# 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

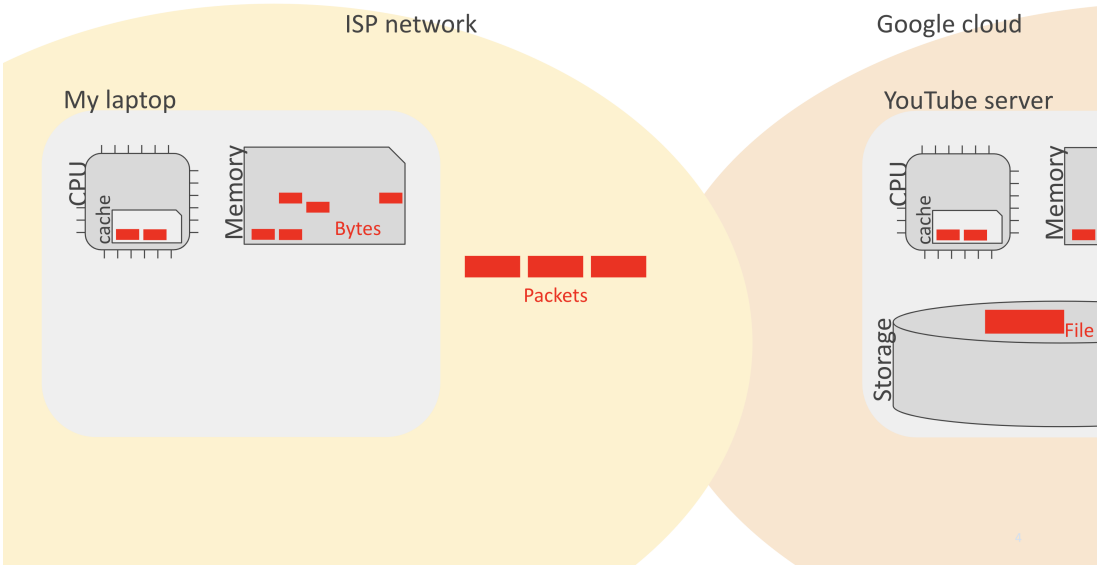
# Chapter 1

# 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

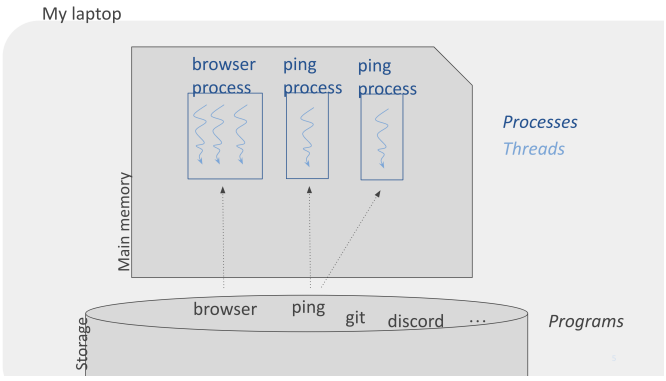
To illustrate these ideas, consider the journey of a YouTube video. The video begins its existence as a file stored on a storage device, is loaded into memory as bytes, transmitted as packets over the Internet, and finally rendered on your screen.



### 1.1.1 Start of the Journey: Inside the Laptop

The journey begins on your laptop.

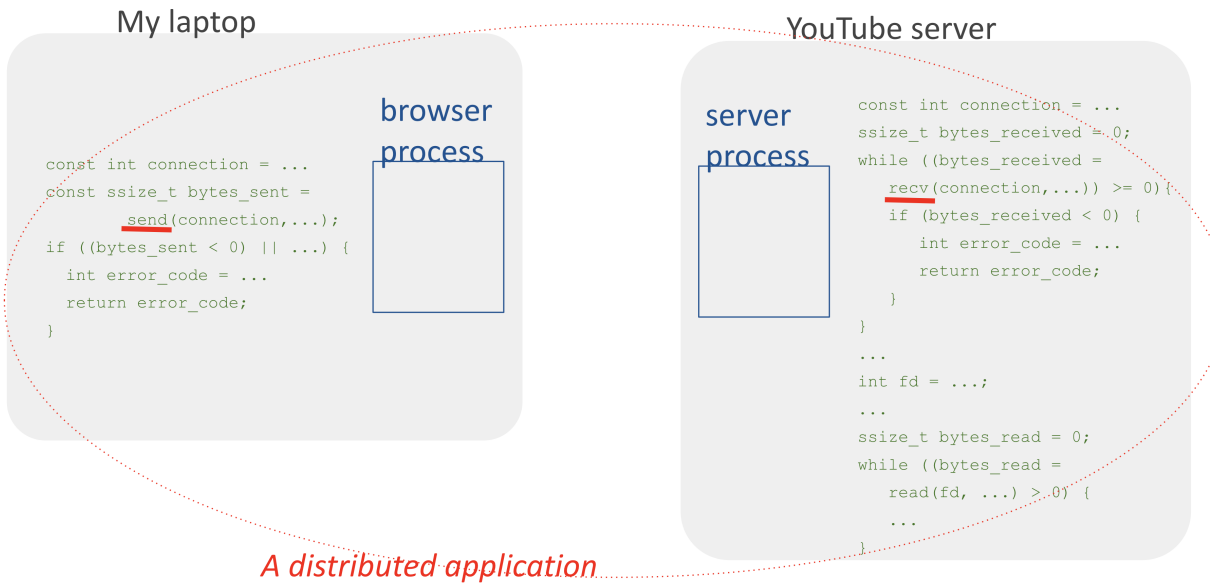
A computer hosts many different programs (e.g., a web browser, a ping utility, a git client). These programs, stored as files on disk, are invoked by user actions such as clicking an icon or typing a command. When a program is invoked, the computer creates a new *process* in main memory. A process represents 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*.

