- Time slice for Q2 = 10 ms, Q1 = 10 ms, Q0 = 30 ms
- Priority boost happens every 50 ms





Phase 1: Single CPU-bound process

1. Process A begins in Q2 (highest priority)
2. A uses its entire 10 ms time slice in Q2 and gets demoted to Q1
3. A uses its entire 10 ms time slice in Q1 and gets demoted to Q0
4. A runs in Q0 for 30 ms until the priority boost occurs
5. After the boost, A returns to Q2 and the cycle repeats

Phase 2: Interactive processes arrive

1. After A has been running for 100 ms, processes B and C join the system
2. All three processes (A, B, C) are now in Q2
3. A runs for 10 ms and gets demoted to Q1
4. Now B runs for a short time but issues an IO request before using its full time slice
5. C also runs briefly before issuing an IO request

6. Since B and C issue frequent IO requests, they never use their full time slices
7. B and C therefore remain in Q2, while A continues to be demoted

Result: Interactive processes (B and C) get quick response times by staying in the high-priority queue, while the CPU-intensive process (A) is given fair access through priority boosting. This demonstrates how MLFQ adapts to different process types without requiring advance knowledge of their behavior.

Good Luck for the midterm !

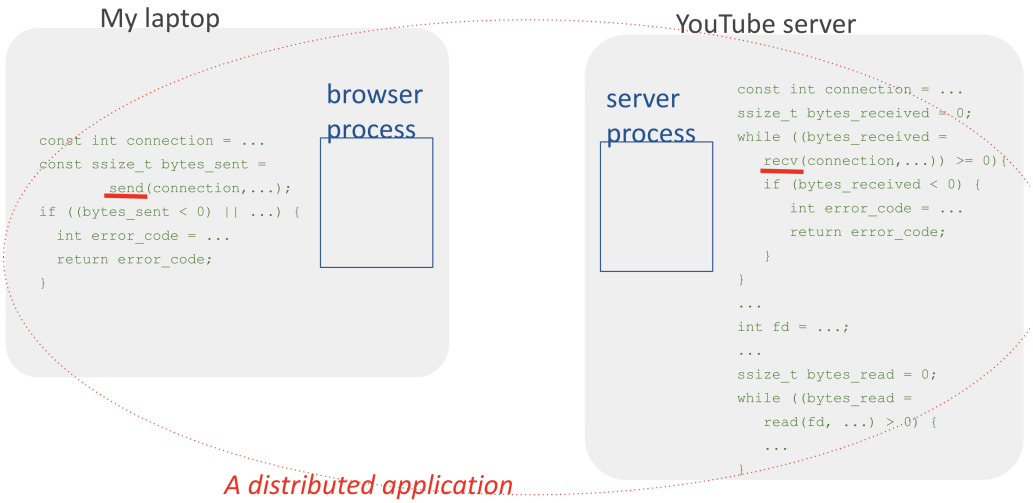
Chapter 10

Client/Server Model & The Web

Network ! Back to our Youtube example. (sorry for the delay was working on something else, I hope midterm went not too bad.)

Introduction

This chapter introduces the client/server model, a foundational concept of the modern web. We use YouTube as an example: when you play a video, your browser (client) communicates with a YouTube server to stream content. These two separate processes exchange messages over the network, forming a distributed application.



10.1 Distributed Applications

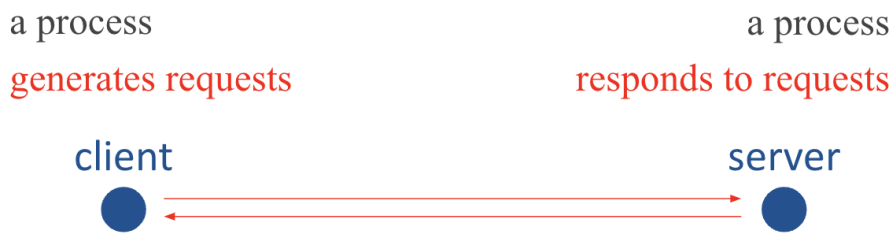
A **distributed application** consists of:

- Processes running on different computers,
- Exchanging messages over a network,
- Collaborating to achieve a shared goal.

10.2 Client/Server Architecture

Most distributed applications use a client/server architecture:

- One process acts as the **client**, making requests.
- Another acts as the **server**, responding to requests.

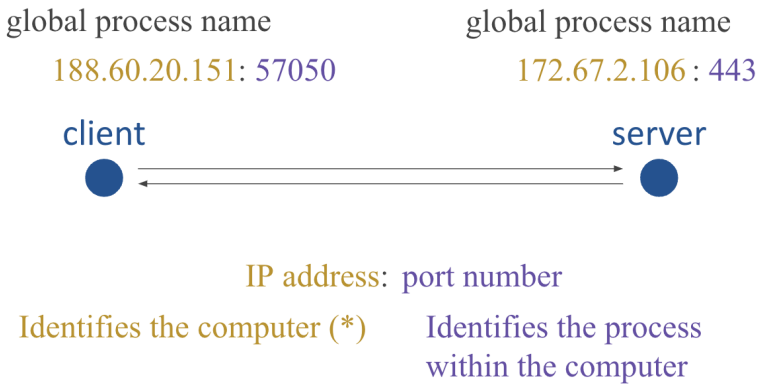


Clients and servers have clearly defined roles. Servers typically run on dedicated infrastructure, which today often means multiple data centers and many server processes.

10.2.1 Naming and Identifying Processes

To communicate, client and server processes need unique identifiers:

- **Local identifiers** (e.g., process ID or PID) only have meaning within a single computer.
- **Global identifiers** are needed for communication across computers.



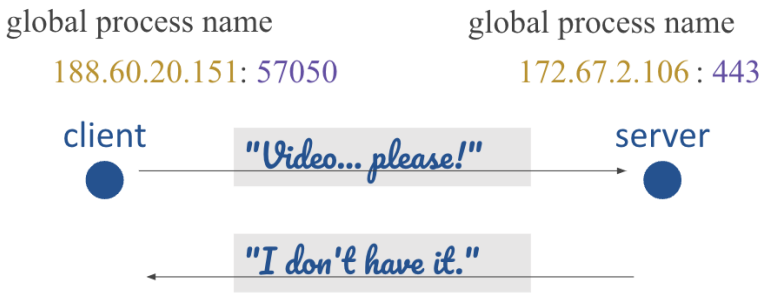
A global process name consists of:

- An **IP address** — uniquely identifies a computer (e.g., `172.67.2.106`),
- A **port number** — uniquely identifies a process on that computer (e.g., `443`, `57057`).

An IP address is like a street name; a port number is like a house number. Together, they uniquely identify a process on the network.

10.2.2 Discovering the Server Process

Before communication can begin, the client must discover the server's global name (IP address and port number). Once known, the client can initiate contact by introducing itself to the server.



Suppose the client requests a video from the server, but the server does not have the requested video. The server replies with a message indicating the resource is not found (for example, an HTTP 404 error).

If the server has the video, it responds with the requested data (such as an HTTP 200 OK response), and the client can begin receiving the video stream.

If the client does not follow the expected protocol—such as repeatedly sending requests for the same video without waiting for a response—the server may choose to ignore these requests or terminate the connection. This is a **protocol violation**.
Once the client knows the server's address, they can exchange messages according to a **communication protocol**.

10.2.3 Communication Protocols

A **communication protocol** defines the set of valid interactions between client and server processes. For example, a typical protocol might allow:

- The client requests information; the server responds with the requested data.
- The client requests information; the server responds that the information is unavailable.

If either party sends an unexpected or invalid message—such as the client repeating the same request rapidly—the protocol is violated. In such cases, the server may choose not to respond or may close the connection.

Analogy: Communication protocols in computers are similar to social protocols in conversation. If someone repeatedly asks questions without waiting for answers, the other person may stop responding.

Summary: Communication between distributed processes is only possible when the client can discover the server's address and both sides follow an agreed protocol. Violating the protocol typically ends the communication.

10.3 The HyperText Transfer Protocol (HTTP)

The HyperText Transfer Protocol (HTTP) is the foundation of communication on the World Wide Web. It defines how web clients (e.g., browsers) and web servers exchange information.

10.3.1 HTTP Requests and Responses

- **Request**: A web client sends an HTTP request to a server to retrieve or manipulate resources.
- **Response**: The web server processes the request and returns an HTTP response.
- **Common Request Methods**:
  - `GET`: Retrieves a specified resource.
  - `HEAD`: Retrieves metadata about a resource without the resource itself (e.g., object size or type).
  - `POST`: Sends data to the server, often used for form submissions.
- **Common Response Status Codes**:
  - `200 OK`: Request was successful.
  - `404 Not Found`: Requested resource could not be found.
  - `400 Bad Request`: Request was malformed.
  - `301 Moved Permanently`: Resource has been relocated to a new URL.

10.3.2 Web Objects

A web object is any resource accessible on the web, identified by a unique Uniform Resource Locator (URL).

- **Types**: Text files, images, videos, scripts, etc.
- **URL Example**: `https://actu.epfl.ch/image/142932/1920x1080.jpg`
- **Access**: Retrieved by a web client from a server using an HTTP request (e.g., `GET /image/142932/1920x1`
- **Uniqueness**: Each URL serves as a globally unique identifier for the resource.

10.3.3 Web Pages

A web page is a specific type of web object, typically composed of multiple resources.

- **Structure**: Consists of a base file (e.g., HTML) that defines the page's structure and references other objects.
- **Referenced Objects**: May include images, videos, scripts, or other web pages.
- **Example**: A web page might include an HTML file, CSS stylesheets, and embedded images, all fetched via separate HTTP requests.

10.3.4 Designing a Distributed Application

When developing a distributed application, it is essential to:

- Decide how many processes (and threads) are needed, and define the role of each.
- Design the communication protocol between the participating processes and threads.

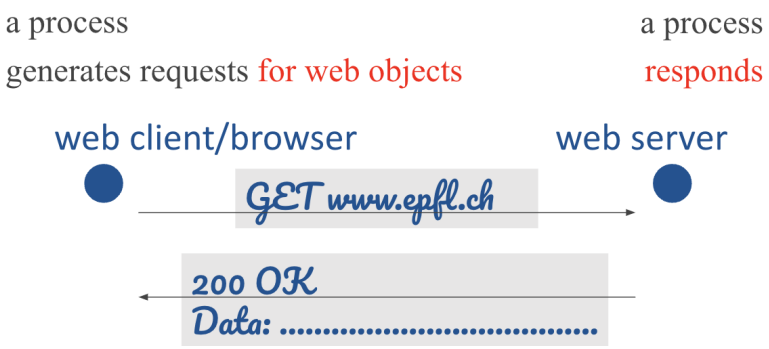
These decisions should be made before any coding begins.

A common and universal example of a client/server application is the web:

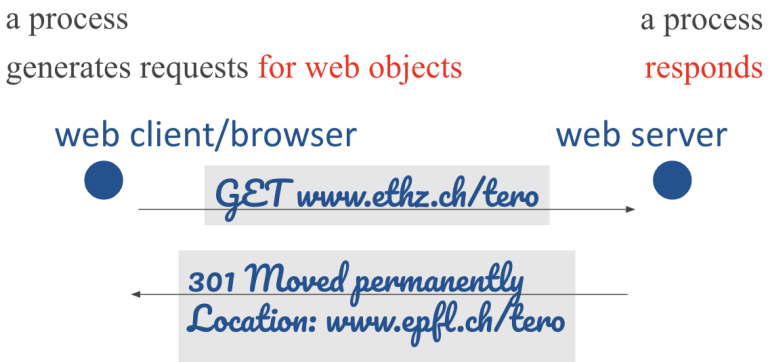
- **Web clients** (web browsers) generate requests for web resources.
- **Web servers** respond to these requests.

The communication protocol between a web client and a web server is the HyperText Transfer Protocol (**HTTP**), which is based on simple request and response interactions.

The most frequent request type is a `GET` request. The server can reply in several ways, each shown below:



When the server has the requested resource, it replies with a `200 OK` response, providing the object to the client.



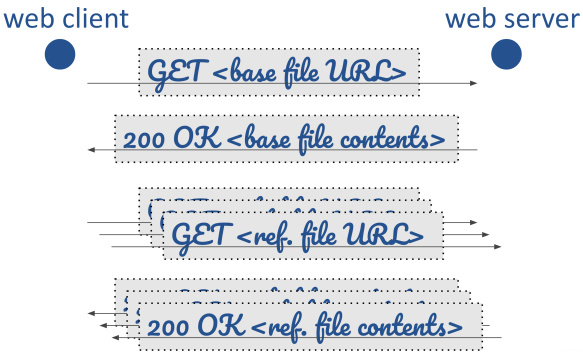
If the resource has been moved, the server replies with a `301 Moved Permanently` response, including the new location.

If the client's request is malformed or invalid, the server responds with a `400 Bad Request` error. This is rare in practice, because web browsers typically generate well-formed requests on behalf of users.

If the requested resource does not exist, the server sends a `404 Not Found` error.

Note: Most protocol errors are handled by the browser itself and are not directly seen by the user.

10.3.5 Example: Common Web Client/Server Exchanges



- A human user types a URL into their web client (e.g., a browser).
- The web client sends a `GET` request for the URL, which identifies the base file of a web page.
- The web server responds with a `200 OK` status, including the content of the requested base file.
- The web client parses the base file, discovers all additional URLs (e.g., images, scripts, stylesheets) needed to display the page, and sends a `GET` request for each one.

When you visit a web page, your web client sends multiple `GET` requests to the web server and receives multiple responses to fully render the page.

10.3.6 Stateless Protocols

- A server process does **not** maintain any information (or *state*) about previous interactions with a client.
- Here, *state* means data saved from past communication exchanges.
- By design, HTTP is a **stateless** protocol: each request from a client to a server is treated independently.

Let us contrast **stateless** versus **stateful** protocols:

- In a *stateful* interaction, the server remembers past conversations and can tailor responses accordingly.
- For example, in human conversation (like a classroom lecture), we recall previous topics discussed.
- In contrast, some services like a hospital system can be stateless: a new doctor might ask you to explain your medical history again because no information was saved.

If HTTP is stateless, then how do websites like Facebook or Amazon recognize you when you return, sometimes even before you log in?

10.3.7 Example: How Cookies Enable State

Suppose you open your web client and enter the URL `news.com/greece.html`.

- Your web client sends a `GET` request to `news.com` for the page `greece.html`.
- The web server responds with the page content and includes a **cookie** — a small piece of metadata that carries information about you or your preferences (in this case, that you are interested in Greece).
- Your web client stores this cookie.
- On subsequent requests to `news.com`, your client sends this cookie back to the server.
- The server reads the cookie, learns your interest in Greece, and can customize the response (e.g., show Greek recipes).
- The server can update or add new cookies to store additional inferred interests.

Over time, the server builds a profile about you based on cookies stored on your client, enabling a personalized experience despite HTTP being stateless.

10.4 Cookies

Cookies represent **state created by the web server but stored on the web client**. They serve as a mechanism to link multiple web requests from the same client, enabling session continuity despite HTTP's stateless nature. Through cookies, websites can recognize returning visitors and maintain user preferences across multiple visits.

10.4.1 Passing State to the Client

- Instead of storing client-specific information on the server, the server **passes** that state to the client.
- The client stores this state (in cookies) and sends it back on future requests.
- This approach reduces server memory and storage requirements.
- It also simplifies server design by offloading state management to the client.

Professor's Analogy (quoted): When I lived in Greece, clubs charged entrance fees but did not give out tickets. Instead, they stamped a client's forearm. If the client left and returned (e.g., to smoke), showing the stamp proved they had already paid.
Here, the club "passed the state to the client" by letting the client carry the proof themselves, rather than maintaining a list or checking IDs repeatedly. This simplified the job and reduced the club's need to keep state information.
Similarly, cookies pass state information from servers to clients, enabling a stateless HTTP protocol to behave like a stateful system.
Example of Cross-Site Information Transfer:
Continuing the previous example:

- **Initial Visit & Cookie Association**: A web client visits 'news.com'. The response from the 'news.com' server (or an embedded element therein) causes the client's browser to store a cookie that is associated with a different server, say 'cooking.com'.
- **Subsequent Visit to Different Server**: When the client later sends its first HTTP request to 'cooking.com', the browser automatically includes this specific cookie with the request.
- **Information Gained by Third Party**: As a result, the 'cooking.com' server receives this cookie. It can then learn information about the client (e.g., an inferred interest in "Greece," perhaps based on the context from 'news.com' or the cookie's content itself), even though this is the client's very first direct communication with 'cooking.com'.

10.4.2 Third-Party Cookies

A **third-party cookie** is a cookie set by a web server for a domain different from the one the user is currently visiting. This allows information about a user's browsing activity to be shared across different websites.

- **Definition**: A cookie created by one web server (e.g., an ad network) to be used when the client visits web pages hosted by other, different web servers.
- **Mechanism**:
    1. A user visits 'news.com'. The 'news.com' server's response might include instructions for the browser to fetch content from 'ads.com' (e.g., an image or script).

2. When the browser requests content from ‘ads.com‘, ‘ads.com‘ can set a cookie in the user’s browser. This is a third-party cookie from the perspective of ‘news.com‘.
3. Later, if the user visits ‘cooking.com‘, and ‘cooking.com‘ also embeds content from ‘ads.com‘, the browser will send the cookie previously set by ‘ads.com‘ along with the request to ‘ads.com‘.

- **Implication**: The third-party server (‘ads.com‘) can track the user’s visits across multiple sites (‘news.com‘, ‘cooking.com‘), building a profile of their interests even if the user has never directly visited the third-party’s website.

10.4.3 Cookie-less Tracking

Cookie-less tracking refers to techniques used by web servers to collect information about web clients and their activities without relying on traditional HTTP cookies.

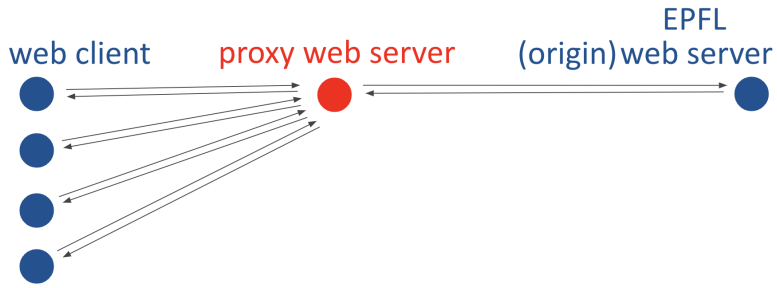
- **Methods**: These can include browser fingerprinting (collecting unique browser configurations), IP address tracking, ETags, or other header information.
- **Privacy Considerations**:
  - A common misconception is that cookie-less tracking is inherently more privacy-preserving simply because it avoids cookies.
  - **This is not necessarily true.** Cookies are just one mechanism for user profiling. The critical factor for privacy is the *nature and extent of the information collected and how it is used*, not the specific mechanism (cookies or otherwise).
  - Cookie-less *tracking* is still *tracking*. Its impact on privacy depends on what data is gathered and for what purpose.

10.5 Web Caching

Web caching is a core technique for improving the performance of web applications by storing copies of frequently accessed resources closer to the user.

10.5.1 Introduction to Web Caching

- A **web cache** (or **proxy web server**) is an intermediary server that stores (caches) copies of web content (e.g., HTML pages, images, files) served by origin web servers.
- It acts as a server to nearby web clients and as a client to the origin web servers.
- **Primary Goals**:
  - **Reduce Delay**: Clients experience faster load times as content is retrieved from a geographically closer cache rather than a distant origin server.
  - **Reduce Load on Origin Server**: Fewer requests reach the origin server, decreasing its workload and bandwidth consumption.



10.5.2 Web Caching Mechanism: An Example

Consider a web client accessing a resource (e.g., 'www.example.com/resource') from a distant origin server. A local proxy web server can be used:

1. **First Client Request (Cache Miss)**:
   - Client 1 sends a GET request for the URL to the nearby proxy server.
   - The proxy checks its local cache. If the resource is not found (a **cache miss**), the proxy forwards the GET request to the origin server.
   - The origin server responds to the proxy with the resource.
   - The proxy stores a copy of the resource in its cache and forwards the resource to Client 1.
   - Client 1 experiences a delay, potentially slightly longer due to the intermediary step.
2. **Subsequent Client Requests (Cache Hit)**:
   - Client 2 (or Client 1 again) requests the same URL from the proxy server.
   - The proxy checks its cache and finds a fresh copy of the resource (a **cache hit**).
   - The proxy immediately sends the cached resource to Client 2, without contacting the origin server.
   - Client 2 experiences significantly reduced delay.

10.5.3 Challenge: Ensuring Data Freshness

A critical challenge in web caching is ensuring that the cached data is not **stale** (i.e., an outdated version of the resource). Clients should receive the most current version.

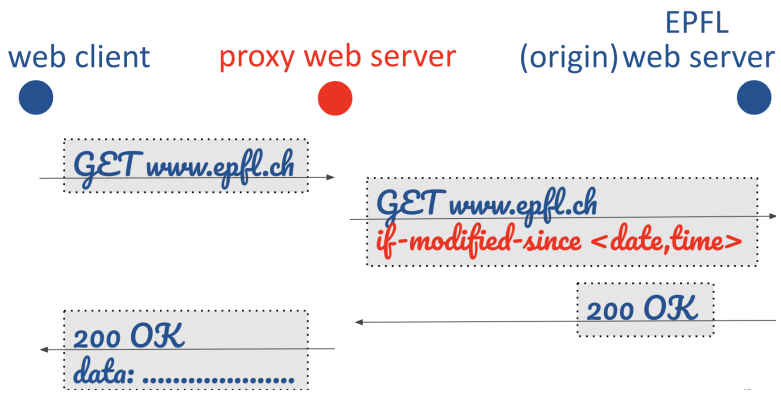
Cache Validation Mechanisms

To address stale data, caches use validation mechanisms:

- **Expiration Time / Max Caching Age**:
  - Origin servers can include HTTP headers in their responses to specify how long a resource can be considered fresh.
  - 'Expires': Provides a specific date/time after which the resource is stale.
  - 'Cache-Control: max-age=¡seconds¿': Specifies the maximum time in seconds that the resource can be cached without revalidation.
  - Once this period elapses, the cache considers its copy stale.
- **Conditional GET Requests**:
  - When a cached resource is stale (or if the cache wants to verify freshness), it can send a **conditional GET request** to the origin server.
  - This request asks the server to send the resource *only if it has been modified* since the version stored in the cache.
  - The 'If-Modified-Since' HTTP request header is used, containing the 'Last-Modified' timestamp of the cached version.

```
GET /resource.html HTTP/1.1
  Host:  www.example.com
If-Modified-Since:  Wed, 21 Oct 2023 07:28:00 GMT
```

  - **Server Response**:
    - If the resource *has not been modified* since the specified date, the origin server responds with 'HTTP/1.1 304 Not Modified'. This response has no body, saving bandwidth. The cache can then serve its stored copy.
    - If the resource *has been modified*, the origin server responds with 'HTTP/1.1 200 OK' and the new version of the resource, which the cache then stores and forwards to the client.



Note: Web clients (browsers) also maintain their own local caches and employ similar mechanisms (expiration policies, conditional GETs) for resources they fetch.

10.5.4 Impact of Conditional GET on Performance

Does a conditional GET request significantly reduce delay?

- **Not always significantly for delay**: A conditional GET still requires a round-trip to the origin server to check for modifications. If the object is small, the time saved by not re-downloading it (in case of a '304 Not Modified') might be marginal compared to the round-trip time itself.
- **Significant for bandwidth and large objects**: If the requested object is large, receiving a '304 Not Modified' response avoids re-transmitting the entire object, leading to substantial savings in bandwidth and a noticeable reduction in delay compared to a full download.

10.5.5 General Principles of Caching

- Caching is a **universal technique** for improving performance in systems where data is accessed repeatedly.
- **Core Idea**: When one entity incurs the cost to fetch data, cache it locally (or closer to other potential consumers) so that subsequent requests for the same data can be served faster and with less resource consumption on the origin.
- **Primary Challenge**: Ensuring **data freshness** or **cache coherency**—that the cached data accurately reflects the current state of the origin data.
- **Common Solutions**:
  - Assigning an **expiration date** or **max caching age** to data.
  - Performing **dynamic checks** (e.g., conditional GETs) with the origin source to validate freshness. The utility of dynamic checks can depend on factors like data size and communication overhead.

Chapter 11

L11 - DNS & P2P

11.1 Global Process Naming and Addressing

Every process communicating over the Internet is uniquely identified by a combination of an IP address and a port number.

```
172.67.2.106:443
```

- **172.67.2.106**: IP address of a network interface.
- **443**: Port number (details below).

11.1.1 Network Interfaces

A network interface connects an end-system (such as a computer or smart device) to a network.

- Each end-system has at least one network interface.
- Each network interface has at least one unique IP address.
- Examples: Your laptop's network card, or your car's onboard Wi-Fi.

11.1.2 Port Numbers

Port numbers identify specific processes within a local system.

- Each process using network communication is assigned a unique port number.
- On a server, port numbers for certain applications are restricted and standardized.

Reserved Port Numbers

Some port numbers are universally reserved for well-known services.

- Reserved ports are used exclusively by specific server processes.
- Client processes and other servers should not use these reserved ports.
- Examples: Ports 80, 8080, 443, and 8443 are reserved for web servers.

11.1.3 Web Server Port Numbers

Web servers use standardized ports for HTTP and HTTPS communication:

- **80, 8080**: HTTP (standard web traffic)
- **443, 8443**: HTTPS (secure web traffic)

Note: HTTPS is the encrypted, secure version of HTTP.

11.1.4 The Domain Name System (DNS)

DNS is an essential Internet system that translates human-friendly domain names into IP addresses.

Example

When you enter a URL such as:

- **https://www.epfl.ch/labs/nal/publications**
- **www.epfl.ch**: Hostname (DNS name)
- **labs/nal/publications**: Path to the resource

The Domain Name System (DNS) maps domain names like `www.epfl.ch` to IP addresses (e.g., `128.178.211.3`). This translation allows users to access network interfaces by name rather than by numeric address, making the Internet more user-friendly and scalable.

DNS Translation in Action

1. The web client reads the URL and identifies the protocol: **https**.
2. For HTTPS, the default port number is **443** (or sometimes 8443).
3. The web client extracts the DNS name (`www.epfl.ch`) and sends a DNS query to obtain its IP address.
4. Once resolved, it can contact the web server process at `128.178.211.3:443`.

11.1.5 Application Design

The design of a distributed application requires answering:

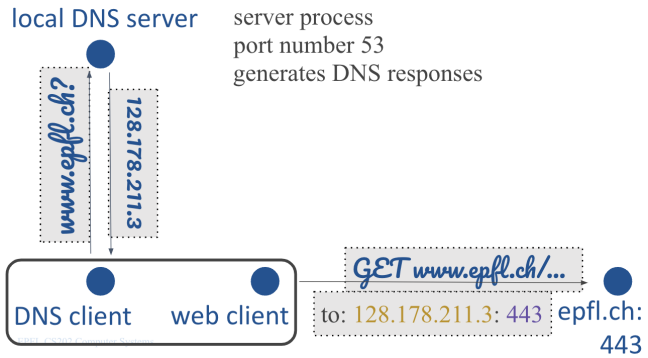
- How many processes/threads are involved, and their roles.
- What communication protocols are used between them.

For DNS, this means defining client and server roles and their communication.

11.1.6 How Does DNS Work?

DNS uses a client-server architecture with specialized processes:

- When you type a URL, your web client extracts the DNS name (e.g., `www.epfl.ch`).
- A local process, called the **DNS client** or **stub resolver**, creates a DNS query.
- The DNS client sends this query to a **local DNS server** (resolver), typically nearby.
- The DNS server process listens on port 53 and replies with the IP address.
- The DNS client relays the response back to the web client.
- The web client can now communicate with the web server using the IP address and correct port.



Why Not a Single DNS Server?

A single-server DNS design would not scale:

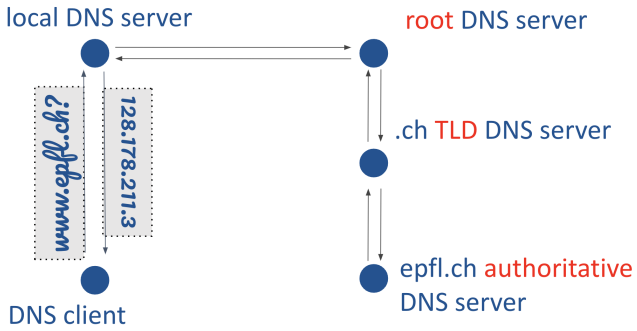
- It would be overloaded by global traffic.
- It could not provide low latency to all clients.
- It would be a single point of failure.
- Maintenance would be unmanageable at Internet scale.

11.1.7 Scalability

Scalability is the ability of a system to grow while maintaining good performance and reasonable cost, independently from size.

11.1.8 Distributed DNS

The Domain Name System (DNS) is not managed by a single server, but as a distributed, hierarchical system. This structure ensures scalability, reliability, and high performance for name resolution on the Internet.

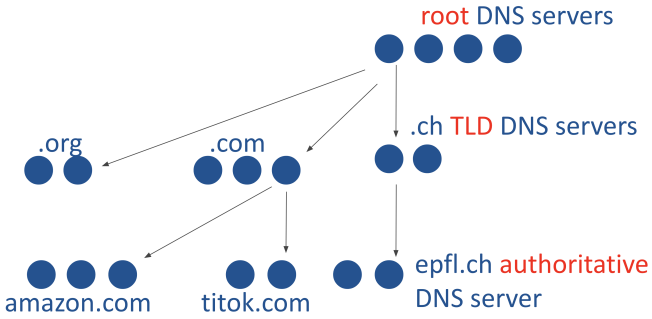


When a DNS client wants to resolve a domain such as `epfl.ch`, the following process occurs:

- The client queries its local DNS server.
- If the local server does not know the answer, it queries a **root DNS server**.
- If the root server does not know the answer, it forwards the query to a **TLD (Top-Level Domain) server** (e.g., for `.ch`).
- If the TLD server does not know, it asks the **authoritative DNS server** for the specific domain (e.g., `epfl.ch`).
- The authoritative DNS server provides the definitive answer for the requested domain.

DNS Hierarchy

DNS servers are organized hierarchically to efficiently manage and resolve domain names. At each level, servers have specific responsibilities.



- **Root DNS servers** are at the top of the hierarchy. They know how to reach all TLD servers.
- **TLD DNS servers** manage domains for top-level domains (e.g., `.ch`, `.com`, `.org`). They know the authoritative servers for all second-level domains under their TLD.
- **Authoritative DNS servers** hold the final, definitive information about their specific domain (e.g., `epfl.ch`, `ricardo.ch`). They know the IP addresses for all hostnames in their domain.

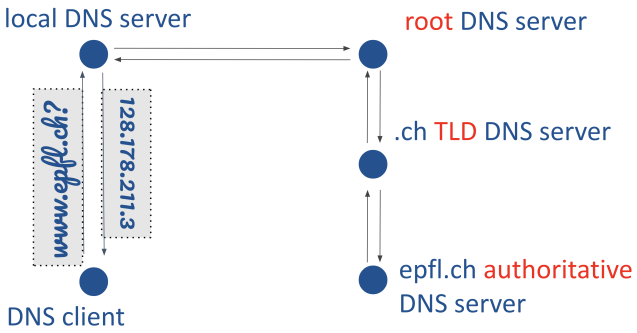
For example, an authoritative server for `epfl.ch` knows the addresses of all hosts ending with `epfl.ch`. Similarly, a TLD server for `.ch` knows how to reach authoritative servers for all `.ch` domains.

11.1.9 DNS Query Resolution

There are two main approaches for a DNS query to reach a server that can answer it. Each approach offers different trade-offs in terms of efficiency and server workload.

Recursive Query

In a **recursive query**, the client asks a DNS server to resolve the name entirely. The server takes responsibility for querying other DNS servers until it obtains the answer, which it then returns to the client.

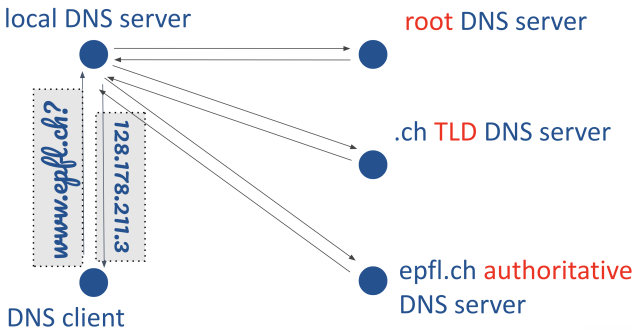


- Every DNS client knows at least one local DNS server.
- Every DNS server knows at least one root DNS server.
- Each root DNS server knows at least one TLD DNS server per TLD.
- Each TLD DNS server knows at least one authoritative DNS server for each second-level domain it manages.

If the hierarchy works correctly, every DNS query should eventually receive a response.

Iterative Query

In an **iterative query**, a DNS server responds to the client with the address of the next server to contact if it cannot answer directly. The client then queries that server, repeating the process until it finds the authoritative answer.



This model is common in practice, often as a mix of recursive and iterative queries: the local DNS server handles recursion for the client, while upstream servers use iteration.

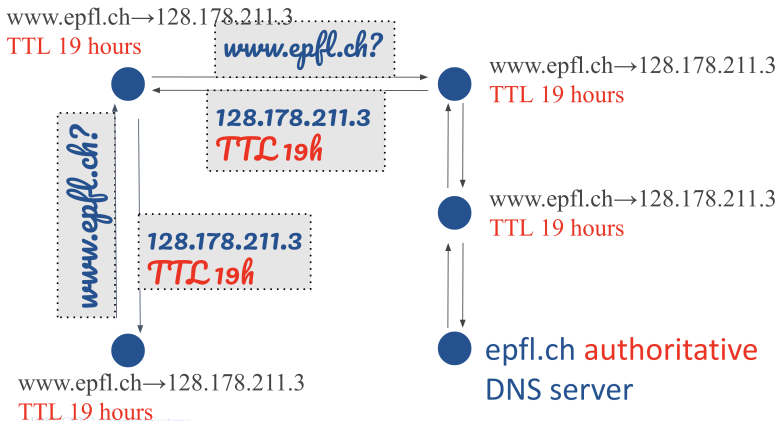
Benefits of the Hierarchy

This hierarchical structure significantly improves DNS scalability:

- No DNS server needs to know all domain names worldwide.
- Each server only needs to know where to find the next more specific server.

DNS Caching

DNS queries can involve multiple servers, which may introduce noticeable delays. To improve performance, DNS extensively uses **caching**.
Caching allows both DNS clients and servers to store recent DNS query results, so repeated requests for the same domain name can be answered much faster without contacting other servers.
The primary challenge of caching is ensuring the data remains up-to-date. Each DNS response includes a **Time To Live (TTL)**, which specifies the maximum time the result should be cached. When the authoritative server sends a response, it sets an initial TTL. As the response is passed along the DNS hierarchy, each server reduces the TTL by the elapsed time before forwarding or serving the cached result.



Example:

- The authoritative server for `epfl.ch` responds with a TTL of 24 hours.
- After 4 hours, a TLD server responds with a TTL of 20 hours.
- After another hour, a root server responds with a TTL of 19 hours, and so on.
- Each caching server and client will use the remaining TTL value.

DNS Processes

DNS operates through a combination of queries and responses, with the help of a distributed hierarchy:

- The **DNS client** generates queries to resolve domain names.
- The **local DNS server** (also called a resolver) processes client requests and can respond directly from cache or contact higher-level servers.
- The **hierarchy of DNS servers** assists in generating accurate responses for the client.

DNS Hierarchy: Summary

The DNS hierarchy consists of three main layers:

- **Root servers** (top): Know how to reach all TLD servers.
- **Top-level domain (TLD) servers** (middle): Responsible for specific TLDs (e.g., `.ch`, `.com`), know authoritative servers for each second-level domain.
- **Authoritative servers** (bottom): Responsible for a specific domain (e.g., `epfl.ch`), and know all hostnames within that domain.

DNS Protocol

DNS communication is based on a simple query-response protocol:

- A **query** (or request) is sent from a client or server to another DNS server to obtain information about a domain name.
- A **response** contains the answer, provided as a list of *resource records* (RRs).

A **resource record (RR)** is a data structure containing specific DNS information. There are several common types:

- **A record:** Maps a DNS name to an IPv4 address.
- **AAAA record:** Maps a DNS name to an IPv6 address.
- **CNAME record:** Maps a DNS name to another DNS name (alias).
- **MX record:** Specifies the mail server for a domain.
- **SOA record:** Indicates the Start of Authority for a domain.

Prof. Note: You do not need to memorize all types, but it is useful to know the main examples and their purpose.

Chapter 12

L12 – Network System Calls & the Internet

12.1 From End–Systems to Processes

Every computer attached to the Internet that exchanges messages is called an **end-system**.

- **Client process**: issues *requests*.
- **Server process**: generates *responses*.

Thus, the words *client* and *server* are used both for the *processes* and for the *machines* executing them.

12.1.1 Naming a Process

A process is uniquely identified by the pair

`IP address` : `port number`

IP address (e.g. `8.8.8.8`) – identifies the end-system.

Port number (e.g. `53`) – identifies the process on that end-system. Certain services have *well-known ports* (DNS uses port 53).

Some definitions

End-system A host connected to the Internet that can send/receive packets.

Socket A file descriptor returned by `socket()`; used with `sendto()` and `recvfrom()` instead of `read()` and `write()`.

Well-known port A globally agreed port number reserved for a specific application-level protocol (e.g. port 53 for DNS).

12.1.2 Network System Calls

The following POSIX calls allow a user process to exchange messages with a remote peer:

Server-side calls

1. `socket()` – create a network endpoint and obtain a **socket**, a special file descriptor.
2. `bind()` – register the local process name `[IP, port]` with the kernel (e.g. `[8.8.8.8,53]`).
3. `recvfrom()` – retrieve an arriving request; the kernel fills in the client's `[IP, port]`.
4. `sendto()` – transmit a response back to the requesting client.
5. `close()` – release the socket when no more requests should be served.

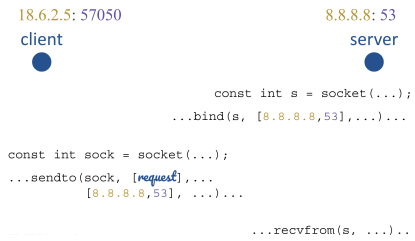
Client-side calls

1. `socket()` – create a socket (no fixed port required).
2. `sendto()` – hand the kernel the request packet and the server's address (e.g. `[8.8.8.8,53]`).
3. `recvfrom()` – wait for (or poll for) the matching response.
4. `close()` – terminate communication once all responses are received.

Blocking vs. Non-blocking Each call (most notably `recvfrom()`) may be invoked in **blocking** mode (the kernel puts the process to sleep until data arrives) or **non-blocking** mode (the call returns immediately with an error code such as `EAGAIN` if no data is ready).

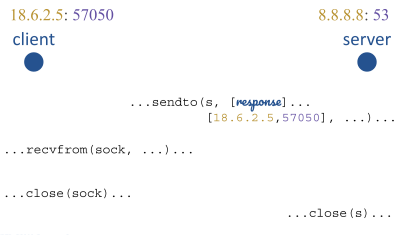
12.1.3 Example Network Flow

Client → Server (Request)



1. Client issues `socket()` then `sendto()`(*request*, [8.8.8.8,53]).
2. Kernel consults its routing tables and places the UDP datagram on the wire.
3. Server's NIC receives the packet; the kernel matches `[8.8.8.8,53]` to the waiting socket produced by `bind()`.
4. Server process unblocks from `recvfrom()` and obtains both the request data and the client's return address.

Server → Client (Response)



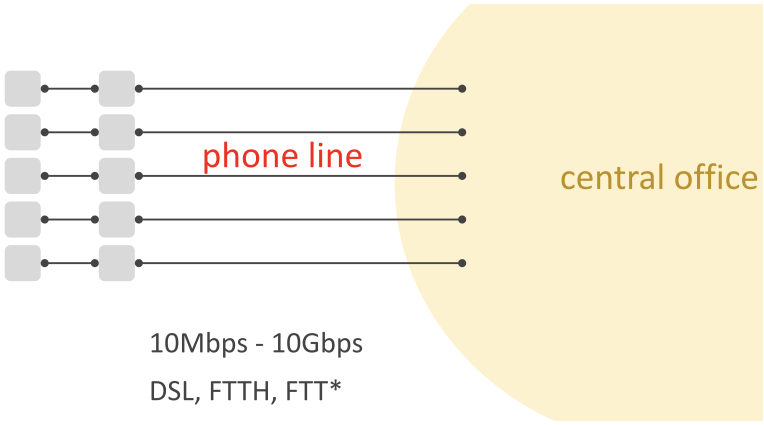
1. Server calls `sendto()`(*response*, [18.6.2.5,57050]).
2. Packet traverses the network back to the client.
3. Client's kernel delivers the datagram to the waiting socket; `recvfrom()` returns the response.
4. Client invokes `close()`. (Whether any data is sent upon `close()` depends on the socket type—topic for a later lecture.)
5. Server eventually calls `close()` when it no longer wishes to serve further requests.

12.2 Internet Components

This section introduces the most common ways an **end-system** (e.g. your laptop) reaches the global Internet. For every access technology we list the physical path, the achievable data rate, and the main engineering trade-offs.

12.2.1 Access via the Public Switched Telephone Network

A typical home setup places the laptop behind a *DSL/Fiber modem–router* that terminates the household telephone line and relays traffic to the *central office* a few kilometres away.



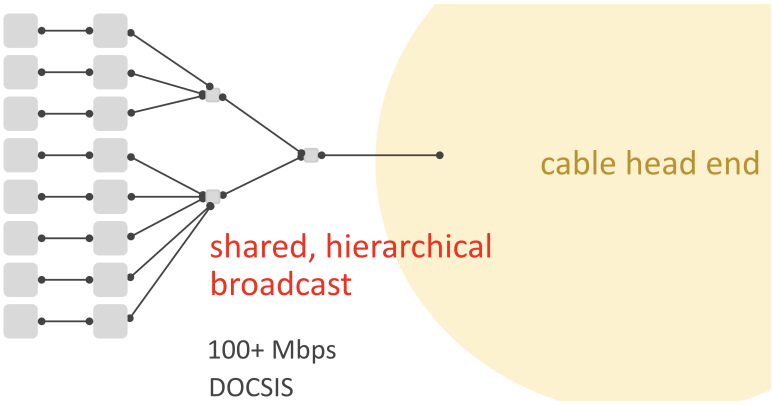
1. **Path**: Laptop → Modem/Router → Copper / Fiber local loop → Central Office → Internet.
2. **Rate**: 10–10,000 Mbps (material and DSL/Fiber technology dependent).
3. **Why legacy copper?**
   - Incumbent telcos *already own* a wire to every household — a compelling business advantage.
   - Pulling new fibre is capital-intensive; upgrades therefore proceed gradually (*FTTH*, *FTTB*, *FTTC*, ...).

End-system Any device exchanging packets over the Internet.

Modem A specialised *packet switch* that adapts IP packets to the physical characteristics of the telephone line.

12.2.2 Access via the Cable TV Network

A second option reuses the coaxial cable that once delivered only television signals.



1. **Path**: Laptop → Cable Modem/Router → Coaxial tree → *Cable Head-End* → Internet.

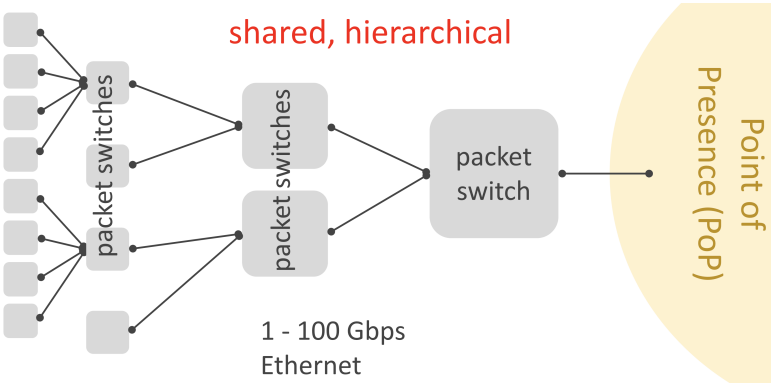
2. **Rate**: Typically 100–1,000 Mbps.
3. **Key differences to DSL**:
   (a) *Shared medium* — the last-mile coax is shared by all subscribers on the tree; throughput per user can drop at peak times (e.g. Saturday night).
   (b) *Broadcast medium* — frames sent by the head-end are physically delivered to *every* household on the tree.

Shared medium Multiple end-systems contend for the same link.

Broadcast medium A single transmission is received by all nodes on the medium.

12.2.3 Access via a Point of Presence (PoP)

Enterprise campuses and data-centres aggregate thousands of machines through a switched hierarchy that ultimately connects to an **ISP Point of Presence (PoP)**.



1. **Path**: Workstation → Access switch → Aggregation/Core switch → PoP → Internet.
2. **Rate**: 1–100 Gbps per link inside the site.
3. **Hierarchy**: Many small switches feed fewer, higher-capacity switches, reducing cost while maintaining performance.