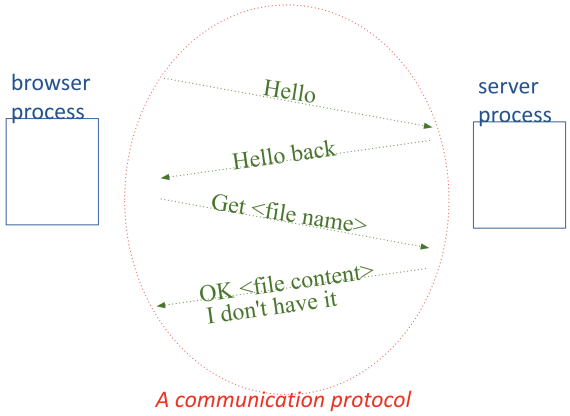


### 1.1.3 Communication Protocols

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:

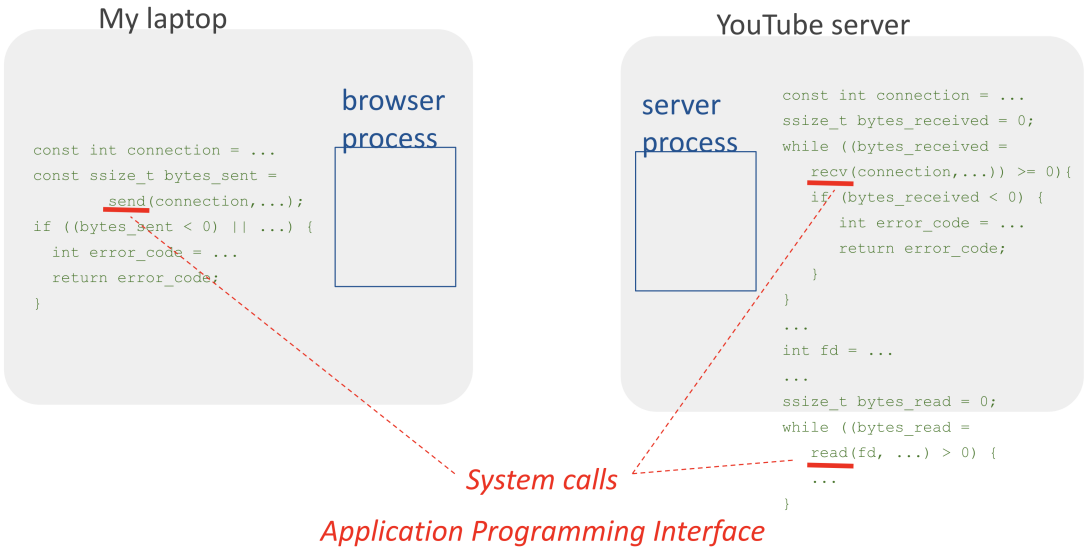
- One process sending "hello" and waiting for a "hello back."
- A subsequent request for a specific file (e.g., `xyz`) with the server responding with the file or an error message.