PoP The interface between a customer network (campus, enterprise, data-centre) and an Internet Service Provider.

Summary

- **Access technologies**: DSL/Fiber over phone lines, Cable-TV coax, cellular, satellite, campus PoP, ...
- **Ownership**: Each path segment is operated by an *Internet Service Provider (ISP)* such as a telco (Swisscom), a cable operator (UPC/Sunrise), or a research network (SWITCH).
- **Design heuristic**: When extending connectivity, always ask:
  1. What infrastructure already exists?
  2. Which media are users accustomed to paying for?

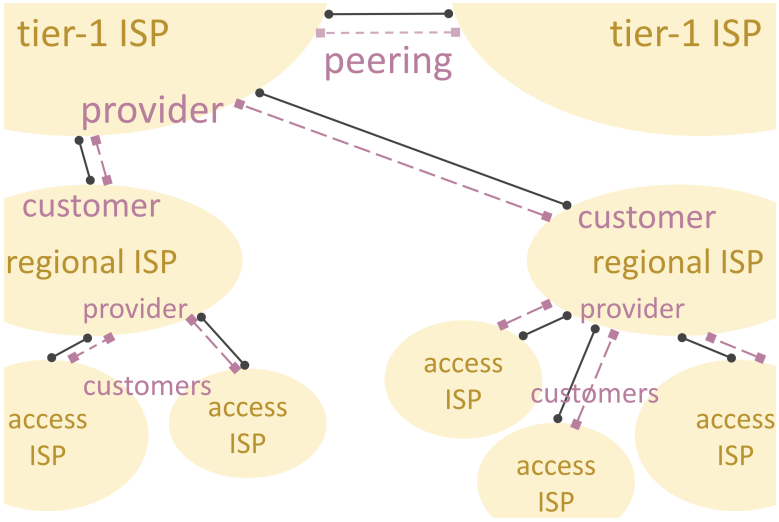
  Leveraging existing assets often yields the most economical solution.

12.3 Internet Service Providers (ISPs)

Every end-system ultimately reaches the global Internet through one or more **Internet Service Providers (ISPs)**. To scale worldwide connectivity, ISPs organise themselves in a *three-tier hierarchy* linked by specific business agreements.

12.3.1 Why a Hierarchy?

Directly wiring every pair of the world's access networks is infeasible (thousands of telcos, cable operators, universities, enterprises, ...). Instead, ISPs form a multi-level structure:



1. **Access ISPs** — interface with end-systems (e.g. Swisscom, Sunrise).
2. **Regional ISPs** — aggregate many access ISPs within a country or region.
3. **Tier-1 ISPs** — few, globe-spanning backbones that can reach every network without paying another provider.

Definition. ***Customer–provider relationship****: the lower-tier ISP (*customer*) pays the higher-tier ISP (*provider*) for global reachability.*

12.3.2 Peering Agreements

When two ISPs exchange large, roughly balanced traffic volumes, they may bypass providers and interconnect as **peers**:

- *Settlement-free peering*: each party carries the other's traffic at no cost.
- *Paid peering*: one ISP compensates the other if traffic is highly asymmetric.

Peering can occur at *any* tier (access ↔ access, regional ↔ regional, tier-1 ↔ tier-1) whenever it is economically attractive.

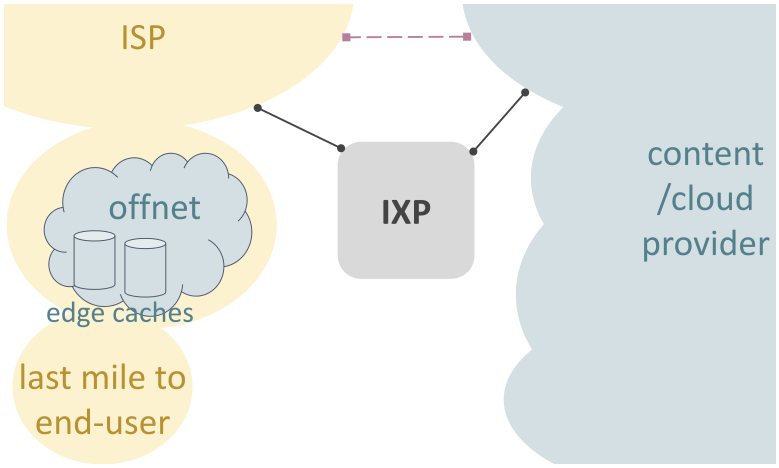
12.3.3 Internet eXchange Points (IXPs)

ISPs rarely drag a private cable to every partner. Instead, they each attach *one* high-speed link to a neutral facility called an **IXP**—essentially a massive Ethernet switch where multiple ISPs exchange traffic.

IXP Neutral switching fabric that enables many bilateral or multilateral interconnections through a single physical port.

12.3.4 Content/Cloud Providers as Networks

Companies such as Google, Microsoft, Amazon, and Meta evolved from regular customers of ISPs into operators of near-global backbones:



1. Directly connect their data-centres to large ISPs and IXPs.
2. Peer with ISPs to minimise transit cost and latency.
3. Deploy **edge caches** *inside* ISP networks to store popular content close to users.
4. Interconnect those caches via an **off-net**—a private overlay that is *logically* theirs but *physically* located within partner ISPs.

12.3.5 Edge Caches and Off-nets

Definition. ***Edge cache****: server cluster owned and managed by the content provider but* installed within *the ISP's premises; stores the most popular objects for that ISP's subscribers.*

Definition. ***Off-net****: the private backbone interconnecting multiple edge caches that reside* outside *the content provider's core network.*
This arrangement improves user experience while reducing upstream traffic for the ISP.

Recap

1. The Internet scales through a three-tier ISP hierarchy (access–regional–tier-1) governed by customer–provider contracts.
2. Peering offers a cost-effective shortcut whenever traffic volumes justify it.
3. IXPs simplify physical connectivity by acting as neutral switching hubs.
4. Large content/cloud providers now run quasi-global networks, peer with ISPs, and deploy edge caches/off-nets for performance.

12.4 The Network Interface and the CPU

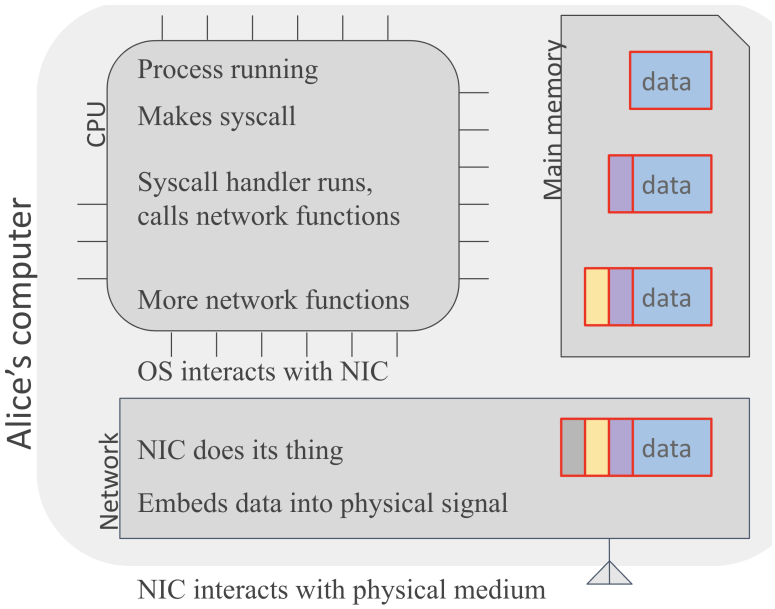
Every Internet message created by a user process must travel from *main memory* to the *Network Interface Card (NIC)*, descend through the five protocol layers, and finally leave the machine as a physical signal. This section shows how the operating system, the NIC, and the layered "network stack" cooperate to make that happen.

12.4.1 Hardware Overview

NIC controller On-board processor that handles packet queues and offloads work from the CPU.

I/O controller CPU's gateway to peripheral devices.

DMA controller Copies data between main memory and device memory *without* CPU intervention.



Send path (user process → wire):

1. User process issues a `sendto()` *network syscall*; CPU traps into *kernel mode*.
2. Kernel prepends *transport* and *network* metadata to the user data.
3. Kernel programs the I/O controller; DMA copies the buffer to NIC memory.
4. NIC adds a *link-layer* header, forming a **packet** = *payload* + *all headers*.
5. NIC converts the packet bits into an electrical/optical/radio signal and transmits it onto the medium.

12.4.2 The Five-Layer Internet Stack

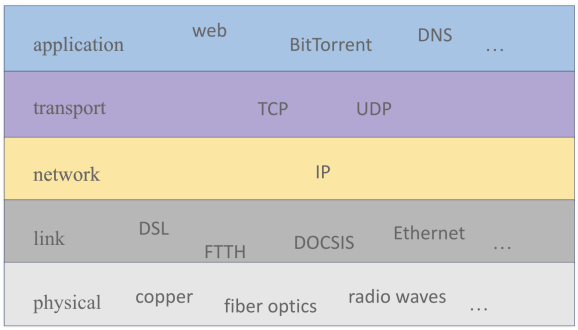
Application User programs (web, chat, file-transfer).

Transport TCP and UDP.

Network IP (Internet Protocol).

Link Ethernet, Wi-Fi, DOCSIS, DSL, ...

Physical Copper wire, fibre optics, radio, free-space optics.

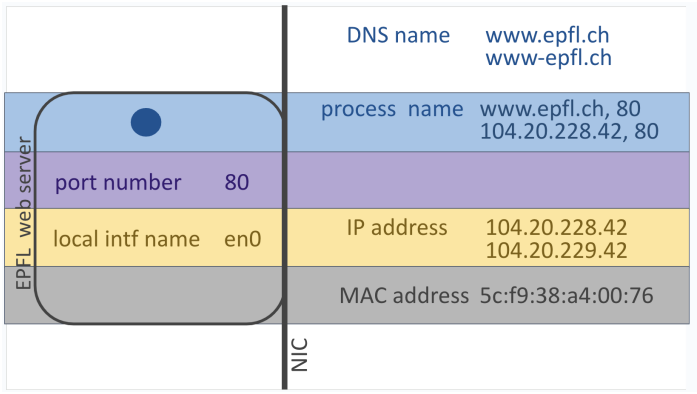


Layer Properties

- **Abstraction**: each layer hides lower-level details from the layer above.
- **Encapsulation**: a layer *prepends* its own header.
- **Decapsulation**: a layer *removes* its header on the receiving host.
- **Interfaces**: adjacent layers communicate through a well-defined API (e.g. a *syscall* between Application and Transport).

Layers reduce complexity and increase flexibility; they do not directly improve raw performance.

Example 12.4.2.1 (EPFL Web Server)**.** *Consider the machine that answers HTTP requests for* `www.epfl.ch`*. Important naming facts:*



1. ***Application layer*** *- Runs a web–server* process.
2. ***Transport layer***
   - *Identifies that process with* ***port*** `80`*.*
   - *Other processes on the same host listen on* different *port numbers.*
3. ***Network layer***
   - *Sees the host's network interface as* `en0`*.*
   - *Associates one or more* ***IP addresses*** *with* `en0` *(e.g.* `104.20.228.42`*).*
4. ***Link layer***
   - *Uses a globally–unique* ***MAC address*** *(e.g.* `5c:f9:38:a4:00:76`*) for the same interface.*

Name Identifiers

- **Interface identifiers**

  DNS name Human-readable alias (`www.epfl.ch`).
  IP address Network-layer locator (`104.20.228.42`).
  MAC address Link-layer locator (`5c:f9:38:a4:00:76`).
  OS handle Local label (e.g. `en0`).

- **Process identifier** *interface–name , port*
  A two-tuple: first choose the interface by DNS/IP, then the process by its port number.

Chapter 13

L13 — Internet Performance

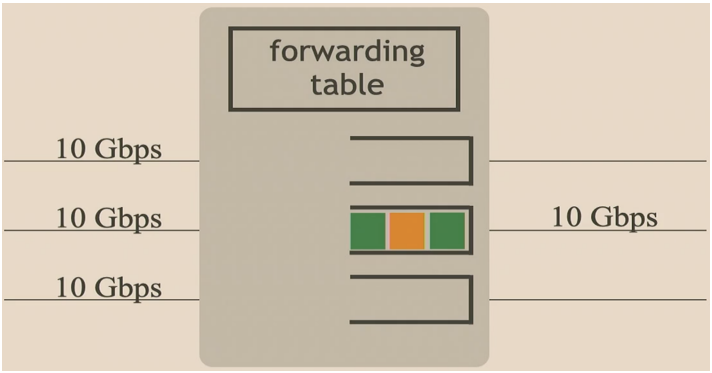
13.1 Properties of a Network Link

Every physical link on the Internet is fully described by three fundamental parameters.

1. **Transmission rate** (R) — the maximum bit rate that can be carried while maintaining an acceptable error probability, measured in bit per second (e.g. 1 Gbps, 10 Gbps, 100 Gbps).
2. **Length** (L) — the physical distance between the two end-points of the link, measured in metres.
3. **Propagation speed** (v) — the speed at which a signal travels through the medium, measured in metre per second ($\approx$ *the speed of light in fibre*).

13.1.1 Packet Switches

Packet switches interconnect links and move packets from their source toward their destination by consulting forwarding rules.



A switch attaches to multiple (usually bidirectional) links. Packets arriving on any link are placed in a queue associated with the chosen outgoing link and are transmitted at that link's rate R (thus the queue drains at R whenever it is non-empty).

Contents of a Packet Switch

Inside a modern switch we find two essential data structures.

1. **Queues** — buffers that temporarily *store* packets awaiting transmission.
2. **Forwarding table** — metadata that maps header fields (e.g. destination address) to the appropriate outgoing link.

Store-and-Forward Packet Switching

Most Internet switches use the store-and-forward paradigm.

1. The switch waits until *all* bits of a packet arrive.
2. It extracts the packet header and consults the forwarding table.
3. It selects the best outgoing link and enqueues the packet.
4. The packet is transmitted at the link's rate R.

Cut-Through Switching (for comparison). Some high-performance switches begin forwarding as soon as the header has been received. These are called **cut-through** switches; unless stated otherwise, assume switches in this course are store-and-forward.

13.1.2 Network Congestion

It is possible that a queue inside a packet switch fills up faster than it can drain.
This can happen, for example, if we have a scenario where 3×10 Gbps links are all feeding traffic to a single 10 Gbps link. In this scenario, packets may arrive at a queue faster than they depart, which means that some packets may have to wait inside the queue and/or some packets may be dropped because they arrive at a moment when the queue is full.
This is an example of **network congestion**, which results in:

- **Packet loss** — packets are dropped inside the network
- **Queuing delay** — packets have to wait in a queue

13.2 Network Performance Analysis

How do we reason about network performance, and what kind of performance does the Internet provide?

13.2.1 Basic Network Performance Metrics

To quantify the performance of the network between a source and a destination, we use three simple metrics.

1. **Packet loss** — the fraction of packets from source to destination that are lost on the way
   - Measured in percentage, e.g., 1% packet loss
2. **Packet delay** — the time it takes for a packet to get from source to destination
   - Measured in time units, e.g., 10 msec
3. **Average throughput** — the average rate at which the destination receives data
   - Measured in bits per second (bps)
   - Example: destination receives 1 GB of data in 1 min; average throughput $= \frac{8\times10^9 \text{ bits}}{60 \text{ sec}} = 133.34 \times 10^6$ bps $= 133.34$ Mbps

13.2.2 Understanding Delay vs. Throughput

To better understand the difference between delay and throughput, consider the following analogy. Visualize a path that is narrow and long: it fits only one person, and once a person gets on it, it takes 1 hour to traverse it. Suppose that there are multiple such paths between the same start and end points.

Scenario 1: Small Group (3 Friends)

Consider a group of 3 friends who want to go from start to end:

- **Option 1:** They all get onto the same path, one after the other
- **Option 2:** Each friend gets on a different path and they traverse in parallel

Whether they pick option 1 or option 2, it takes about 1 hour for all of them to traverse. The choice of using parallel paths does not make a significant difference.

Scenario 2: Large Group (1,000,000 People)

Now consider a group of 1,000,000 people:

- **Option 1:** They all get on the same path, one after the other
- **Option 2:** They use multiple paths in parallel

By using N parallel paths, they reduce the time it takes for all of them to traverse by about a factor of N.

Key Insight. The difference between the two scenarios is the amount of time that a person has to wait *before* they get on the path:

- In the 3-people scenario, the 2nd and 3rd person wait only a few extra seconds
- In the 1,000,000-people scenario, many people have to wait significantly longer

What Metric Do We Improve? Using parallel paths:

- Does **not** improve the delay for one person to go from start to end
- **Does** improve the throughput: the rate at which people reach the end

Application to Computer Networks

- **Packet delay** matters for exchanging small messages fast (e.g., interactive applications like voice or gaming)
- **Average throughput** matters for bulk transfers (e.g., downloading large files)
- They are related to each other, but not in an obvious way

13.3 Packet Delay Components

Estimating packet delay between various end-systems under different scenarios is one of the main activities of network engineers.

13.3.1 Direct Connection Scenario

Two end-systems are directly connected through a single link.
When the source transmits a packet over the link, there are two delay components:

Transmission Delay

The time to push all bits of the packet onto the link:

$$\text{Transmission delay} = \frac{\text{packet size}}{\text{link transmission rate}}$$

Example: For a 3-bit packet on a 1 Gbps link:

$$\text{Transmission delay} = \frac{3 \text{ bits}}{1 \times 10^9 \text{ bps}} = 3 \text{ nsec}$$

Propagation Delay

The time for the last bit to reach the destination:

$$\text{Propagation delay} = \frac{\text{link length}}{\text{link propagation speed}}$$

Example: For a 1-meter link with speed of light propagation:

$$\text{Propagation delay} = \frac{1 \text{ meter}}{3 \times 10^8 \text{ m/s}} = 3.34 \text{ nsec}$$

Total Packet Delay.

$$\text{Total packet delay} = \text{transmission delay} + \text{propagation delay}$$

13.3.2 Store-and-Forward Switch Scenario

Two end-systems connected through a store-and-forward packet switch.
In this scenario, the packet delay includes additional components:

$$\begin{aligned}\text{Packet delay} &= \text{transmission delay over 1st link} && (13.1)\\ &+ \text{propagation delay of 1st link} && (13.2)\\ &+ \text{queuing delay at switch} && (13.3)\\ &+ \text{processing delay at switch} && (13.4)\\ &+ \text{transmission delay over 2nd link} && (13.5)\\ &+ \text{propagation delay of 2nd link} && (13.6)\end{aligned}$$

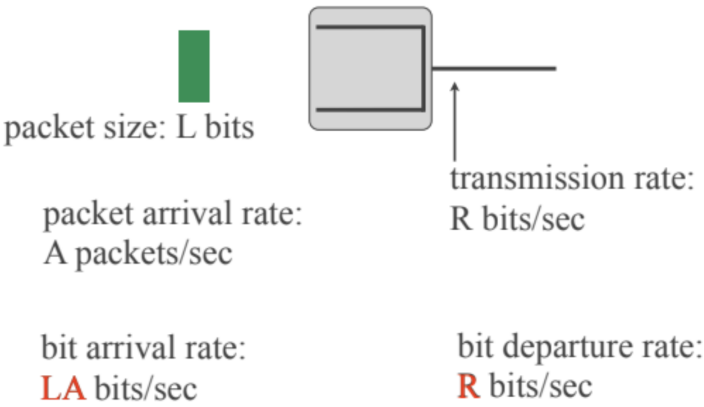
13.3.3 Queuing Delay

The most variable component of packet delay.
Unlike transmission and propagation delays, queuing delay cannot be precisely calculated because it depends on other traffic arriving at the switch. However, if we have information about the other traffic (e.g., its arrival rate at the queue and how bursty it is), it is possible to compute statistical measures of the queuing delay.

Queuing Delay Analysis

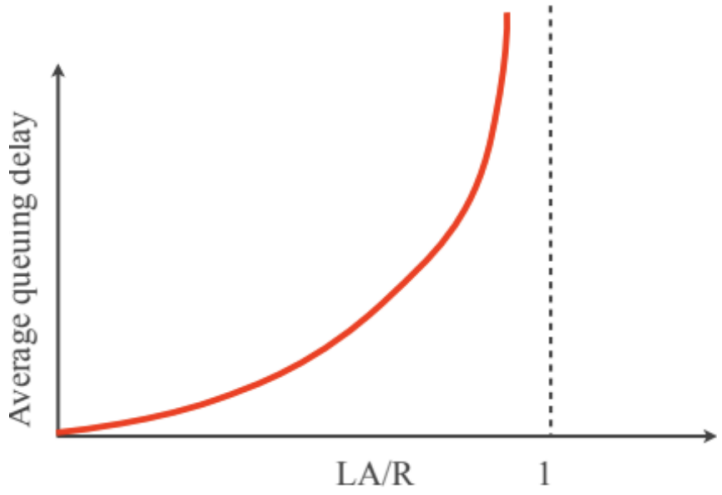
With an infinite queue assumption, queuing delay characteristics.

- Approaches infinity if arrival rate > departure rate
- Depends on burst size when arrival rate ≤ departure rate



Consider a packet switch, with packets arriving and departing at an outgoing link at rate R. Suppose all packets have size L bits and arrive at rate A packets/sec (i.e., LA bits/sec). Assuming an infinite buffer at the switch, we compare the bit arrival rate LA to the bit departure rate R to reason about queuing delay.

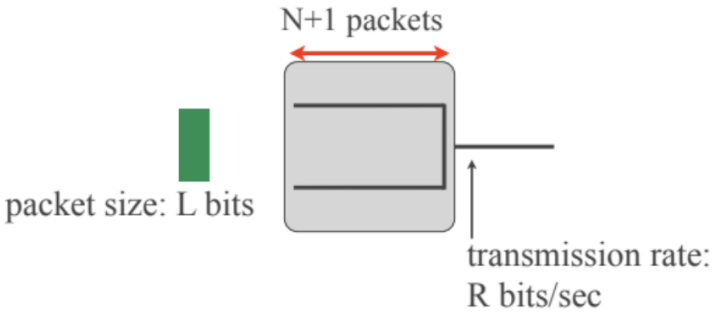
- **Scenario 1:** $LA > R$. Queuing delay grows without bound as more packets accumulate.
- **Scenario 2:** $LA \leq R$. Queuing delay depends on the burstiness of arrivals; instantaneous bursts can cause non-zero delay even if the average rate does not exceed R.



This plot shows the average queuing delay experienced by a packet arriving at a queue, as a function of the arrival rate divided by the departure rate. As LA/R approaches 1 (the arrival rate approaches the departure rate), the average queuing delay goes to infinity. The important characteristic of this curve is its exponential shape: there exists a threshold on the x-axis beyond which the curve increases rapidly. When a queue operates beyond this point, small changes in arrival rate can have a dramatic impact on queuing delay. In system design, queues should operate well below this threshold.

Finite Queue Capacity

Real switches have limited buffer capacity.



Queuing delay upper bound: N L/R

In reality, switches don't have infinite queues. When a packet arrives and the queue is full, the switch drops the packet. The capacity of the queue imposes an upper bound on the queuing delay that a packet can suffer.

Maximum Queuing Delay. If the queue can fit $N + 1$ packets, what is the maximum queuing delay that a packet may experience? Assuming that processing delay is insignificant, the maximum queuing delay is N times the transmission delay (L/R). This occurs when a packet must wait for N other packets to be transmitted over the outgoing link.

13.4 File Transfer Analysis

Understanding delay and throughput for bulk data transfer.
Packet delay between two end-systems has many components: transmission, propagation, queuing, and processing delays. The relative importance of each component depends on the network topology, link properties (transmission rates and propagation delays), switch operation, queue capacity, and traffic patterns.

13.4.1 Direct Link File Transfer

Two end-systems directly connected through a single link.
Consider two end-systems connected by a physical link with rate R bps. Previously, we computed the delay for one packet of size L bits. Now we compute the delay for a file of size F bits, assuming the source cuts the file into multiple packets of size L bits and sends them consecutively.

Transfer Time Analysis. The file transfer time consists of:

1. The source pushes all F bits onto the link: F/R
2. The last bit propagates to the destination: propagation delay

Total transfer time:

$$\text{Transfer time} = \frac{F}{R} + \text{propagation delay}$$

In practice, F/R is typically large enough that the propagation delay becomes insignificant and can be ignored.

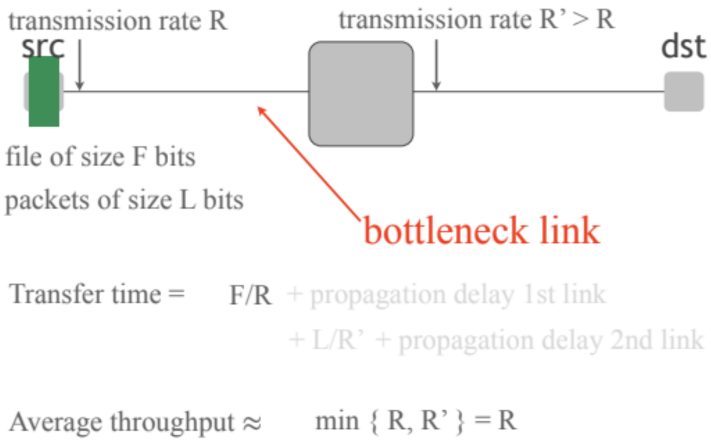
Average Throughput. The source sent F bits in approximately F/R time units:

$$\text{Average throughput} \approx \frac{F}{F/R} = R$$

The throughput is always slightly smaller than R due to the propagation delay. When two end-systems communicate over a link of transmission rate R, their throughput can never exceed R.

13.4.2 Store-and-Forward with Multiple Links

File transfer through packet switches.



Consider the same scenario with a store-and-forward packet switch between the end-systems. Assume the switch introduces negligible processing delay and receives no other traffic. Let the second link have transmission rate $R' > R$.

Transfer Time Analysis. The transfer consists of:

1. Source pushes all F bits onto the first link: F/R
2. Propagation delay of first link
3. Last packet transmission on second link: L/R'
4. Propagation delay of second link

Since $R' > R$, packets leave the switch faster than they arrive, so the switch doesn't become a bottleneck. The total time is approximately F/R.

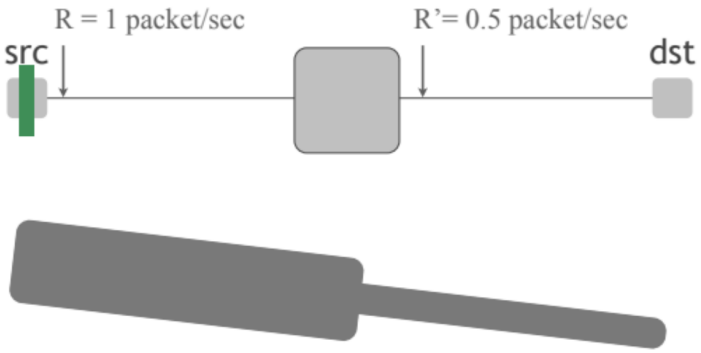
Bottleneck Link Concept. When multiple switches are added between source and destination, the average throughput equals the transmission rate of the slowest link. This slowest link is called the **bottleneck link**, as it determines the maximum rate at which traffic can flow between the end-systems.

13.4.3 Bottleneck Link Examples

Concrete examples illustrating bottleneck behavior.

Example 1: Second Link as Bottleneck

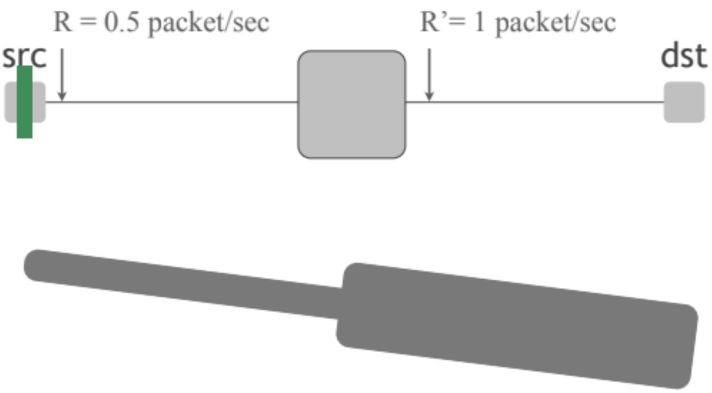
Consider a scenario where the first link has twice the transmission rate of the second link. If the first link supports 1 packet/sec for a given packet size, the second link supports 0.5 packet/sec.



When the source transmits at 1 packet/sec, packets arrive at the switch faster than they can be transmitted on the second link. The switch must space them out, transmitting at only 0.5 packet/sec on the second link. This is analogous to a bottle where the narrow neck (second link) determines the flow rate, regardless of the bottle's capacity.

Example 2: First Link as Bottleneck

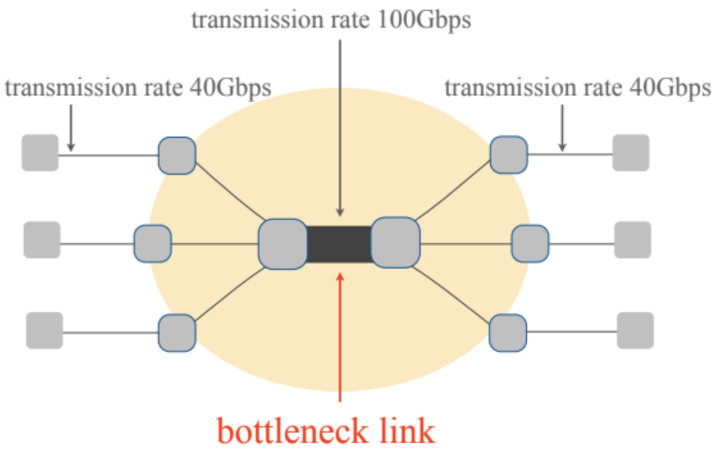
Now consider the reverse scenario where the second link has twice the transmission rate of the first link. If the source transmits at 0.5 packet/sec, the switch can immediately forward packets on the faster second link without spacing them out. However, it cannot bring packets closer together; it maintains the existing spacing determined by the first link.



This resembles an inverted bottle where liquid enters through the narrow neck and exits through the wide opening. The bottleneck still determines the overall flow rate.

Bottleneck Location

When two end-systems communicate over the Internet, the bottleneck link is often at the edge of their communication path, near one of the end-systems. However, the bottleneck link is not always the one with the smallest transmission rate.



Example: Consider a scenario where edge links have 40 Gbps transmission rates, while a middle link has 100 Gbps capacity. If the middle link receives aggregate traffic exceeding 100 Gbps from multiple sources, it becomes congested. The resulting queuing delay makes this high-capacity link the bottleneck, despite its superior transmission rate.

13.5 Bottleneck Link Summary

Key characteristics of bottleneck links.
The bottleneck link between communicating end-systems is defined as the link where traffic flows at the slowest rate. This can occur due to two primary factors:

- **Low transmission rate** — the link has insufficient capacity
- **Queuing delay** — the link becomes congested due to excessive traffic load

Understanding bottleneck identification and mitigation is crucial for network performance optimization and capacity planning.

13.6 Resource Management in Packet Switches

Understanding different approaches to managing network resources.
To understand Internet performance characteristics, we must examine how packet switches manage their resources. There are two fundamental approaches to resource management in packet-switched networks.

13.6.1 Packet Switching

Resources allocated on-demand per packet.
In packet switching, when a packet arrives at a switch, the switch decides whether it has sufficient resources (e.g., queue space) to store and process the packet. If resources are available, the switch uses its forwarding table to select the appropriate outgoing link and places the packet in the corresponding queue. Otherwise, the switch drops the packet.
Key characteristics:

- Each packet is treated as an independent entity
- Resources are allocated on-demand
- Admission and forwarding decisions are made per packet

13.6.2 Circuit Switching

Resources reserved in advance for virtual circuits.
Circuit switching (specifically virtual circuit switching) creates the illusion of dedicated physical circuits without requiring actual dedicated links. When a source wants to communicate with a destination, it contacts all switches on the path and requests: "This source wants to communicate with this destination, sending at a maximum rate of X Mbps, over the next Y minutes. Can you accommodate this?"

Virtual Circuit Establishment

If a switch accepts the request, it:

1. Chooses the best outgoing link for the destination
2. Reserves the necessary capacity (X Mbps) on that link
3. Allocates a dedicated queue for the source/destination pair
4. Commits to serving this queue at the requested rate

If all switches on the path accept the request, a **virtual circuit** is established. As long as the source doesn't exceed X Mbps, its packets experience no packet loss or unpredictable queuing delay.
Key characteristics:

- Traffic from each source/destination pair is treated as one entity
- Resources are reserved in advance
- Admission and forwarding decisions are made per virtual circuit

13.6.3 Performance Comparison

Analyzing the trade-offs between the two approaches.

Circuit Switching Analysis

Advantages: Predictable Performance. Consider a switch participating in two virtual circuits, both using the same 1 Gbps outgoing link. Each virtual circuit reserves 500 Mbps. As long as each circuit sends no more than its reserved rate, there is no packet loss or unpredictable queuing delay.

Disadvantages: Resource Inefficiency. If one virtual circuit (e.g., orange) becomes silent while another (e.g., green) wants to send 1 Gbps, the switch cannot accommodate the green circuit's extra traffic. Reserved resources for the silent orange circuit remain unused, similar to a restaurant table reservation where the customer doesn't show up. This inefficiency occurs when the system turns down service requests despite having idle resources.

Packet Switching Analysis

Advantages: Resource Efficiency. Without reservations, sources send packets into common queues per outgoing link. Whether two sources each send 500 Mbps simultaneously, or one source sends 1 Gbps while the other is silent, the system accommodates all traffic up to the link capacity. Packets are never dropped if the switch has available resources.

Disadvantages: Unpredictable Performance. Consider a scenario where one source consumes all available switch resources at 1 Gbps. When a second source attempts to send packets, they will likely be dropped due to resource exhaustion. Unlike circuit switching, packet switching provides no performance guarantees.

13.6.4 Implementation Complexity

Comparing the complexity of each approach.
Packet switching is simpler to implement, requiring no special mechanisms at switches. Circuit switching requires more sophisticated switches that can:

- Evaluate and accept/reject reservation requests
- Determine appropriate resource allocations per request
- Perform actual resource reservation and management

However, packet switching introduces its own complexity: the need for **congestion control**. Without congestion control, aggressive sources can monopolize switch resources, preventing other sources from sending packets effectively.

13.6.5 Resource Management Summary

Comparing the fundamental trade-offs.

- **Packet switching:** Efficient resource use, no performance guarantees, simpler to implement but requires congestion control
- **Circuit switching:** Performance guarantees, inefficient resource use, more complex to implement

The Internet uses packet switching, which is why it offers a **"best effort" service**. There is no guarantee that packets will be delivered. When traffic traverses multiple packet switches, each switch does its best to store and process packets without typically reserving resources in advance.

13.7 Statistical Multiplexing

Efficient resource sharing based on statistical behavior of users.
Statistical multiplexing occurs when many users share the same resource, but not all users are expected to be active simultaneously. This principle enables packet switching to support significantly more users than circuit switching.

13.7.1 Video Server Example

Demonstrating the benefits of statistical multiplexing.
Consider a video server connected to a switch via a 10 Gbps link. Video clients connect to the switch and download videos, requiring at least 1 Gbps each for reasonable performance. Each client downloads only 10% of the time (the remainder is spent watching the downloaded content).

Circuit Switching Capacity

With circuit switching, the switch must reserve 1 Gbps for each client. Given the 10 Gbps link capacity, the switch can serve at most **10 clients**.

Packet Switching Capacity

With packet switching and the 10% activity rate, the switch can serve approximately **35 clients**. The probability that 10 or fewer clients are downloading simultaneously is 99.96%, providing nearly the same performance as circuit switching while serving more than three times as many clients.

13.7.2 Resource Efficiency Example

Illustrating dynamic resource allocation.
Consider 10 potential clients where only one is active, downloading a 10 Gb video file:

- **Circuit switching:** With 1 Gbps reserved per client, the download takes 10 seconds
- **Packet switching:** The active client can use the full 10 Gbps capacity, completing the download in 1 second

This demonstrates efficient resource utilization: inactive clients don't consume resources, allowing active clients to achieve better performance.

13.7.3 Historical Context: Traditional Circuit Switching

Understanding the origins of circuit switching terminology.
Traditional telephone networks used actual circuit switches that created dedicated physical circuits. When a source wanted to communicate with a destination, a **circuit establishment phase** configured each circuit switch on the path to create a real physical circuit. Once established, the switches played no further role—the source could send signals directly to the destination through the dedicated physical circuit.
Modern "virtual" circuit switching simulates this behavior without requiring dedicated physical circuits, using resource reservations to provide similar guarantees.

13.8 Circuit Implementation Techniques

Different approaches to creating circuit-switched communications.
There are multiple ways to implement circuit switching, each creating the illusion of separate physical circuits for communicating pairs while sharing physical infrastructure.

13.8.1 Types of Circuits

Physical versus virtual circuit implementations.

- **Physical circuits:** Separate sequence of physical links per communicating end-system pair
- **Virtual circuits:** Manage resources as if there was a separate sequence of physical links per communicating pair

13.8.2 Multiplexing Techniques

Methods for sharing physical resources among multiple communications.

Time Division Multiplexing (TDM)

A single physical circuit is divided into time slots, with each source/destination pair assigned a separate time slot. This is analogous to a classroom used by multiple courses, where each course uses the room during a different time period. Each class has the illusion of having their own classroom.

Frequency Division Multiplexing (FDM)

A single physical circuit's bandwidth is divided into frequency bands, with each source/destination pair assigned a separate frequency band. This resembles radio broadcasting, where multiple stations share the airwaves but each uses a different frequency band.
Both techniques create the illusion of dedicated physical circuits while efficiently sharing underlying physical resources.

13.9 System Design Considerations

Choosing between on-demand and reservation-based approaches.

13.9.1 The Restaurant Analogy

Understanding the fundamental trade-offs through a familiar example.
Consider designing a restaurant's reservation policy:

No Reservations (On-Demand):

- Restaurant stays maximally utilized when customers keep arriving
- Customers may wait or leave when capacity is exceeded
- Higher resource efficiency but unpredictable service quality

Reservation System:

- Tables must be held for reserved customers who may not show up
- Guaranteed service for customers with reservations
- Lower resource efficiency but predictable service quality

13.9.2 Network Design Trade-offs

Applying the restaurant analogy to network systems.
The choice between packet switching (on-demand) and circuit switching (reservations) requires balancing:

- **Infrastructure cost and complexity** versus **quality of service guarantees**
- **Resource utilization efficiency** versus **performance predictability**
- **Implementation simplicity** versus **service reliability**

Network designers must evaluate these trade-offs based on application requirements, user expectations, and economic constraints to determine the most appropriate resource management approach.

13.10 Network Security Considerations

Understanding threats and vulnerabilities in shared networks.
When multiple users share a network, there are opportunities for misbehavior. Network security must address various types of malicious activities that can compromise communication integrity, confidentiality, and availability.

13.10.1 Common Network Threats

Categorizing different types of network attacks.

Eavesdropping (Sniffing)

Eve the eavesdropper attempts to listen in on communications between legitimate users (e.g., Alice and Bob) and copy their data. This passive attack compromises confidentiality by allowing unauthorized access to private information.

Impersonation (Spoofing)

Persa the impersonator pretends to be a legitimate user (e.g., Alice) to extract information from another user (e.g., Bob). Unlike eavesdropping, this is an active attack where the attacker generates traffic and participates in the communication.

Denial of Service (DoS)

Denis the denial-of-service attacker disrupts communication between legitimate users by consuming network or system resources. This can be accomplished by:

- Sending excessive junk traffic to exhaust the target's resources
- Coordinating attacks using multiple compromised systems (**Distributed DoS**)
- Enlisting networks of compromised computers (**botnets**)

Malware

Malik the malware master infects users' computers with malicious software designed to:

- Delete or corrupt data
- Steal sensitive information

- Force computers to send spam or participate in attacks
- Provide unauthorized remote access to systems

13.10.2 Trust Models in System Design

Defining assumptions about user behavior.
A critical design question for any computing/communication system is determining what to assume about user behavior. Trust models define the security assumptions and threat models that inform system design decisions.

13.11 Fundamental Design Questions

Key considerations for computing and communication systems.
Through our examination of Internet architecture and performance, several fundamental questions emerge that must be addressed when designing any computing or communication system:

1. **What physical infrastructure is already available?** — Understanding existing resources and constraints
2. **What layers to define?** — Determining system architecture and abstraction boundaries
3. **Treat on demand or take reservations?** — Choosing between dynamic allocation and advance reservation
4. **What trust model to design for?** — Defining security assumptions and threat models

These questions form the foundation for making informed architectural decisions that balance performance, security, complexity, and cost considerations in system design.

Chapter 14

L14 — Transport Layer and TCP

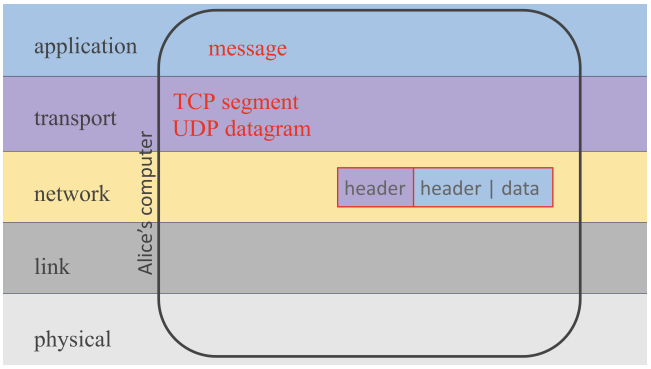
14.1 Introduction to the Transport Layer

Understanding the role and responsibilities of the transport layer in network communication.
Throughout this semester, we have focused on the application layer, examining how processes interact through system calls and how applications like web clients, servers, and DNS operate. Today, we explore what happens beneath the application layer when processes make network system calls.

14.1.1 Protocol Stack and Data Structures

Understanding how data is transformed as it moves through network layers.
The Internet uses a layered architecture where each layer adds header information:

1. **Application Layer** — Creates **messages** for inter-process communication
2. **Transport Layer** — Adds transport headers, creating:
   - **UDP datagrams** (User Datagram Protocol)
   - **TCP segments** (Transmission Control Protocol)
3. **Network Layer** — Adds IP headers, creating **IP packets**



The transport layer provides essential services enabling reliable process-to-process communication over an unreliable network infrastructure.

14.2 User Datagram Protocol (UDP)

A minimal transport protocol providing basic services with low overhead.
UDP adds minimal functionality to the network layer, making it suitable for applications requiring low overhead that can tolerate data loss.

14.2.1 UDP Services and Communication

Core functions and operation of UDP.
UDP provides three fundamental services:

Multiplexing and Demultiplexing

- **Multiplexing** — Handles messages from multiple application processes, encapsulating each in UDP datagrams
- **Demultiplexing** — Uses port numbers to deliver incoming datagrams to the correct application process
- Enables multiple applications to share network resources efficiently

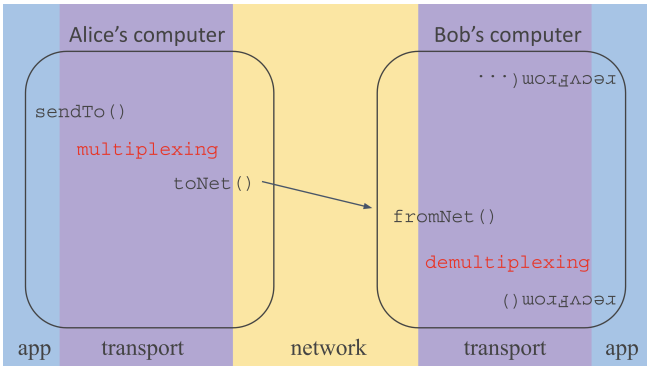
Basic Error Detection

- **Checksum computation** — Sender computes checksum from header and data
- **Integrity verification** — Receiver recomputes and compares checksums
- **Corruption handling** — Corrupted datagrams are discarded

The checksum uses one's complement arithmetic on 16-bit words, providing simple but effective error detection for most transmission errors.

14.2.2 UDP Communication Process

Step-by-step UDP communication between client and server.



1. Both processes create UDP sockets: `socket()`
2. Server binds to specific address: `bind()`
3. Server prepares to receive: `recvfrom()`
4. Client sends message: `sendto()`
5. Transport layers handle UDP datagram transmission
6. Server receives message and `recvfrom()` returns

14.2.3 UDP Header Structure and Limitations

Understanding UDP's simple structure and constraints.

UDP Header (8 bytes total)

- **Source Port (16 bits)** — Sending process identifier
- **Destination Port (16 bits)** — Receiving process identifier
- **Length (16 bits)** — Total datagram length
- **Checksum (16 bits)** — Error detection value

UDP Capabilities and Limitations

- **Provides:** Process identification, basic error detection, low overhead
- **Lacks:** Reliability guarantees, ordering, flow control, congestion control, error correction

The network layer operates on a **best-effort** basis, potentially dropping, corrupting, or reordering packets. UDP provides a minimal abstraction over this unreliable foundation.