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




### 1.1.4 Distributed Applications and APIs

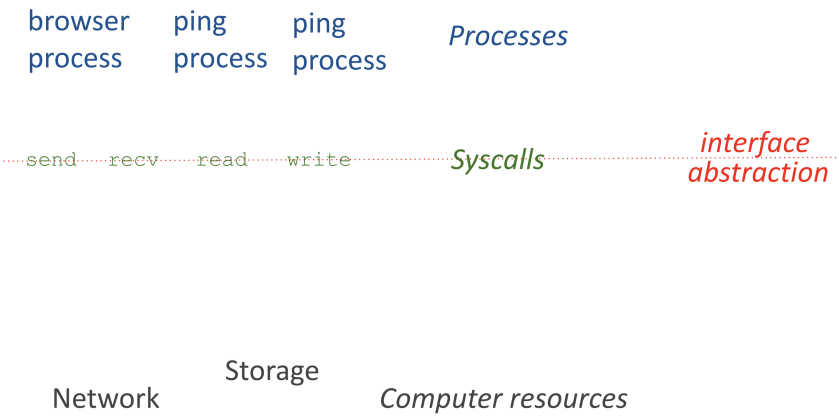
Distributed applications consist of separate pieces of code running as processes on different machines but working toward a common goal. These processes exchange messages over the Internet by following communication protocols. To simplify the development of these applications, developers use *system 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.*

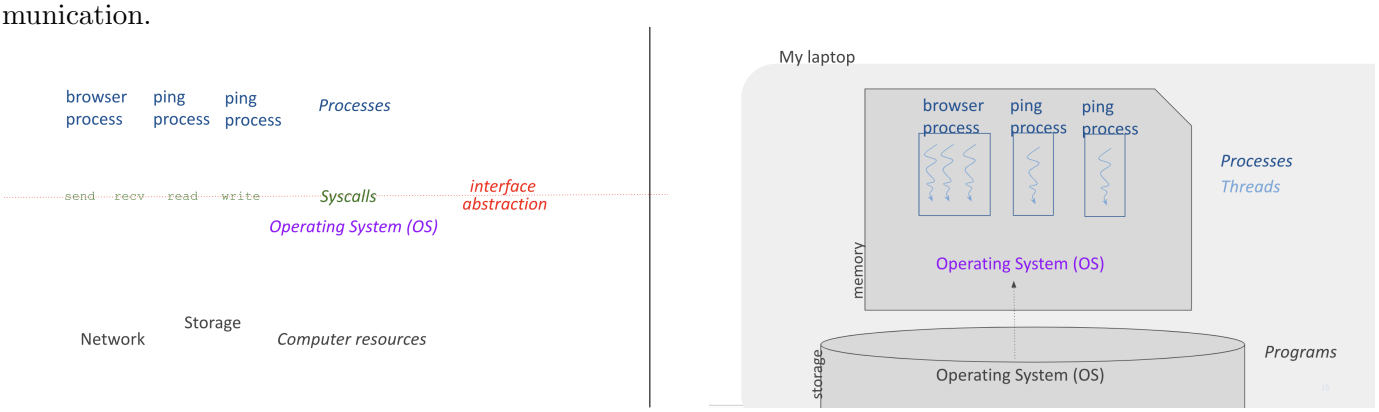

### 1.1.5 System Calls (Syscalls)

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

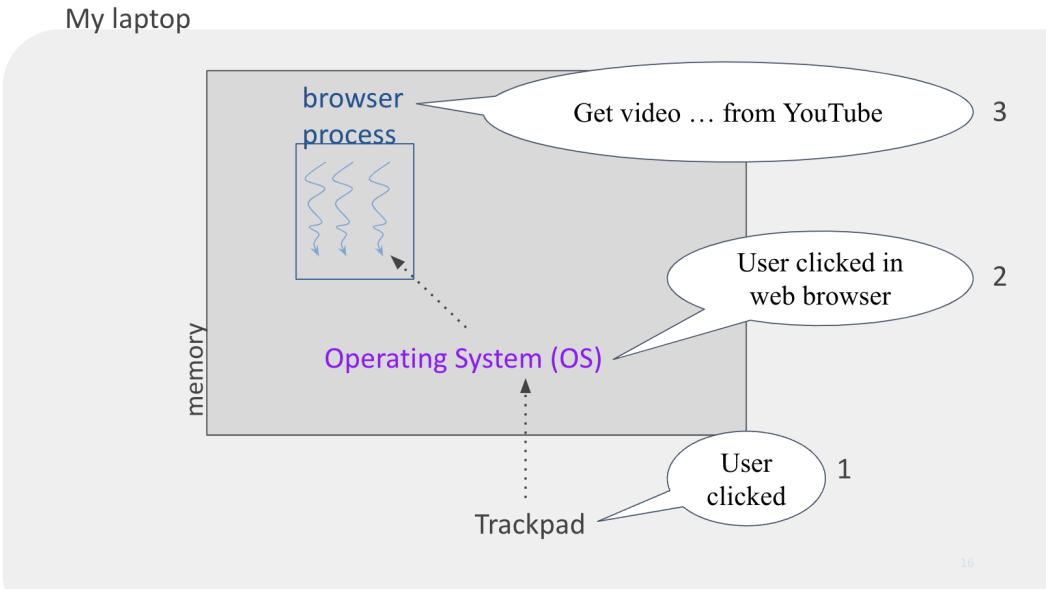
Conceptually, the OS sits between running processes and the underlying hardware resources. It provides the syscall interface and handles tasks such as file system management and network communication.