14.3 Transmission Control Protocol (TCP)

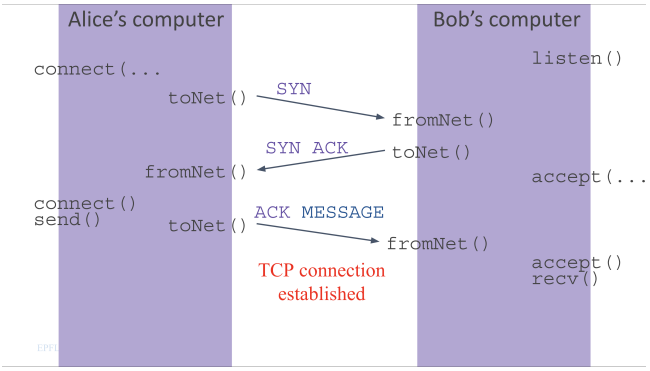
A comprehensive transport protocol providing reliable, ordered data delivery.
TCP builds upon UDP's basic services while adding connection management, reliability, flow control, and congestion control.

14.3.1 TCP Connection Management

How TCP establishes and maintains reliable communication channels.

Connection Establishment: Three-Way Handshake

Unlike UDP's connectionless approach, TCP requires establishing a logical connection before data exchange. This ensures both parties are ready to communicate and agree on communication parameters.



Handshake Concept. The connection establishment works like a conversation:

1. **Client Request:** "I want to establish a connection with you"
2. **Server Response:** "I accept your request, and I also want to establish a connection with you"
3. **Client Confirmation:** "I acknowledge your acceptance, connection established"

Technical Implementation. TCP implements this using control flags in segment headers:

1. **Server Setup:** `socket()`, `bind()`, `listen()`
2. **Client Initiation:** `socket()`, `connect()` → sends **SYN** (synchronize) segment
3. **Server Response:** Sends **SYN-ACK** (synchronize-acknowledge) segment
4. **Client Confirmation:** Sends **ACK** (acknowledge) segment (may include data)
5. **Server Acceptance:** `accept()` creates connection socket

Resource Allocation. During handshake, both sides allocate send/receive buffers and establish connection state (sequence numbers, window sizes).

14.3.2 TCP Socket Types and Multiplexing

Understanding TCP's sophisticated connection management.

Socket Type Distinction

- **Listening Socket** — Server uses for accepting new connections (`socket()`, `bind()`, `listen()`)
- **Connection Socket** — Dedicated to specific client-server communication, created by `accept()`
- **Connection Identification** — Four-tuple: (source IP, source port, destination IP, destination port)

Key Differences from UDP

- Each TCP connection socket communicates with exactly one remote process
- Servers can handle multiple simultaneous connections using separate connection sockets
- Client uses `connect()` to establish connection to specific server process
- Server uses `accept()` to create dedicated socket for each client connection

14.3.3 TCP Reliability Mechanisms

How TCP ensures reliable data delivery over unreliable networks.
TCP implements comprehensive reliability mechanisms to overcome network limitations including corruption, loss, and reordering.

Error Detection and Acknowledgment

Basic Operation:

1. Sender transmits TCP segment with data and sequence number
2. Receiver verifies integrity using checksum
3. If uncorrupted: receiver sends ACK and delivers data to application
4. If corrupted or lost: sender detects via timeout and retransmits

Sequence Numbers and Acknowledgments

Sequence Numbers:

- Identify data bytes: sequence number indicates the first data byte in segment
- Enable duplicate detection: retransmitted segments have same sequence number
- Support ordered delivery: receiver can reorder segments if needed

Cumulative Acknowledgments:

- ACK n means "I have received all bytes up to and including byte $n-1$, expecting byte n"
- No explicit negative ACKs: repeated ACKs serve as implicit NACKs
- Simplifies protocol: single ACK field confirms multiple segments

Example. If sender transmits SEQ 1 then SEQ 2, and receiver responds with ACK 2 twice, the second ACK 2 implicitly signals that SEQ 2 was not received correctly.

Timeout and Retransmission

Timeout Mechanism:

- Sender starts timer for each transmitted segment
- If ACK not received before timeout, assume segment lost and retransmit
- Handles both segment loss and ACK loss scenarios
- May cause unnecessary retransmissions due to delayed ACKs

Timeout Calculation:

$$\text{EstimatedRTT} = 0.875 \times \text{EstimatedRTT} + 0.125 \times \text{SampleRTT} \tag{14.1}$$

$$\text{Timeout} = \text{EstimatedRTT} + 4 \times \text{DevRTT} \tag{14.2}$$

Where DevRTT measures RTT variance. Conservative estimation prevents premature timeouts while adapting to network conditions.

14.3.4 TCP Header Structure

Understanding the complexity required for reliable communication.

Key TCP Header Fields (minimum 20 bytes)

- **Source/Destination Ports** — Process identification
- **Sequence Number** — First data byte number in this segment
- **Acknowledgment Number** — Next expected byte number
- **Control Flags** — SYN, ACK, FIN, RST for connection management
- **Window Size** — Flow control information
- **Checksum** — Error detection

Control Flags

- **SYN** — Connection establishment (handshake), 1-bit field
- **ACK** — Acknowledgment field is valid
- **FIN** — Connection termination
- **RST** — Immediate connection reset

14.4 Reliability Mechanisms Summary

Comprehensive overview of transport layer reliability techniques.

14.4.1 Basic Reliability Components

TCP achieves reliable data delivery through the combination of three fundamental mechanisms:

1. **Checksums** — Detect corruption in transmitted data
2. **Sequence Numbers + Acknowledgments + Retransmissions** — Overcome corruption through confirmed delivery
3. **Timeouts + Sequence Numbers + Acknowledgments + Retransmissions** — Detect and overcome loss

14.4.2 Protocol Comparison

Understanding when to use UDP vs TCP.

Protocol Trade-offs

- **UDP:** Low overhead (8-byte header), fast, simple; no reliability guarantees
- **TCP:** Higher overhead ($\geq$20-byte header), complex; provides reliability, ordering, flow control

Application Suitability

- **UDP Applications:** Real-time communication (VoIP, gaming), DNS queries, streaming media
- **TCP Applications:** File transfer (HTTP, FTP), email, remote login (SSH), e-commerce

Design Principle. The transport layer demonstrates a fundamental trade-off in protocol design: simplicity and efficiency versus feature richness and guarantees. Both approaches serve different application requirements in the Internet ecosystem.

14.5 TCP Bidirectional Communication

Understanding sequence numbers and acknowledgments in realistic scenarios.
In practice, TCP connections carry bidirectional communication where both client and server processes exchange data simultaneously. This requires careful coordination of sequence numbers and acknowledgments.

14.5.1 Bidirectional Data Exchange

How TCP handles simultaneous data transmission in both directions.

Basic Bidirectional Example

Consider a simple exchange where both client and server send single-byte messages:

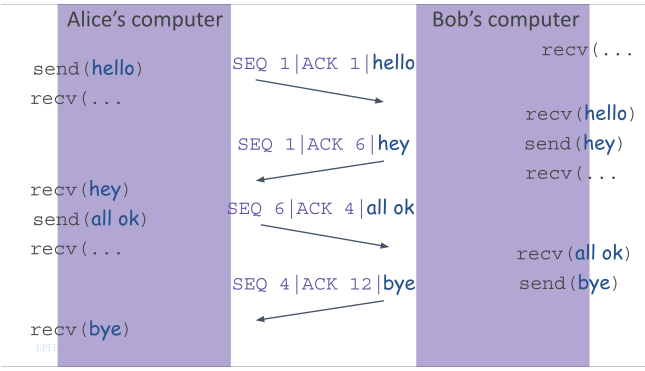
- **Client sends "A":** SEQ 1, ACK 1 (first byte to server, expecting server's first byte)
- **Server sends "B":** SEQ 1, ACK 2 (first byte to client, received client's byte 1, expecting byte 2)

Key Insight. Each TCP segment contains both data (if any) and acknowledgment information, enabling efficient bidirectional communication without separate ACK-only segments.

Multi-Byte Message Example

Realistic example with variable-length messages.
Consider an application-layer exchange:

- Client sends: "hello" (5 bytes)
- Server responds: "hey" (3 bytes)
- Client responds: "all ok" (6 bytes)
- Server responds: "bye" (3 bytes)



Transport Layer Implementation:

1. **Client → Server:** "hello" with SEQ 1, ACK 1
2. **Server → Client:** "hey" with SEQ 1, ACK 6 (received bytes 1-5, expecting byte 6)
3. **Client → Server:** "all ok" with SEQ 6, ACK 4 (received bytes 1-3, expecting byte 4)
4. **Server → Client:** "bye" with SEQ 4, ACK 12 (received bytes 6-11, expecting byte 12)

14.5.2 Real-World HTTP Example

Analyzing TCP behavior in web communication.

HTTP Request-Response Pattern

Web communication demonstrates practical TCP usage with asymmetric data flows:

- **Client:** HTTP GET request (200 bytes)
- **Server:** HTTP response with data (3,100 bytes)

Segmentation and Acknowledgment Pattern

Client Request:

- Single segment: SEQ 1, ACK 1 (200-byte GET request)

Server Response (segmented):

- Segment 1: SEQ 1, ACK 201 (bytes 1-1500)
- Segment 2: SEQ 1501, ACK 201 (bytes 1501-3000)
- Segment 3: SEQ 3001, ACK 201 (bytes 3001-3100)

Client Acknowledgments:

- ACK 1501: Confirming receipt of first 1500 server bytes
- ACK 3001: Confirming receipt of bytes 1-3000
- ACK 3101: Confirming receipt of entire response

Important Observations

- **Persistent SEQ:** Client ACKs use SEQ 201 (no new data to send)
- **Cumulative ACKs:** Each ACK confirms all bytes received so far
- **Data Segmentation:** Large messages split across multiple segments for efficient transmission
- **Piggybacked ACKs:** Acknowledgments combined with data when possible

Segmentation Rationale. Transport layers segment large messages to optimize network utilization and enable efficient error recovery at the segment level rather than requiring retransmission of entire large messages.

14.6 TCP Flow Control and Congestion Control

Managing sender transmission rates to prevent receiver overload and network congestion.
TCP must control the rate at which data is transmitted to prevent overwhelming both the receiver and the network infrastructure. This requires two complementary mechanisms operating simultaneously.

14.6.1 Maximum Segment Size and Segmentation

Understanding how TCP determines segment boundaries.

Maximum Segment Size (MSS)

The Maximum Segment Size represents the maximum amount of application-layer data that a single TCP segment may carry:

- **Determination:** MSS is dictated by network properties, particularly the bit error rate of links between sender and receiver
- **Discovery:** Transport layer discovers MSS through network interface properties and path MTU discovery
- **Typical Value:** 1500 bytes in current Internet infrastructure
- **Impact:** Large messages require segmentation across multiple TCP segments

Practical Segmentation Example

Consider an HTTP response scenario:

- Client sends 200-byte GET request → Single segment
- Server responds with 3100-byte data → Three segments (1500 + 1500 + 100 bytes)
- Each segment acknowledged independently for reliable delivery

14.6.2 Sender Window Management

Controlling the maximum number of unacknowledged bytes in transmission.

Sender Window Concept

The **sender window** indicates the maximum number of unacknowledged bytes that the sender may transmit:

- **Purpose:** Prevents overwhelming receiver and network
- **Dynamic Adjustment:** Changes based on current network and receiver conditions
- **Computation:** Minimum of flow control window and congestion control window

Dual Control Mechanisms

TCP sender window management combines two independent mechanisms:

1. **Flow Control** — Prevents overwhelming the receiver
   - Receiver explicitly signals maximum acceptable unacknowledged bytes
   - Communicated via receiver window field in TCP header
   - Direct feedback mechanism
2. **Congestion Control** — Prevents overwhelming the network
   - Sender estimates network capacity independently
   - No explicit network feedback available
   - Inferred through packet loss and acknowledgment patterns

Window Computation.
At each moment: Sender Window = $\min(\text{Flow Control Window}, \text{Congestion Window})$

14.7 TCP Congestion Control Algorithms

Adaptive algorithms for inferring and responding to network congestion.
TCP congestion control operates on the principle of **self-clocking**, where the sender adjusts its transmission rate based on acknowledgment patterns without explicit network feedback.

14.7.1 Key Congestion Control Concepts

Essential terminology and variables for understanding TCP algorithms.

Slow Start Threshold (ssthresh)

The **slow start threshold** is a critical variable that acts as TCP's "memory" of previous network congestion:

- **Purpose:** Remembers the congestion window size when congestion was last detected
- **Initial Value:** Set to a large value (effectively infinite) when connection starts
- **Updated When:** Congestion occurs (timeout or duplicate acknowledgments)
- **Calculation:** Always set to half the current congestion window when congestion detected
- **Usage:** Determines when to switch from exponential growth to linear growth

Why Half the Window? When congestion occurs, TCP assumes the network can handle approximately half of what was being sent, providing a conservative estimate for future transmissions.

Congestion Window (cwnd)

The **congestion window** represents the sender's estimate of how much data the network can handle:

- **Purpose:** Controls the maximum unacknowledged data the sender may transmit
- **Initial Value:** 1 Maximum Segment Size (conservative start)
- **Dynamic Adjustment:** Increases when network performs well, decreases when congestion detected
- **Units:** Measured in bytes

Maximum Segment Size

The **Maximum Segment Size** is the largest amount of application data that fits in one TCP segment:

- **Typical Value:** 1500 bytes in most modern networks
- **Determined By:** Network path properties and interface capabilities
- **Usage:** Unit of measurement for window adjustments

14.7.2 Self-Clocking Principle

How TCP infers network conditions from acknowledgment behavior.

Basic Self-Clocking Logic

The sender makes decisions based on acknowledgment patterns, following this simple logic:

- **New acknowledgment received** → Network conditions good → Increase congestion window
- **No new acknowledgment (timeout)** → Network conditions bad → Decrease congestion window
- **Adaptive Behavior** → Continuously adjusts to changing network conditions

Step-by-Step Decision Process:

1. Send data segments up to the current congestion window limit
2. Wait for acknowledgments from the receiver
3. **If acknowledgments arrive promptly:** Network can handle current load → increase window size
4. **If acknowledgments are missing or delayed:** Network may be congested → decrease window size
5. Repeat this process for every round of transmission

Algorithm Variations

Multiple congestion control algorithms implement this principle with different strategies:

- **Historical Algorithms:** Tahoe and Reno (foundational concepts we will study)
- **Modern Algorithms:** Cubic, New Reno (currently deployed in practice)
- **Key Design Questions:** How aggressively to react to new acknowledgments vs. missing acknowledgments

14.7.3 TCP Tahoe Algorithm

Foundational congestion control algorithm demonstrating core principles.
The Tahoe algorithm operates in two distinct states, each with different strategies for growing the congestion window based on network feedback.

Tahoe Algorithm States

Slow Start State. Aggressive window growth for initial connection:

- **Initial Window:** 1 Maximum Segment Size (conservative start)
- **Growth Rule:** Increase window by 1 Maximum Segment Size for each new acknowledgment received
- **Growth Pattern:** Window doubles every round-trip time (exponential growth)
- **Transition Condition:** Switch to congestion avoidance when window reaches slow start threshold

Slow Start Step-by-Step Process:

1. Start with congestion window = 1 Maximum Segment Size
2. Send data segments up to the current window limit
3. For each acknowledgment received, increase window by 1 Maximum Segment Size
4. Send more data with the larger window
5. Continue until window size equals slow start threshold

Congestion Avoidance State. Cautious window growth near suspected limits:

- **Growth Rule:** Increase window by $\frac{\text{Maximum Segment Size}^2}{\text{current window size}}$ for each new acknowledgment
- **Growth Pattern:** Window increases by approximately 1 Maximum Segment Size per round-trip time (linear growth)
- **Purpose:** Probe for additional network capacity carefully without causing congestion

Congestion Avoidance Step-by-Step Process:

1. Current window size has reached slow start threshold
2. Send data segments up to the current window limit
3. For each acknowledgment received, increase window by $\frac{\text{Maximum Segment Size}^2}{\text{current window size}}$ bytes
4. This small increase results in approximately 1 Maximum Segment Size growth per round-trip time
5. Continue until congestion is detected (timeout occurs)

Tahoe Transition Events

Timeout Event. Response to suspected severe congestion:
When the sender waits too long for an acknowledgment (timeout occurs), it assumes severe network congestion and responds conservatively:

1. **Update slow start threshold:** Set slow start threshold = $\frac{\text{current window size}}{2}$ (remember where congestion occurred)
2. **Reset congestion window:** Set congestion window back to 1 Maximum Segment Size (start over conservatively)
3. **Retransmit lost data:** Send the oldest unacknowledged segment again
4. **Return to slow start state:** Begin exponential growth again, but more cautiously

Why These Steps?

- **Halving the threshold:** Assumes network can handle about half of what caused congestion
- **Resetting to 1:** Conservative restart ensures we don't immediately cause more congestion
- **Retransmission:** Ensures data reliability despite network problems
- **Slow start restart:** Allows gradual ramp-up to test current network conditions

Window Reaches Slow Start Threshold. Transition to careful growth:
When the congestion window grows to equal the slow start threshold during slow start:

1. **State change:** Switch from slow start to congestion avoidance
2. **Window maintenance:** Keep current window size unchanged
3. **Growth pattern change:** Begin linear growth instead of exponential growth
4. **Rationale:** We're approaching the size that previously caused congestion, so be more careful

14.7.4 Detailed Tahoe Example

Step-by-step illustration of Tahoe algorithm behavior.

Scenario Setup

Consider Alice establishing a TCP connection to Bob with Maximum Segment Size = 100 bytes:

- **Connection Established:** Three-way handshake completed successfully
- **Initial State:** Alice starts in slow start state
- **Initial Congestion Window:** 1 Maximum Segment Size = 100 bytes
- **Initial Slow Start Threshold:** Undefined (will be set after first congestion event)

What Alice Will Do: Alice will start sending data conservatively, then gradually increase her sending rate based on network feedback. Let's trace through exactly what happens step by step.

Phase 1: Slow Start Growth

Round 1 — Conservative Beginning:

- **Alice sends:** Sequence 1 (bytes 1-100), current window = 100 bytes
- **Bob responds:** Acknowledgment 101 (confirming receipt of bytes 1-100)
- **Window update:** Alice increases window: 100 + 100 = 200 bytes (doubled)
- **Explanation:** Alice received 1 acknowledgment, so she adds 1 Maximum Segment Size to her window

Round 2 — Exponential Growth:

- **Alice sends:** Sequence 101 (bytes 101-200), Sequence 201 (bytes 201-300)
- **Bob responds:** Acknowledgment 201, Acknowledgment 301
- **Window update:** Alice increases window: 200 + 100 + 100 = 400 bytes (doubled again)
- **Explanation:** Alice received 2 acknowledgments, so she adds 2 Maximum Segment Sizes to her window

Round 3 — Network Limit Reached:

- **Alice sends:** Sequence 301, Sequence 401, Sequence 501, Sequence 601 (400 bytes total)
- **Network congestion:** All 4 segments lost due to network overload
- **Timeout occurs:** Alice detects no acknowledgments received within timeout period
- **Alice's conclusion:** The network cannot handle 400 bytes sent simultaneously

Phase 2: Timeout Response and Recovery

Congestion Detection and Response: Alice now realizes the network is congested and responds with the Tahoe algorithm's timeout procedure:

1. **Set slow start threshold:** $\frac{400}{2} = 200$ bytes (remember where congestion occurred)
2. **Reset congestion window:** 400 $\rightarrow$ 100 bytes (back to 1 Maximum Segment Size)
3. **Return to slow start state:** Begin conservative exponential growth again
4. **Retransmit lost data:** Send Sequence 301 (oldest unacknowledged segment)

Why Alice Does This:

- **Threshold = 200 bytes:** Alice remembers that 400 bytes caused problems, so 200 bytes is probably safe
- **Window = 100 bytes:** Start conservatively to avoid immediately causing more congestion
- **Slow start state:** Use exponential growth to quickly find the right sending rate
- **Retransmit:** Ensure data reliability by resending what was lost

Recovery Transmission:

- **Alice sends:** Sequence 301, current window = 100 bytes
- **Bob responds:** Acknowledgment 401 (confirming receipt of bytes 301-400)
- **Window update:** Alice increases window: 100 + 100 = 200 bytes
- **Important:** Window now equals slow start threshold (200 bytes)

Phase 3: Transition to Congestion Avoidance

Reaching Slow Start Threshold:

- **Current window:** 200 bytes = slow start threshold
- **State change:** Slow start $\rightarrow$ Congestion avoidance
- **Alice sends:** Sequence 401, Sequence 501 (200 bytes total)
- **Reason for change:** Alice is approaching the window size that previously caused congestion

Linear Growth Phase: Now Alice switches to much more conservative growth to avoid causing congestion again:

- **Acknowledgment 501 received:** Window = $200 + \frac{100^2}{200} = 200 + 50 = 250$ bytes
- **Acknowledgment 601 received:** Window = $250 + \frac{100^2}{250} = 250 + 40 = 290$ bytes
- **Acknowledgment 701 received:** Window = $290 + \frac{100^2}{290} = 290 + 34 = 324$ bytes

Understanding the Linear Growth Formula:

- **Formula:** New window = Current window + $\frac{(\text{Maximum Segment Size})^2}{\text{Current window}}$
- **Effect:** As window gets larger, the increase gets smaller per acknowledgment
- **Result:** Window grows by approximately 1 Maximum Segment Size per round-trip time
- **Contrast:** Much slower than slow start's exponential doubling

14.7.5 TCP Reno Algorithm Enhancement

Improved congestion control with fast retransmit and fast recovery.
TCP Reno enhances Tahoe by reacting more intelligently to packet loss, distinguishing between single segment loss and severe network congestion.

Fast Retransmit Mechanism

Duplicate ACK Detection Scenario: Consider Alice with 500-byte congestion window sending 5 segments where only the first is lost:

- **Alice sends:** SEQ 301, SEQ 401, SEQ 501, SEQ 601, SEQ 701
- **Network behavior:** First segment (SEQ 301) lost, others received successfully
- **Bob's response:** ACK 301, ACK 301, ACK 301, ACK 301 (duplicate ACKs)

Fast Retransmit Logic: Upon receiving 3 duplicate ACKs, Alice infers single segment loss rather than network collapse:

1. **Immediate retransmit:** Send SEQ 301 without waiting for timeout
2. **Set ssthresh:** $\frac{500}{2} = 250$ bytes
3. **Enter fast recovery:** Specialized state for handling isolated loss

Fast Recovery State

Window Inflation Strategy: Alice must handle segments that are officially unacknowledged but likely received:

1. **Base window:** Set to ssthresh = 250 bytes
2. **Inflation:** Add 3 MSS (300 bytes) for segments indicated by duplicate ACKs
3. **Working window:** 250 + 300 = 550 bytes
4. **New transmission:** Can send 50 additional bytes beyond already transmitted 500 bytes

Continued Fast Recovery: For each additional duplicate ACK received:

- **Interpretation:** One more segment confirmed received by Bob
- **Window adjustment:** Inflate by 1 MSS
- **Transmission:** Send additional data if window permits

Recovery Completion: When Alice receives new ACK (e.g., ACK 801):

1. **Conclusion:** Lost segment successfully retransmitted and received
2. **State transition:** Exit fast recovery
3. **Window reset:** Set congestion window to ssthresh (250 bytes)
4. **Mode switch:** Enter congestion avoidance state

Chapter 15

L15 — Forwarding and IP

15.1 What is the Network Layer?

The network layer is where we first see actual network devices called **routers**.
We have end-systems (like Alice's computer and Bob's computer) and routers that help move packets between them. A router is a smart device that can look at packets and decide where to send them next.

15.2 Packet Headers We Care About

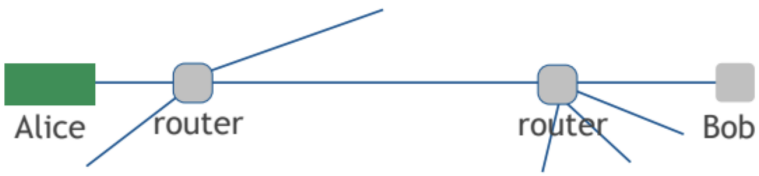
We need to understand two types of packet headers:

- **TCP header**: Contains *source port* and *destination port* (plus other things like SYN flag, checksum, sequence numbers, etc.)
- **IP header**: Contains *source IP address* and *destination IP address*

The IP addresses are what routers use to figure out where packets should go.

15.3 Two Main Jobs of the Network Layer

The network layer does two main things: **forwarding** and **routing**.



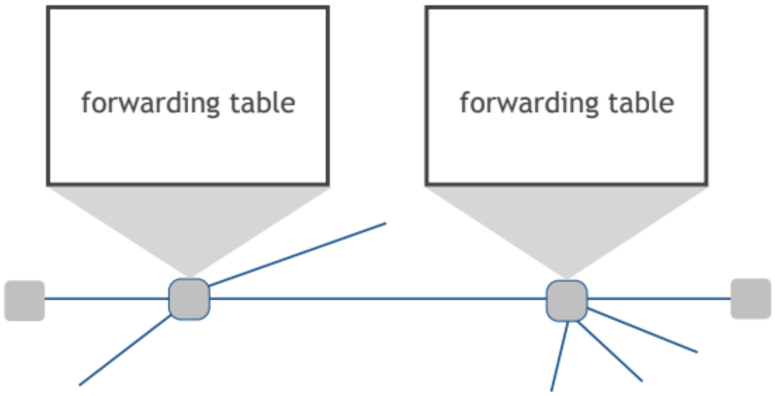
These work together to get packets from source to destination.

15.3.1 Forwarding: What Each Router Does

Forwarding is what happens when a packet arrives at a router. The router asks: "Where should I send this packet next?"
This is a quick decision that each router makes for every packet that comes through.

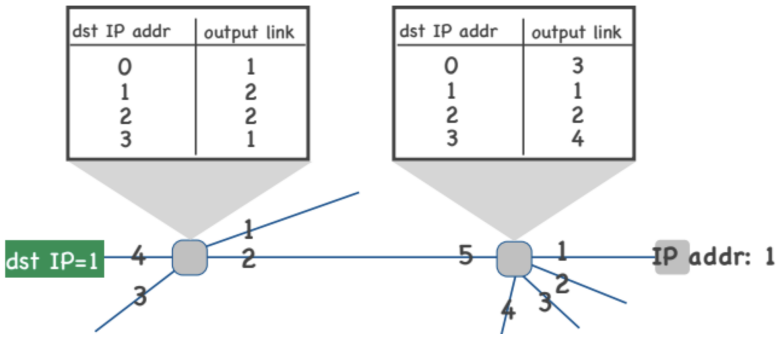
How Routers Are Set Up

Each router has several connections (called links). The router gives each link a number, like Link 1, Link 2, Link 3, etc.
The router also has a **forwarding table** - think of it like a phone book that says "if a packet is going to IP address X, send it out Link Y."



Step-by-Step: How Forwarding Works

Let's say Alice wants to send a packet to Bob, and Bob's IP address is 1.

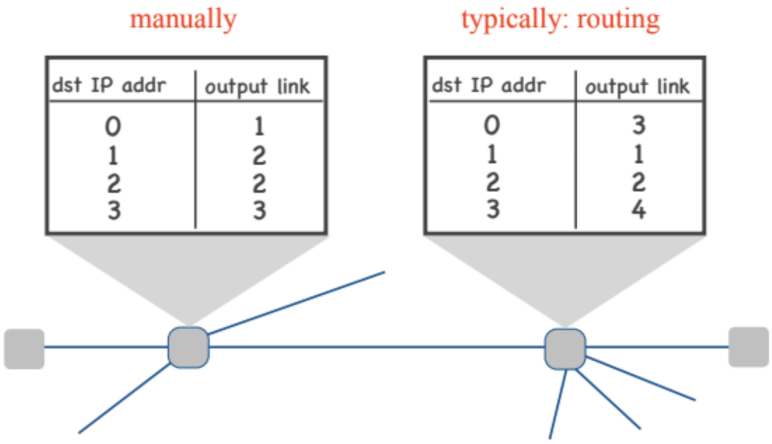


Here's what happens:

1. Alice puts "1" as the destination IP address in her packet
2. The first router gets the packet and reads "destination = 1"
3. The router looks in its forwarding table: "packets for IP 1 go out Link 2"
4. The router sends the packet out Link 2
5. The next router does the same thing with its own table
6. This continues until the packet reaches Bob

15.3.2 Routing: How Do Forwarding Tables Get Filled?

Routing is about filling up those forwarding tables. Someone has to tell each router "for IP address X, use Link Y."



Forwarding asks "where does this packet go?" Routing asks "how do we figure out where packets should go?"

Two Ways to Fill Forwarding Tables

- **Manual**: A network administrator types in the rules
- **Automatic**: Software figures it out and fills in the tables

Most of the time, it's automatic.

Centralized Routing: One Brain Controls Everything

Imagine one smart computer (called a **network controller**) that knows about all the routers in the network.

Good things about this approach:

- The controller sees the whole network, so it can make good decisions
- All routers get consistent information
- Easy to implement complex policies

The controller:

- Knows where all routers are and how they connect
- Figures out the best forwarding table for each router
- Sends this information to each router

Distributed Routing: Routers Talk to Each Other

Instead of one controller, the routers talk directly to each other and work together to figure out the best routes.

Good things about this approach:

- If one router breaks, the others keep working
- Works better when you have lots of routers
- Routers can quickly adapt to changes nearby

Real Internet: Mix of Both

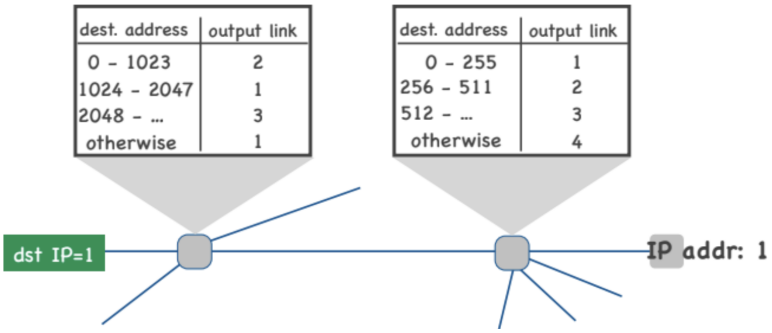
The actual Internet uses both approaches in different places.

15.4 Making Forwarding Tables Smaller

Here's a problem: there are about 4 billion possible IP addresses. Do routers really need 4 billion entries in their forwarding tables?
No! That would be way too big.

15.4.1 Using Ranges Instead of Individual Addresses

Instead of storing every single IP address, routers store **ranges** of IP addresses.



For example, instead of having separate entries for IP addresses 0, 1, 2, ..., 1023, a router can have one entry that says "all IP addresses from 0 to 1023 go out Link 2."

How This Works

When a packet arrives:

1. Router looks at the destination IP address
2. Router finds which range this IP address fits into
3. Router sends the packet out the link for that range

dest. address range			output link
0 - 3	0000 - 0011	00**	1
4 - 7	0100 - 0111	01**	2
8 - 11	1000 - 1011	10**	3
12 - 15	1100 - 1111	11**	4

15.4.2 IP Prefixes: A Smart Way to Write Ranges

Let's say we only have 16 IP addresses (0 through 15) to make this easier to understand.

Range	Binary	Prefix	Link
0-3	0000-0011	00**	1
4-7	0100-0111	01**	2
8-11	1000-1011	10**	3
12-15	1100-1111	11**	4

The "00**" means "any address that starts with 00". The ** can be anything (00, 01, 10, or 11).

15.4.3 Longest Prefix Matching: Handling Exceptions

Sometimes we need exceptions. What if most addresses starting with "00" go to Link 1, but address "0011" (which is 3) needs to go somewhere else?

Prefix	Link	What This Covers
00**	1	addresses 0, 1, 2, 3
0011	2	address 3 (exception!)

When a packet for address 3 arrives:

- It matches both "00**" and "0011"
- The router picks the **longer** (more specific) match: "0011"
- The packet goes to Link 2

This is called **longest prefix matching**.

More Exceptions = Bigger Tables

The more exceptions you need, the bigger your forwarding table gets:

dest. address range			output link
0 - 1	0000 - 0001	000*	1
2	0010	0010	2
3	0011	0011	3
4, 6, 7	0100, 0110, 0111	01**	2
5	0101	0101	4
8 - 15	1000 - 1111	1***	1
10	1010	1010	3

This is why we want to avoid lots of exceptions.

15.5 Why Location Matters for IP Addresses

For the Internet to work well, IP addresses should be related to location.

15.5.1 The Basic Idea

- Devices that are close together should have similar IP addresses
- Devices in the same building/city/country should share the same prefix
- Similar IP addresses should mean similar locations

15.5.2 Why This Helps

- **Smaller forwarding tables**: One entry can cover many destinations
- **Easier routing**: Routes can be grouped together
- **Faster updates**: Changes affect fewer table entries

15.5.3 What Happens When Location Rules Break

Here's an example of what goes wrong:

1. Let's say all EPFL computers have IP addresses starting with the same prefix
2. Routers worldwide can have one simple rule: "send all EPFL traffic toward Switzerland"
3. Now imagine EPFL students travel around the world but keep their EPFL IP addresses
4. Suddenly routers need special exception rules for each traveling student
5. If lots of people do this, forwarding tables become huge and unmanageable

This shows why keeping IP addresses tied to location is important for the Internet to work efficiently.

15.6 How IP Addresses Are Written

Now let's learn how to actually write and read IP addresses and IP prefixes.

15.6.1 IP Address Format

An IP address is just a number from 0 to $2^{32} - 1$ (that's about 4 billion different numbers).
We could write it in binary (all 0s and 1s), but that would be really long. Instead, we use something called "dot format."
Here's how it works:

- Take the 32-bit binary number
- Split it into 4 groups of 8 bits each
- Convert each group to a regular decimal number (0-255)
- Put dots between them

Example:

$$\text{Binary:} \quad 11011111\ 00000001\ 00000001\ 00000001 \tag{15.1}$$

$$\text{Dot format:} \quad 223.1.1.1 \tag{15.2}$$

15.6.2 IP Prefix Format

Remember how we said routers use ranges of IP addresses? We call these ranges **IP prefixes**.
An IP prefix is written as: **IP address / mask number**
For example: `223.1.1.0/24`

What the Mask Number Means

The mask number tells us how many bits from the left are "fixed" (can't change).
For `223.1.1.0/24`:

- The first 24 bits must stay the same
- The last 8 bits can be anything
- This covers all addresses from 223.1.1.0 to 223.1.1.255

In binary, this looks like:

$$\text{Fixed part:} \quad 11011111\ 00000001\ 00000001\ ******** \tag{15.3}$$

$$\text{Or simply:} \quad 223.1.1.* \tag{15.4}$$

The * means "can be any combination of 0s and 1s."

Different Ways to Write the Same Prefix

Here's something interesting: these are all the same IP prefix!

- `223.1.1.0/24`
- `223.1.1.74/24`
- `223.1.1.113/24`
- `223.1.1.*`

Why? Because they all have the same first 24 bits. The last 8 bits don't matter for defining the prefix.

More Examples

Bigger range: `223.1.1.0/8`

- Only the first 8 bits are fixed
- Covers: `223.*.*.*`
- That's all addresses from 223.0.0.0 to 223.255.255.255

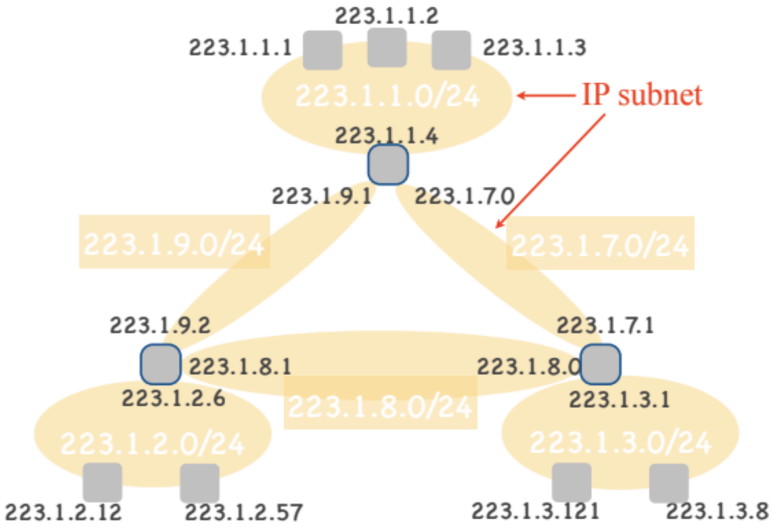
Tricky example: `223.1.1.0/12`

- First 12 bits are fixed
- In binary: `11011111 0000**** ******** ********`
- Covers: 223.0.0.0 to 223.15.255.255

Pro tip: When the mask isn't a multiple of 8 (like /12), it's tricky to figure out the range in your head. It's better to convert to binary and count bits carefully!

15.7 How the Internet Is Organized

The Internet is divided into chunks called **IP subnets**.



15.7.1 What Is an IP Subnet?

Think of an IP subnet as a neighborhood:

- It contains end-systems (computers, phones, etc.)
- It has routers at its "borders" that connect to other neighborhoods
- Everyone in the neighborhood has similar addresses (same IP prefix)
- The routers don't live "inside" the neighborhood - they're at the edges

15.7.2 How IP Addresses Are Assigned in Subnets

Each subnet gets its own IP prefix. For example:

- Top subnet might get `223.1.1.0/24`
- Bottom subnet might get `223.1.2.0/24`
- Middle subnets might get `223.1.3.0/24`, etc.

All devices in a subnet get IP addresses from that subnet's prefix.

15.7.3 Routers Have Multiple IP Addresses

Here's something cool: each router has one IP address for each subnet it touches.
Look at the top router in the picture:

- It touches 3 different subnets
- So it has 3 different IP addresses
- Each address belongs to the prefix of that subnet

It's like living at the corner of three neighborhoods - you need an address in each one!

15.7.4 How Routers Use This Information

When a router gets a packet, it looks at the destination IP address and asks:

- "Is this address in one of my local subnets?" → Send it directly there
- "Is this address in a foreign subnet?" → Forward it toward that subnet

For example:

- Packet for `223.1.1.1` → "That's in my top subnet!" → Send it up
- Packet for `223.1.2.16` → "That's in a different subnet" → Forward it toward that subnet

15.8 Special IP Addresses

15.8.1 Broadcast Address

Each subnet has a special **broadcast address**:

- It's the biggest IP address in the subnet
- When you send a packet to this address, it goes to *everyone* in the subnet
- For subnet `223.1.1.0/24`, the broadcast address is `223.1.1.255`

It's like shouting "Hey everyone!" in a room.

15.9 How Do Organizations Get IP Addresses?

15.9.1 Getting IP Prefixes

Organizations get their IP prefixes from:

- Their Internet Service Provider (ISP)
- A regulatory body (for big organizations)

15.9.2 Assigning Individual IP Addresses

Once an organization has its prefix, it assigns individual addresses:
For router interfaces:

- Usually done manually by network administrators

For end-systems (computers, phones):

- Can be done manually
- More often done automatically using DHCP (Dynamic Host Configuration Protocol)

DHCP is like having an automatic address assignment system - when your laptop joins a network, DHCP gives it an available IP address from that network's range.

15.10 Best-Effort Delivery

Here's an important thing to understand about the Internet: it provides **best-effort delivery**.

15.10.1 What Best-Effort Means

- The Internet tries its best to deliver your packets
- But it makes **no promises** that they'll actually arrive
- Packets might get lost, delayed, or arrive out of order
- The network says "I'll do my best, but no guarantees!"

15.10.2 Why Best-Effort?

This might sound bad, but it's actually a smart design choice:

- Keeps the network simple and fast
- Makes it cheaper to build and operate
- Applications can add their own reliability on top if they need it
- Works well for most uses (web browsing, email, etc.)

Think of it like the postal service - they try to deliver your mail, but sometimes letters get lost. For important things, you can pay extra for certified mail (like how applications can add extra reliability).

15.11 Virtual Circuits: A Different Way to Do Networking

So far we've talked about how the Internet works today (packet switching with best-effort delivery). But there's another way to build networks called **virtual circuits**.

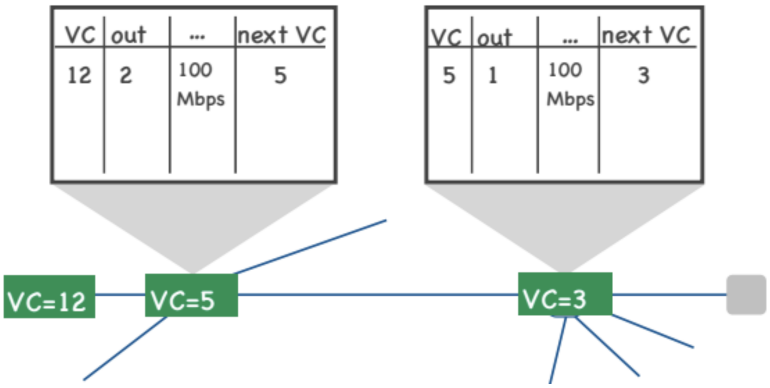
15.11.1 The Problem with Best-Effort

Remember how the Internet makes no promises about packet delivery? Sometimes you want guarantees. For example:

- Video calls need consistent, fast delivery
- Important business applications can't afford lost packets
- Some applications need a minimum guaranteed speed

15.11.2 How Virtual Circuits Work

Virtual circuits are like making a reservation at a restaurant - you book resources in advance. Here's how Alice could get a guaranteed 100 Mbps connection to Bob:



Step 1: Connection Setup Request

Alice sends a special "connection-setup request" packet that says: "I want a network connection to Bob with guaranteed 100 Mbps speed."

Step 2: Each Router Decides

The packet travels through routers on the path to Bob. Each router asks itself:

- "Do I have enough spare capacity to guarantee 100 Mbps?"
- "Can I reserve these resources for Alice and Bob?"

If a router says yes:

- It reserves 100 Mbps of capacity for this connection
- It assigns a Virtual Circuit (VC) number to this connection (like giving it a name)
- It creates a table entry to remember this reservation

For example:

- First router assigns VC #12
- Second router assigns VC #5
- Third router assigns VC #3

Step 3: Connection Establishment

If ALL routers on the path agree, Bob gets the request and sends back an "OK" message. This travels back through all the routers so they can coordinate their VC numbers.
Now Alice has a guaranteed 100 Mbps "highway" to Bob!

Step 4: Using the Virtual Circuit

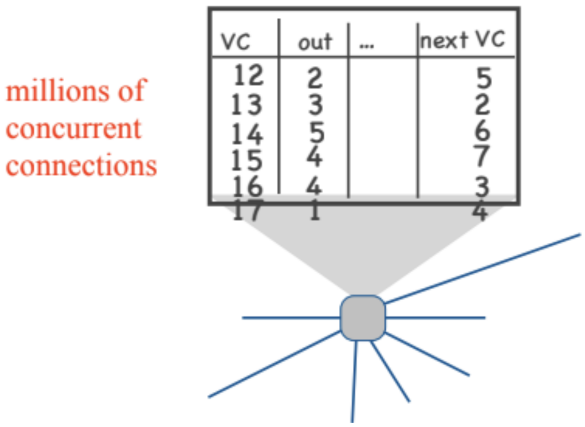
When Alice sends packets to Bob:

- She puts "VC #12" in each packet header
- First router sees "VC #12", knows this is Alice's guaranteed connection
- Router changes the header to "VC #5" and forwards to next router
- Second router sees "VC #5", changes to "VC #3", forwards to third router
- And so on until the packet reaches Bob

Each router knows exactly how to treat these packets because of the reservation.

15.11.3 The Big Challenge: Too Much State

Here's the problem with virtual circuits: routers have to remember LOTS of information.



Memory Requirements

Each router needs to keep a table entry for every active connection passing through it. A busy Internet router might have:

- Millions of concurrent connections
- Each connection needs memory for state information
- This adds up to huge memory requirements

15.11.4 Packet-Switched Networks (Like Today's Internet)

How they work:

- Use packet switching - no network-layer connections
- Best-effort delivery (no guarantees)
- Routers only store destination prefixes and output links
- State is populated by routing protocols

Good for: Best-effort service

15.11.5 Why the Internet Chose Packet Switching

Packet switching won because it:

- Makes forwarding tables smaller (no per-connection state in routers)
- Makes routers simpler (no connection setup/teardown needed)
- Eliminates security risks from connection-based attacks

15.12 A Fundamental Internet Principle

Every computer should be able to talk to every other computer.

15.12.1 Global Reachability

The Internet is built on this idea:

- Every end-system must be reachable from any other end-system
- This requires a globally unique IP address for every end-system
- No two devices anywhere in the world should have the same IP address

This is like having a postal system where every house has a unique address - no duplicates allowed anywhere!

15.13 The IP Address Crisis

In the 2000s, we started running out of IP addresses.