### 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.

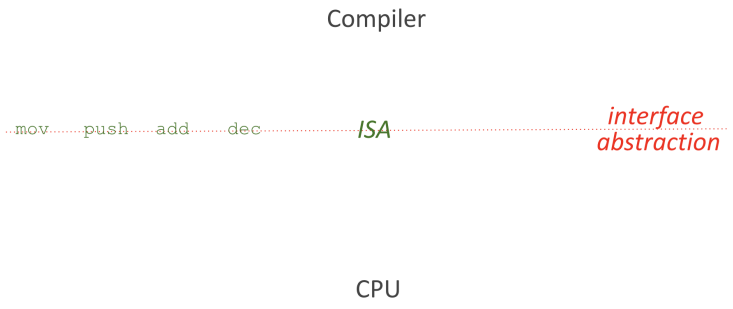


## 1.3 Program and ISA

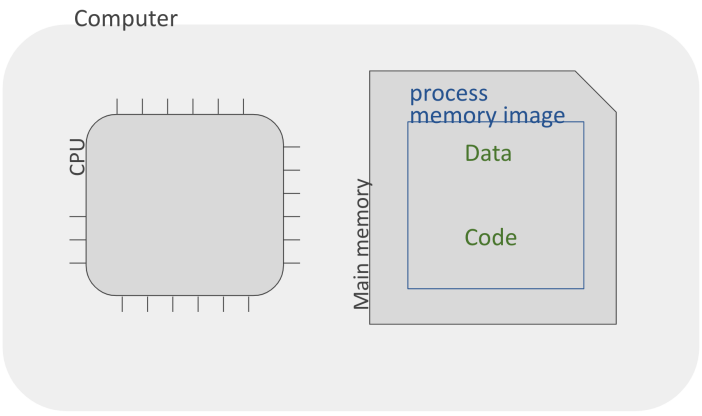
### 1.3.1 Program

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.*

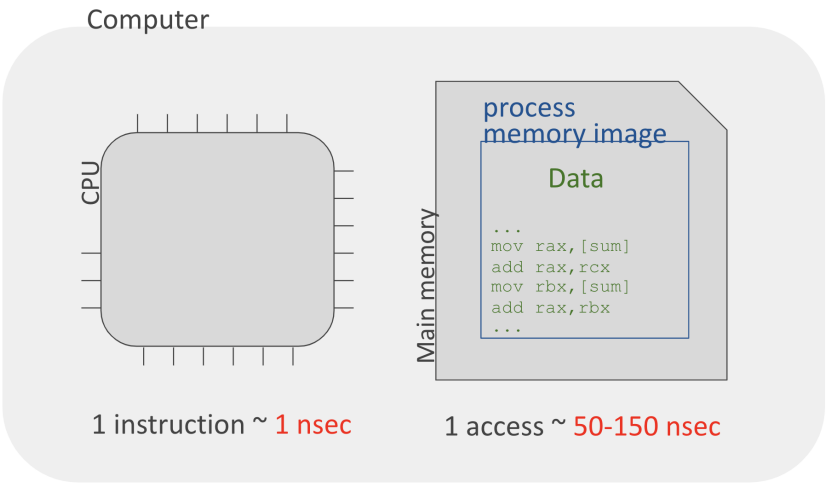
***Question:*** *What is the meaning of the Hz metric?*