15.13.1 The Problem

- IPv4 has about 4 billion possible addresses
- The Internet grew faster than expected
- We were running out of unique addresses to assign

15.13.2 Two Solutions

Solution 1: IPv6

- A new version of IP with way more addresses
- Deployed in many areas, but not everywhere yet
- Long-term solution but takes time to implement

Solution 2: Network Address Translation (NAT)

- A clever workaround that lets multiple devices share one public IP address
- Widely deployed and working today
- Has some limitations (which we'll explain)

15.14 Network Address Translation (NAT)

How to let many devices share one public IP address.

15.14.1 The Basic Idea: Private Address Spaces

Some IP address ranges are designated as "private":

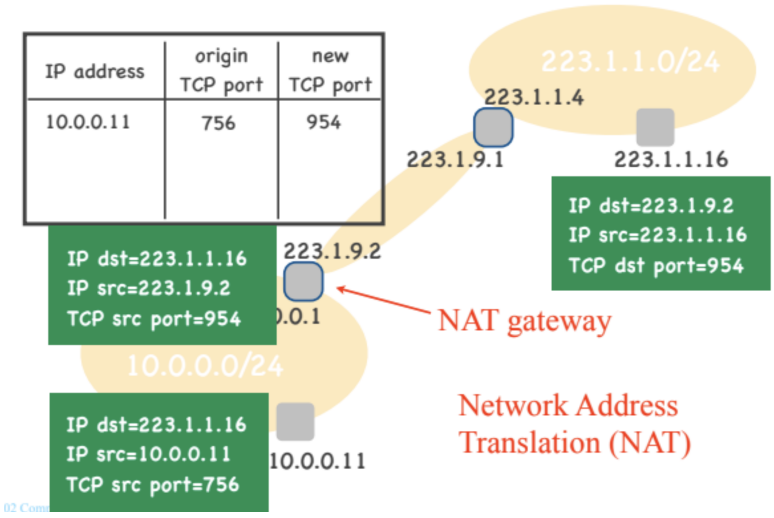
- Example: 10.0.0.0/24 is a private range
- Multiple networks can use the same private addresses
- Like having multiple houses with address "123 Main Street" - it's OK as long as they're in different towns

15.14.2 The Rule for Private Addresses

- Private IP addresses can only be used within their local network
- Packets with private addresses can NEVER leave the local network
- It's like calling "John!" in your house - only works if there's only one John there

15.14.3 How NAT Works: A Step-by-Step Example

Let's say device 10.0.0.11 (private address) wants to talk to 223.1.1.16 (public address):



Step 1: Outgoing Packet

- Device 10.0.0.11 sends packet with source=10.0.0.11, destination=223.1.1.16
- It also has a TCP source port (let's say 756)

Step 2: NAT Gateway Translation

The NAT gateway (border router) does magic:

- Replaces source IP 10.0.0.11 with its own public IP 223.1.9.2
- Replaces source port 756 with a new port number (say 954)
- Remembers this mapping in a table: "10.0.0.11:756 ↔ 954"

Step 3: Response Packet

- Server 223.1.1.16 responds to 223.1.9.2:954
- It thinks it's talking to 223.1.9.2, not 10.0.0.11

Step 4: Return Translation

- NAT gateway gets response packet addressed to 223.1.9.2:954
- Looks up port 954 in its table: "Oh, this goes to 10.0.0.11:756"
- Changes destination to 10.0.0.11:756 and forwards internally

15.14.4 What NAT Does

For outgoing packets:

- Rewrites source IP address and port number
- Maps original address:port to new port number
- Stores this mapping for future use

For incoming packets:

- Rewrites destination IP address and port number
- Uses stored mapping to find the right internal device

15.15 Problems with NAT

NAT works, but it breaks some fundamental Internet principles.

15.15.1 Problem 1: You Can't Reach Devices from Outside

- Devices with private addresses are "hidden" behind the NAT gateway
- External devices can't initiate connections to them
- Internal devices can call out, but external devices can't call in
- Unless you manually configure the NAT gateway with special rules

This breaks the global reachability principle!

15.15.2 Problem 2: NAT Gateways Need State

Remember how we said routers shouldn't keep per-connection state? Well, NAT gateways do exactly that:

- They remember mappings for every active connection
- This is the same problem we had with virtual circuits!

Why is this OK?

- NAT gateways are usually at the edge (like your home router)
- They only handle connections from a small number of devices
- Not millions of users like a core Internet router

15.15.3 Problem 3: It's a Layering Violation

NAT breaks the clean separation between network layers:

- The network layer (IP) shouldn't care about transport layer info (TCP ports)
- But NAT has to look at and modify TCP port numbers
- This makes the system more complex and fragile

Chapter 16

L16 — Routing and BGP

16.1 Quick Review: Forwarding vs. Routing

Let's quickly review the two main jobs of the network layer from last time.

16.1.1 Forwarding: What Each Router Does

Forwarding is a local operation that happens whenever a packet arrives at a router:

- Goal: Figure out which output link to send the packet to
- How: Read the destination IP address from the packet header and look it up in the forwarding table
- It's fast and happens for every single packet

16.1.2 Routing: How Forwarding Tables Get Filled

Routing is a network-wide operation that populates forwarding tables:

- Goal: Figure out what should go in each router's forwarding table
- How: Run routing algorithms either on a centralized controller or on the routers themselves
- It's slower and happens when the network changes

16.2 How Internet Forwarding Works

The Internet uses packet switching with best-effort delivery.
Remember these key points about Internet forwarding:

- Uses packet switching (no virtual circuits or network-layer connections)
- Provides best-effort service (no guarantees about delivery or performance)
- Each forwarding table entry contains: destination IP prefix + output link
- These entries are populated by routing protocols

We talked a lot about forwarding tables in the last lecture, but we didn't really discuss how big they are. Today we'll get more concrete about forwarding table sizes and how routing works.

16.3 Every Router Knows About Every Destination

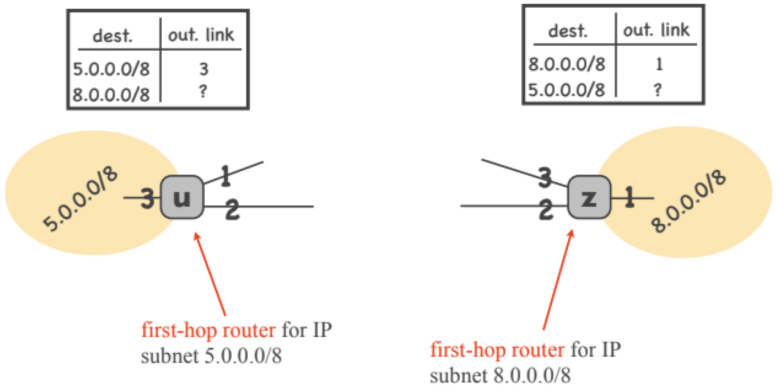
Every router on the Internet can reach every public IP address.
Let's say there's an end-system somewhere with IP address 8.0.0.1. Here's the amazing part:

- Every router on the Internet has an entry in its forwarding table that matches 8.0.0.1
- That entry tells the router which output link to use to get packets closer to 8.0.0.1
- This is true for ANY public Internet IP address, not just 8.0.0.1

This is what makes global connectivity possible - every router knows how to reach every destination!

16.4 First-Hop Routers: Where It All Starts

Some routers have a special job - they're the first ones to handle packets from local networks.



Let's look at an example with two IP subnets:

- Left subnet: uses IP prefix 5.0.0.0/8
- Right subnet: uses IP prefix 8.0.0.0/8

Each subnet connects to a router:

- Router u connects to the left subnet (5.0.0.0/8)
- Router z connects to the right subnet (8.0.0.0/8)

We call these **first-hop routers** because they're the first routers to handle packets coming from their local subnets.

16.4.1 What First-Hop Routers Know Automatically

Each first-hop router automatically knows about its own local subnet:
Router u knows:

- "Any packet for 5.0.0.0/8 goes out my link 3 (to the local subnet)"
- The network administrator configures this when setting up the router

Router z knows:

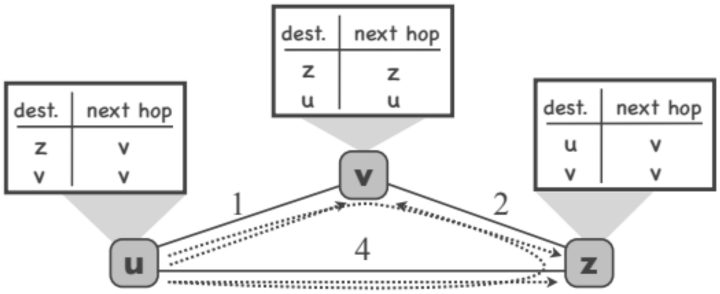
- "Any packet for 8.0.0.0/8 goes out my link 1 (to the local subnet)"
- This is also configured manually by the administrator

16.4.2 The Big Question: What About Foreign Subnets?

But here's the problem: how does router u learn to forward packets to router z's subnet (8.0.0.0/8)? And how does router z learn about router u's subnet (5.0.0.0/8)?
The network administrator can't manually configure every possible destination - there are millions of them!
The answer: A routing protocol.
Routing protocols are software systems that automatically figure out how to reach all destinations and populate the forwarding tables accordingly.

16.5 How Routing Protocols Work: A Simple Example

Let's see what happens when routers need to figure out the best paths to each other.
Suppose we have 3 routers: u, v, and z. When these routers participate in a routing protocol, their goal is to learn the best path to reach each other.



least-cost path from u to z: u v z
least-cost path from u to v: u v

Let's think about router u. It needs to answer these questions:

- "When I want to send a packet to router z, should I send it directly or through router v?"
- "When I want to send a packet to router v, what's the best way?"

The other routers (v and z) need to answer similar questions about reaching their destinations.

16.5.1 What Does "Best Path" Mean?

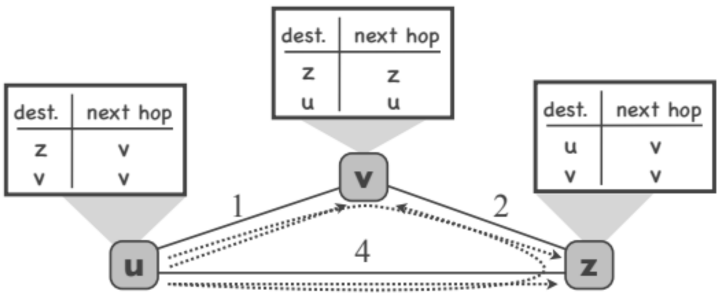
To pick the best path, we need to define what "best" means.
Each link in the network has a **cost** that represents how "bad" or expensive it is to use that link. The cost could be based on:

- How long it takes for signals to travel across the link (propagation delay)
- How much money it costs to send traffic over that link
- How congested the link is
- Some other cost metric that network administrators choose

The cost of a **path** is simply the sum of the costs of all links in that path. The **best path** from one router to another is the path with the lowest total cost—this is called the **least-cost path**.
Important note: In our examples, a link's cost is the same in both directions. In reality, links can have different costs in each direction, but we'll keep it simple for now.

16.6 Working Out the Best Paths: Step by Step

Let's figure out the best paths for router u in our example.
Looking at the network diagram, let's calculate the costs:



least-cost path from u to z: u v z
least-cost path from u to v: u v

16.6.1 From Router u to Router z

Router u has two options to reach router z:

Option 1: Direct path

- Go directly from u to z
- Cost = 4

Option 2: Indirect path through v

- Go from u to v, then from v to z
- Cost = 1 + 2 = 3

Since $3 < 4$, the best path is the indirect one through router v. Therefore, router u should send packets destined for z to router v as the next hop.

16.6.2 From Router u to Router v

Router u has two options to reach router v:
Option 1: Direct path

- Go directly from u to v
- Cost = 1

Option 2: Indirect path through z

- Go from u to z, then from z to v
- Cost = 4 + 2 = 6

Since $1 < 6$, the best path is the direct one. Therefore, router u should send packets destined for v directly to v.

16.7 Link-State Routing Algorithms

This is the first major family of routing algorithms.
The process we just did by hand is an example of a **link-state routing algorithm**. Here's the general idea:

- **Input:** A complete map of the network—all the routers and the costs of all links connecting them.
- **Output:** The least-cost path from one starting router to every other router in the network.

The most famous link-state routing algorithm is called **Dijkstra's algorithm**.

Why It's Called "Centralized"

Link-state routing algorithms are sometimes called **"centralized"** algorithms.

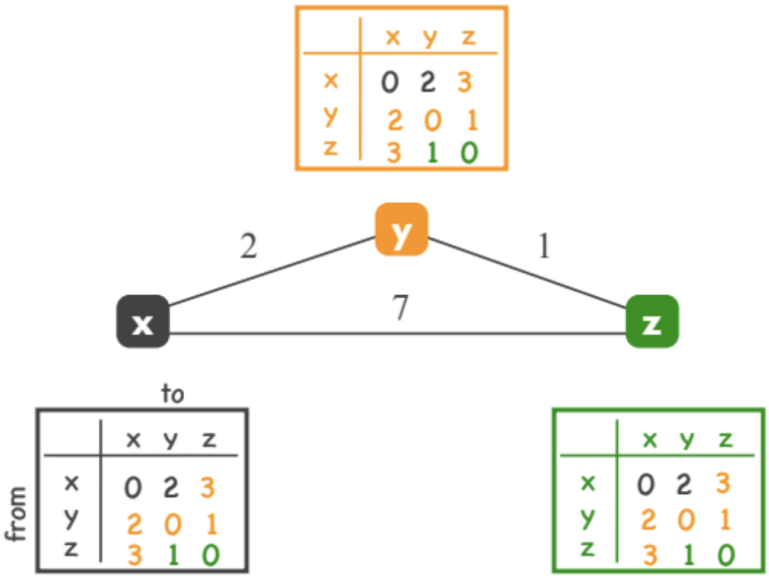
- Each router first gets a copy of the entire network map.
- Once a router has the full map, it can calculate the best paths to all other routers **by itself**, without any more communication.
- Each router runs the algorithm independently on its own copy of the map.

16.8 Distance-Vector Routing Algorithms

This is the second major family of routing algorithms, and it works very differently.
Instead of knowing the whole map, routers in a **distance-vector** algorithm only know about their immediate neighbors. They work together in rounds to figure out the best paths.

16.8.1 The Setup: What Each Router Knows Initially

Let's use our 3-router example again. In a distance-vector algorithm, each router starts by creating a table with the costs to its direct neighbors.



Router x knows:

- Cost to itself (x) is 0
- Cost to neighbor y is 2
- Cost to neighbor z is 7

Router y knows:

- Cost to neighbor x is 2
- Cost to itself (y) is 0
- Cost to neighbor z is 1

Router z knows:

- Cost to neighbor x is 7
- Cost to itself (z) is 0
- Cost to neighbor y is 1

They don't know about any indirect paths yet!

16.8.2 Round 1: Exchanging Information

In the first round, all routers send their tables to their immediate neighbors.

- Router x sends its table to y and z.
- Router y sends its table to x and z.
- Router z sends its table to x and y.

16.8.3 Round 1: Updating the Tables

Now, each router looks at the information it received from its neighbors and asks: "Can my neighbors get me somewhere cheaper than I can get there myself?"
Let's look at router x:

- Router x knows it can get to z with a direct cost of 7.
- But it just learned from router y that "y can get to z with a cost of 1".
- Router x knows it can get to y with a cost of 2.
- So, x thinks: "I can get to y (cost 2), and y can get to z (cost 1). The total cost is 2 + 1 = 3."
- Since 3 is cheaper than 7, router x updates its table: "The new best cost to reach z is 3, by going through y."

Now let's look at router z:

- Router z knows it can get to x with a direct cost of 7.
- But it just learned from router y that "y can get to x with a cost of 2".
- Router z knows it can get to y with a cost of 1.
- So, z thinks: "I can get to y (cost 1), and y can get to x (cost 2). The total cost is 1 + 2 = 3."
- Since 3 is cheaper than 7, router z updates its table: "The new best cost to reach x is 3, by going through y."

Router y doesn't find any new cheaper paths in this round because its direct connections are already the best.

16.8.4 Round 2: The Final Check

In the second round, all routers exchange their new, updated tables with their neighbors again.
This time, when they check the new information, they find that they can't improve their paths any further. Everyone already has the best possible path.
When no router can find a cheaper path, the algorithm is done! Each router has now successfully computed the least-cost path to every other router in the network.

16.9 The Bellman-Ford Algorithm

The specific distance-vector algorithm we just learned has a name.
The particular distance-vector routing algorithm we discussed is called the **Bellman-Ford algorithm**.

16.9.1 How Bellman-Ford Works

Here's the process in summary:

- **Step 1:** All neighbors exchange their routing tables
- **Step 2:** Each router checks whether it can use the new information to improve its current paths
- **Step 3:** If improvements are found, update the table
- **Repeat:** Continue until no router can improve its paths any further

The algorithm ends when no improvement is possible, which means everyone has found their optimal paths.

16.10 Link-State vs. Distance-Vector: The Big Picture

Both approaches solve the same problem but in very different ways.
We've now learned about two major families of routing algorithms:

- **Link-state algorithms** (like Dijkstra's)
- **Distance-vector algorithms** (like Bellman-Ford)

The goal is the same for both: Compute the least-cost path from each router to every other router in the network.

16.11 Which Approach Is Better?

Each approach has its own advantages and trade-offs.

16.11.1 Link-State Advantages: Speed

Link-state converges faster:

- Each router starts with the full picture of the network
- Once a router has the complete map, it can calculate all paths immediately
- The computation time can be reduced by using faster computers
- No waiting for information to propagate through multiple rounds

Think of it this way: if you have the complete map, you can plan the fastest route right away.

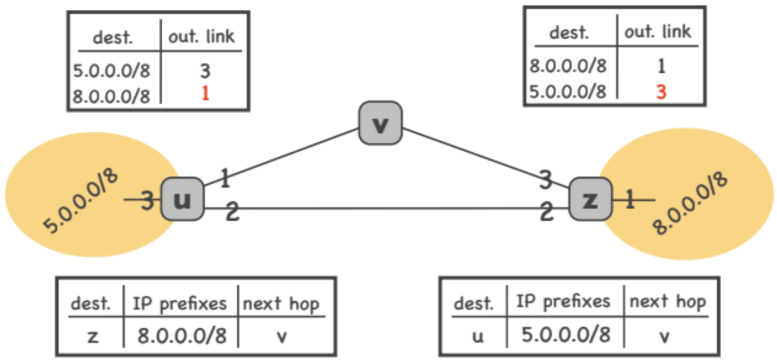
16.11.2 Distance-Vector Advantages: Efficiency

Distance-vector uses less bandwidth:

- Each router only talks to its immediate neighbors
- The number of messages per round stays constant regardless of network size
- No need to flood the entire network with information
- More efficient use of network resources

16.12 Bringing It All Together: How Routing Completes Forwarding Tables

Let's see how routing protocols solve the real problem we started with.
Remember our example from the beginning? Router u knew how to handle packets for its local subnet (5.0.0.0/8) but didn't know what to do with packets destined for 8.0.0.0/8.



Here's how a routing protocol solves this problem:

Step 1: Routers Advertise What They Own Each router announces which IP prefixes it "owns":

- Router u advertises: "I own IP prefix 5.0.0.0/8"
- Router z advertises: "I own IP prefix 8.0.0.0/8"

Step 2: Routing Protocol Finds Best Paths Router u, router v, and router z all participate in a routing protocol (could be Dijkstra, Bellman-Ford, or any other routing algorithm). Through this protocol:

- Router u learns that the best next hop to reach router z is router v
- Router u also learns that router z owns IP prefix 8.0.0.0/8

Step 3: Complete the Forwarding Table Router u combines this information:

- "To reach 8.0.0.0/8, I need to get to router z"
- "To reach router z, my best next hop is router v"
- "Router v is reachable through output link 1"
- **Conclusion:** "Map IP prefix 8.0.0.0/8 to output link 1"

Similarly, router z learns that:

- The best next hop to reach router u is router v
- Router u owns IP prefix 5.0.0.0/8
- **Conclusion:** "Map IP prefix 5.0.0.0/8 to output link 3"

Now both routers have complete forwarding tables and can route packets anywhere!

16.13 The Reality Check: Internet Routing Challenges

The algorithms we've learned work great in theory, but the real Internet is much more complex. Designing a routing algorithm for the entire Internet faces some serious challenges:

16.13.1 Challenge 1: Scale

The Internet is **massive**:

- Millions of routers worldwide
- Millions of IP subnets to track
- Constant changes as networks go up and down

Why our simple algorithms won't work:

- **Link-state** would cause flooding disasters - imagine every router trying to tell every other router about its links!
- **Distance-vector** would never converge - it would take too many rounds for information to propagate across millions of routers
- Forwarding tables would be enormous - one entry for every IP subnet in the world

16.13.2 Challenge 2: Administrative Autonomy

Different parts of the Internet are owned by different organizations:

- Internet Service Providers (ISPs) like Comcast,Swisscom
- Universities like EPFL
- Companies like Google, Facebook, Amazon
- Government networks

The problems this creates:

- ISPs may not want to do least-cost routing (they have business reasons for their choices)
- ISPs want to hide their internal network details from competitors
- Different organizations have different policies about who can send traffic through their networks

16.14 The Internet's Solution: Hierarchical Routing

The Internet addresses these challenges through **hierarchy**: dividing into separate networks called **Autonomous Systems** (ASes).

16.14.1 Autonomous Systems

An AS is a collection of routers under single administrative control (e.g., ISP network, university network, company network). Each AS gets a unique AS number.

16.14.2 Two-Level Routing

Intra-AS routing: Within each AS, routers run their chosen algorithm (Dijkstra, Bellman-Ford, etc.)
Inter-AS routing: Between ASes, border routers use BGP to exchange routes

16.14.3 Benefits

Scale: Thousands of small algorithms instead of one massive algorithm
Autonomy: Each AS controls its own routing policies and can hide internal details

16.15 Examples of Intra-AS Routing

Different ASes can choose different routing algorithms for their internal networks.

AS 1 might choose:

- Dijkstra's algorithm for fast convergence
- Link costs based on bandwidth and delay

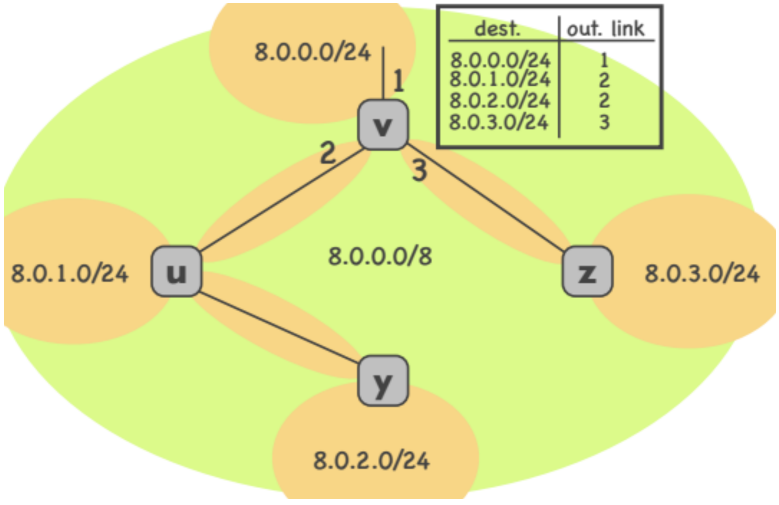
AS 2 might choose:

- Bellman-Ford algorithm to save bandwidth
- Link costs based on monetary cost

Each AS makes its own decision based on its specific needs and constraints.

16.16 A Concrete Example: How an AS Works

Consider an AS that has 4 routers, each providing connectivity to 1 IP subnet:

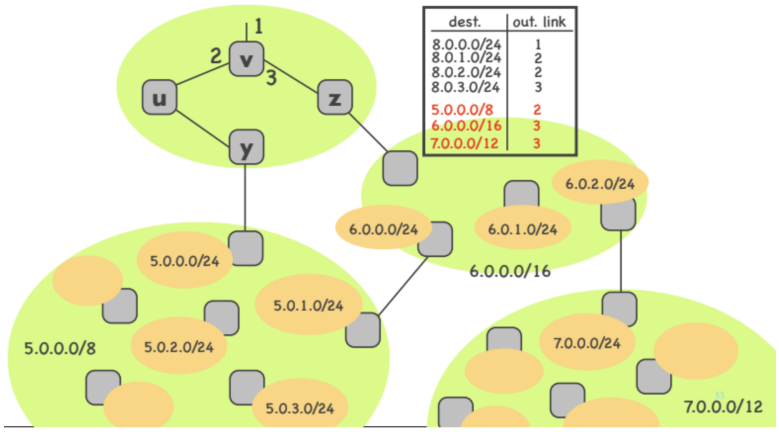


16.16.1 Step 1: Learn About Local Destinations

Each of the 4 routers must discover the best path to each local router and subnet. They achieve this by participating in an intra-AS routing protocol (like Dijkstra or Bellman-Ford).

16.16.2 Foreign Destinations

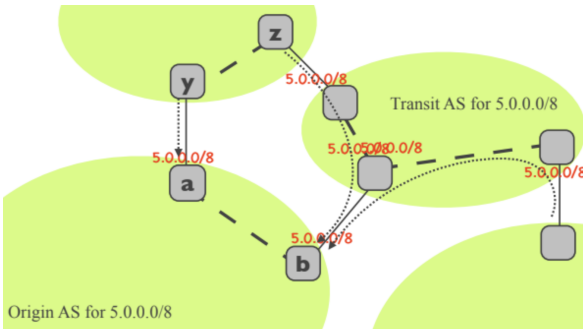
Each router must also learn routes to foreign ASes, but **not** to every individual subnet in those ASes.



Router v needs only 1 entry for each of the 3 foreign ASes, not separate entries for every subnet within those ASes. This dramatically reduces forwarding table sizes.

16.17 Border Routers and BGP

Border routers sit at the "edge" of each AS and handle communication between different ASes.



All border routers participate in **Border Gateway Protocol (BGP)**, a variant of Bellman-Ford. Through BGP, each border router:

- Advertises prefixes from its own AS to neighbors
- Learns routes to foreign prefixes

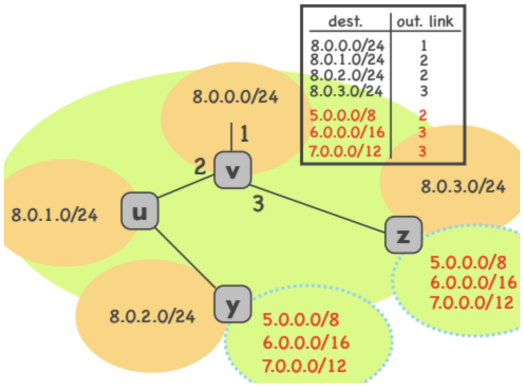
16.17.1 BGP Example

- Bottom left AS aggregates local subnets into 5.0.0.0/8 (becomes the "origin AS")
- Border routers a and b advertise "route to 5.0.0.0/8" to their neighbors
- Top right AS decides to do transit, propagating the route further
- Eventually all ASes learn how to reach 5.0.0.0/8

16.18 How Non-Border Routers Learn External Routes

Router v (not a border router):

- Learns local routes from other local routers (intra-AS routing)
- Learns foreign routes from border routers y and z
- Chooses least-cost route when multiple options exist



16.19 Internet Routing Summary

16.19.1 Intra-AS Routing

- **Participants:** All routers in the same AS
- **Protocols:** OSPF, RIP, others (each AS chooses)
- **Goal:** Propagate routes within local AS

16.19.2 Inter-AS Routing

- **Participants:** Border routers between ASes
- **Protocol:** BGP (universal - only one protocol used)
- **Goal:** Propagate routes between ASes

Internet Architecture Foundation

The Internet's network layer rests on two components:

- **IP:** Specifies forwarding, packet format, addressing
- **BGP:** Inter-domain routing protocol enabling global connectivity

Chapter 17

L17 - The Link Layer

Having studied how the network layer routes packets across the Internet, we now examine how individual network segments deliver those packets locally. This brings us to the link layer.

17.1 Fundamentals

17.1.1 Packet Switch Types

Networks contain two fundamentally different types of packet switches:

- **Link-layer switches**: Operate at physical and link layers, forwarding packets within a single network segment using physical addresses
- **Network-layer switches (routers)**: Operate at physical, link, and network layers, routing packets between different networks using IP addresses

Terminology: Throughout this course, "switch" refers to link-layer switches, while "router" refers to network-layer switches.

17.1.2 Scope Comparison: Link vs Network Layer

The key distinction lies in their operational scope:

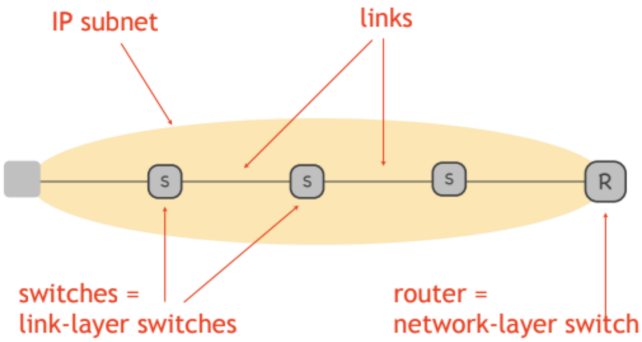
Network Layer: Delivers packets *end-to-end across the entire network* (e.g., New York to Tokyo across the Internet)

Link Layer: Delivers packets *across a single physical link* (e.g., laptop to wireless access point)

Think of the network layer as the postal service routing mail globally, while the link layer is the local delivery truck carrying mail from the post office to your house.

17.1.3 Layer Roles Within IP Subnets

Consider a single IP subnet—essentially one network segment like your home WiFi:

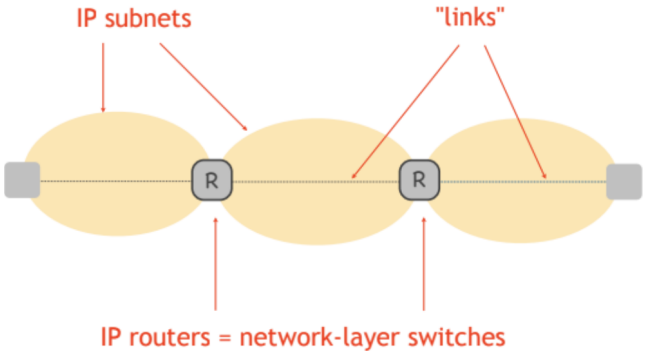


Within this subnet containing end-systems, boundary routers, and interconnecting switches:

- **Link layer:** Moves packets across individual physical connections (one "hop")
- **Network layer:** Moves packets across the entire subnet (multiple hops, e.g., from left computer to router R)

17.1.4 Internet-Scale Architecture

Zooming out to the full Internet reveals multiple interconnected IP subnets:



This layered approach enables Internet scalability:

- Network layer handles inter-subnet routing without knowing physical link details
- Link layer handles local delivery without understanding global routing

Each layer focuses on its specific scope, making the overall system manageable and efficient.

17.1.5 Perspective Matters: Two Views of "Link Layer"

The term "link layer" actually has different meanings depending on your perspective:

IP Subnet Perspective: Link layer moves packets across individual physical links (cable, WiFi connection)

Internet Perspective: Link layer moves packets across entire IP subnets (what we call network layer within a subnet)

From the Internet's viewpoint, each IP subnet is just one "link" in the larger network. This is why the Internet's "link layer" is actually the network layer of individual subnets.

17.2 Link-Layer Services

The link layer (focusing on physical links within subnets) provides several key services:

17.2.1 Error Detection

- Receivers detect and drop corrupted packets using checksums
- Similar to UDP/TCP error detection but at the physical link level

17.2.2 Reliable Data Delivery

- Sender/receiver detect corruption and loss, attempting recovery
- Uses checksums, acknowledgments, and retransmissions
- Typically deployed only on error-prone links (especially wireless)

Why Link-Layer Reliability?

Since TCP provides end-to-end reliability, why also implement it at the link layer? The answer is **performance optimization**.
Consider a long network path where one link experiences frequent packet loss:
Without link-layer reliability:

1. TCP times out waiting for acknowledgments
2. Resets congestion window to minimum
3. Slowly ramps up transmission rate again
4. Overall throughput drops significantly

With link-layer reliability:

1. Link layer locally detects and retransmits lost packets
2. TCP never sees the packet loss
3. No TCP timeout or congestion window reset
4. Maintains higher end-to-end throughput

This local recovery is much faster than end-to-end TCP recovery, especially over long network paths.

17.2.3 Medium Access Control (MAC)

- Manages access to shared physical medium (e.g., wireless spectrum)
- Detects collisions when multiple devices transmit simultaneously
- Implements backoff and retry mechanisms

17.3 Ethernet Networks

Now let's examine how packets actually move within an IP subnet, focusing on Ethernet—the dominant technology for local area networks.

17.3.1 MAC Addresses

Every network interface in an Ethernet subnet has a unique **MAC address** (also called Ethernet address or physical address):

- **Format:** 48-bit number, typically written as six hexadecimal bytes
- **Example:** `5c:f9:38:a4:00:76`
- **Addressing:** Flat (not hierarchical like IP addresses)
- **Scope:** Globally unique but location-independent

Intra-Subnet Communication

When devices communicate within the same IP subnet, packets carry Ethernet headers containing source and destination MAC addresses:

Link-layer header	Network header	Data
src MAC — dst MAC	IP header	Payload

For communication between two devices in the same subnet:

- **Source MAC:** Address of the sending device's network interface
- **Destination MAC:** Address of the receiving device's network interface

Inter-Subnet Communication

When a device sends packets to a different IP subnet, the MAC addressing changes as the packet traverses the local subnet:

- **Source MAC:** Sending device's MAC address
- **Destination MAC:** Border router's MAC address (not the final destination!)

Key principle: Within any IP subnet, packets always carry MAC addresses from devices *within that subnet*. The source MAC belongs to whichever device first forwards the packet in this subnet, while the destination MAC belongs to whichever device will receive the packet last in this subnet. This means MAC addresses change as packets cross subnet boundaries, while IP addresses remain constant end-to-end.

17.3.2 Switch Forwarding

Ethernet switches use **forwarding tables** to decide where to send packets. Each switch:

1. Names its network interfaces (called *links* or *ports*)
2. Maintains a forwarding table mapping MAC addresses to output links
3. For each incoming packet: reads destination MAC address, looks it up, and forwards to the correct output link

L2 vs L3 Forwarding: A Critical Difference

The flat nature of MAC addresses creates a fundamental difference from IP forwarding:

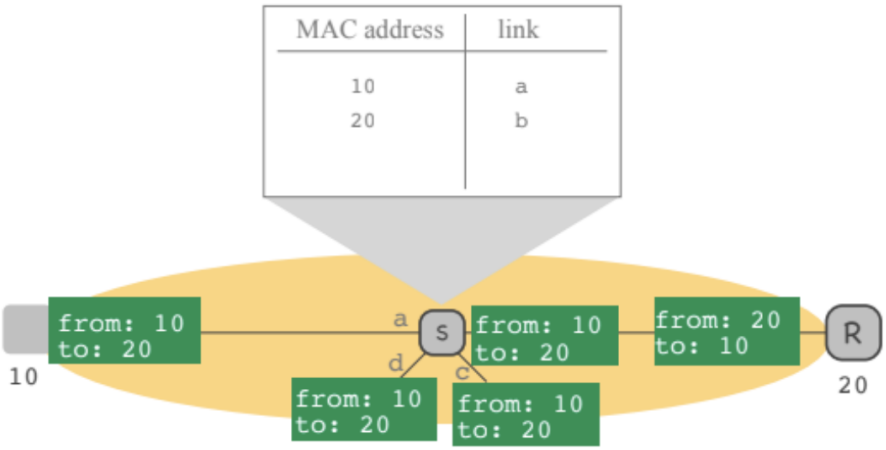
L2 Forwarding (Flat addresses): Cannot group MAC addresses into prefixes; forwarding table size equals the number of active MAC addresses in the subnet

L3 Forwarding (Hierarchical addresses): Groups IP addresses into prefixes; forwarding table size equals local prefixes plus aggregated foreign prefixes

This means Ethernet switches must track every individual device, while IP routers can aggregate millions of addresses into single routing entries.

17.3.3 L2 Learning: Self-Configuring Networks

Unlike IP routers that exchange explicit routing information, Ethernet switches learn automatically from traffic:



Learning Algorithm

1. **Initial state:** Forwarding table is empty
2. **Learning:** When a packet with source MAC address X arrives at link Y, add "MAC X → link Y" to the forwarding table
3. **Unknown destinations:** When a packet arrives with unknown destination MAC, broadcast it to all links

The figure above illustrates this process: when the switch receives a packet from MAC address 10 on link a, it learns that MAC 10 is reachable via link a. Similarly, when it receives traffic from MAC 20 on link b, it updates its forwarding table accordingly.

Learning vs Routing Comparison

L2 Learning: Passive learning from actual traffic; no explicit control messages

IP Routing: Active exchange of routing protocol messages between routers

The Broadcasting Problem: Forwarding Loops

Broadcasting unknown destinations creates a serious problem: **forwarding loops**. If switches naively broadcast to all links, packets can circulate forever in network loops.
Spanning Tree Solution:
To prevent loops, switches use the **Spanning Tree Protocol**:

- Creates a loop-free subgraph connecting all devices
- Includes all nodes but only a subset of links
- Broadcasts propagate only along tree edges
- Eliminates forwarding loops while maintaining connectivity

A spanning tree includes just enough links to reach every device without creating cycles—you cannot remove any edge without disconnecting some node.

17.3.4 Address Resolution Protocol (ARP)

We've explained how switches forward packets using MAC addresses, but a crucial question remains: **How does a device determine the MAC address of its intended recipient?** This is where the Address Resolution Protocol (ARP) comes in.

The Problem: IP to MAC Address Mapping

When Alice wants to send a packet, she faces two requirements:

1. **Destination IP address:** Obtained from DNS (e.g., Bob's IP address)
2. **Destination MAC address:** Unknown—this is what ARP solves

Scenario 1: Same Subnet Communication

When Alice wants to send a packet to Bob in the same IP subnet:

1. **ARP Request:** Alice broadcasts a request asking "Who has IP address 128.178.2.20? Tell 128.178.2.10 (Alice's IP)"
2. **Broadcasting:** Request uses special broadcast MAC address `FF-FF-FF-FF-FF-FF`, reaching every device in the subnet
3. **ARP Response:** Bob recognizes his IP address and responds directly to Alice with his MAC address
4. **Communication:** Alice now knows Bob's MAC address and can send packets directly

Scenario 2: Different Subnet Communication

When Alice wants to send a packet to Bob in a different IP subnet:

1. **Default Gateway:** Alice knows her router's IP address (e.g., 128.178.2.1) through configuration
2. **ARP Request:** Alice broadcasts asking for the router's MAC address
3. **Router Response:** Router responds with its MAC address
4. **Packet Forwarding:** Alice sends packets to Bob using the router's MAC address as destination

ARP vs DNS Comparison

ARP: Uses broadcasting within local subnet; no centralized infrastructure; each device knows its own MAC address

DNS: Uses hierarchical server infrastructure; logically centralized mapping; dedicated servers maintain databases

Both serve address resolution roles but operate at different scales and use different mechanisms.

17.4 Design Trade-offs and Architecture

17.4.1 Could We Eliminate IP Addresses?

An interesting question: Could the entire Internet be one big Ethernet subnet using only MAC addresses?
Answer: No, this wouldn't scale.

- **Forwarding table explosion:** Switches would need individual entries for millions of active devices
- **Broadcasting chaos:** Every address resolution would broadcast to the entire Internet
- **No aggregation:** Flat MAC addresses prevent the hierarchical aggregation that makes IP routing scalable

17.4.2 Could We Eliminate MAC Addresses?

Alternatively: Could we use only IP addresses and routers everywhere?
Answer: Yes, this would scale, but we'd lose flexibility.
The Internet was designed as a *network of networks*—different types of subnets (Ethernet, WiFi, fiber, etc.) interconnected by IP routers. Eliminating the link layer would require replacing all specialized network equipment with IP routers, reducing the diversity and adaptability that make the Internet robust.

17.4.3 Ethernet Elements Summary

The complete Ethernet system relies on three core components:

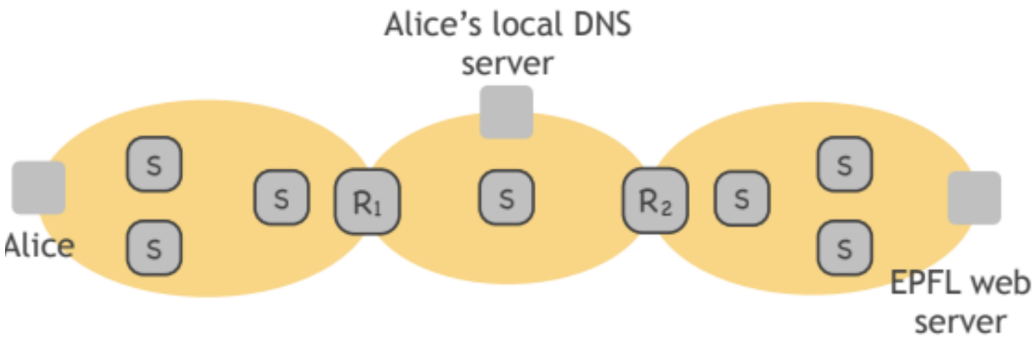
Address Resolution Protocol: Maps IP addresses to MAC addresses (analogous to DNS)

L2 Forwarding: Routes packets based on flat MAC addresses (analogous to IP forwarding)

L2 Learning: Populates switch forwarding tables (analogous to IP routing protocols)

17.5 Complete Example: Packet Journey

Now let's trace through a complete example showing how all these pieces work together when Alice sends a DNS request to access a web server:



17.5.1 The Scenario

When Alice types `http://www.epfl.ch` in her browser, this generates at least four packets:

1. Alice's DNS request to local DNS server
2. Local DNS server's response to Alice
3. Alice's HTTP GET request to web server
4. Web server's response to Alice

We'll focus on the first packet: Alice's DNS request to her local DNS server.

17.5.2 Step-by-Step Packet Journey

Step 1: Application Layer Creates DNS Request

Alice's DNS client process creates a DNS request to resolve `www.epfl.ch`, which is passed down to the transport and network layers:

- **Source IP:** Alice's IP address
- **Destination IP:** Local DNS server's IP address (known via configuration)

Step 2: ARP Request for Local Router

Alice's network layer needs to determine the appropriate destination MAC address. Since the DNS server is in a different subnet, Alice must send the packet to her default gateway (router R1). She broadcasts an ARP request to resolve the router's IP address to its MAC address.

Step 3: ARP Response from Router

Router R1 receives the ARP request, recognizes its own IP address, and responds with its MAC address.

Step 4: DNS Request with MAC Headers

Alice can now send the DNS request with proper addressing:

- **Source MAC:** Alice's MAC address
- **Destination MAC:** R1's MAC address
- **Source IP:** Alice's IP address
- **Destination IP:** DNS server's IP address

Step 5: Router Performs IP Forwarding

R1 receives the packet, examines the destination IP address, and consults its forwarding table to determine the next hop toward the DNS server.

Step 6: Router's ARP Request

R1 needs to forward the packet to the next subnet containing the DNS server. It broadcasts an ARP request to resolve the DNS server's IP address to its MAC address.

Step 7: DNS Server's ARP Response

The DNS server receives the ARP request and responds with its MAC address.

Step 8: Final Packet Delivery

R1 forwards the DNS request with updated MAC addresses:

- **Source MAC:** R1's MAC address (in the new subnet)
- **Destination MAC:** DNS server's MAC address
- **Source IP:** Alice's IP address (unchanged)
- **Destination IP:** DNS server's IP address (unchanged)

Address Requirements

- **End-systems and routers need MAC addresses:** Otherwise, switches wouldn't know where to forward packets within subnets
- **End-systems need IP addresses:** Otherwise, routers wouldn't know where to forward packets across the Internet
- **Switches don't need MAC addresses for forwarding:** They learn MAC-to-port mappings automatically
- **Routers don't need IP addresses for forwarding:** They use destination IP addresses in packet headers

17.6 Network Hierarchy Summary

The Internet operates at three distinct levels, each with its own addressing and routing mechanisms:

17.6.1 Level 1: IP Subnets

- **Devices:** Link-layer switches connecting end-systems and routers
- **Forwarding:** L2 forwarding based on MAC addresses
- **Learning:** L2 learning populates switch forwarding tables automatically

17.6.2 Level 2: Autonomous Systems (AS)

- **Devices:** IP routers connecting multiple IP subnets within one administrative domain
- **Forwarding:** IP (L3) forwarding based on IP addresses
- **Routing:** Intra-domain routing protocols (OSPF, RIP) populate router forwarding tables

17.6.3 Level 3: Internet

- **Devices:** Border routers connecting multiple Autonomous Systems
- **Forwarding:** IP (L3) forwarding based on IP addresses
- **Routing:** Inter-domain routing protocol (BGP) coordinates between ASes

Each level uses appropriate addressing schemes and protocols optimized for its scale and administrative requirements, creating a hierarchical system that enables the Internet's remarkable scalability and robustness.