***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.

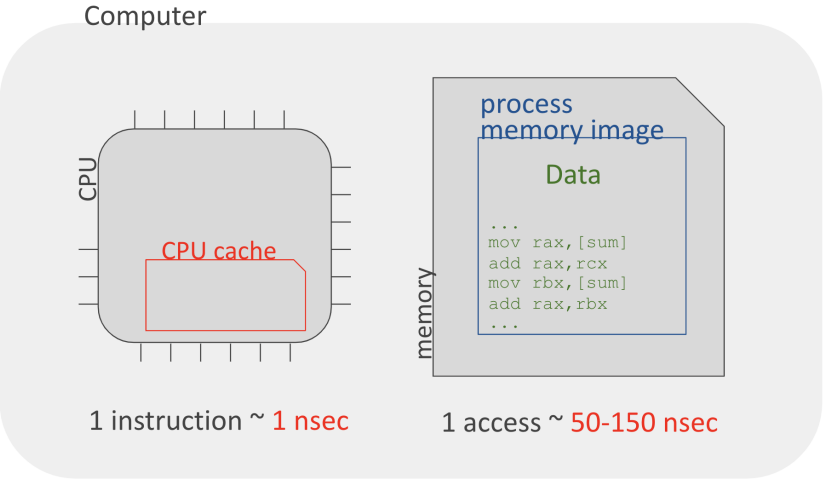



**Question:** How can we improve the efficiency of a system with such a frequency imbalance?

**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.

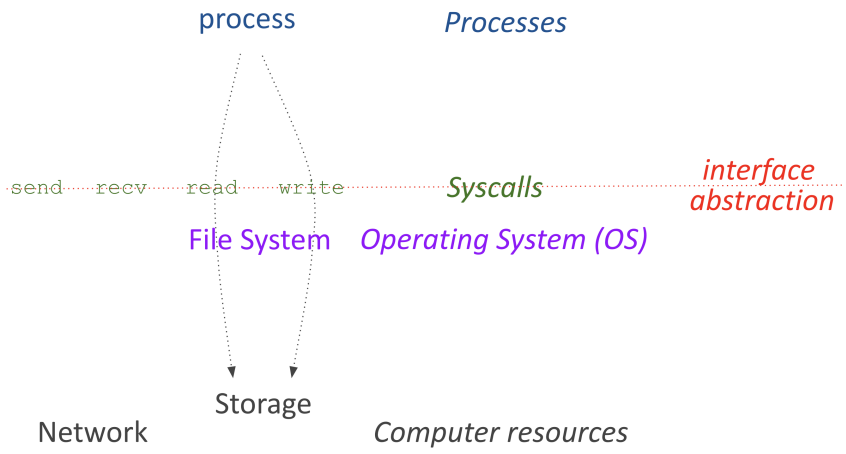
*The browser C code* → Compiler → *The browser executable*

```
...
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.*

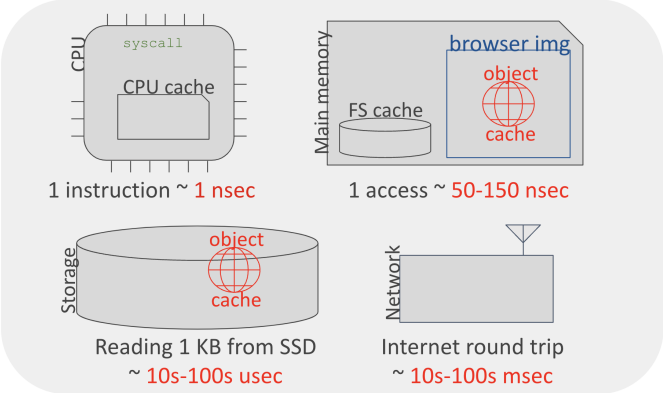
**Exam Question:** If a program does not create or manipulate any data, will executing this program require reading anything from memory?

**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:

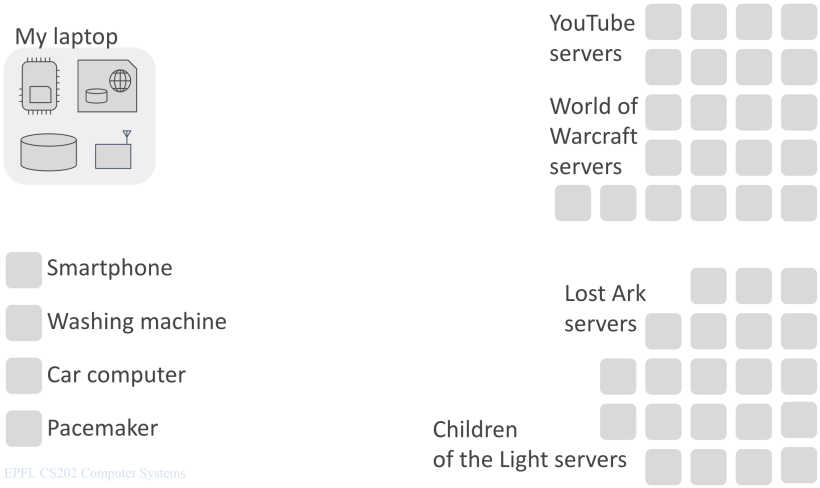
- Simple CPU instructions: ∼1 nsec.
- Main memory accesses: tens to hundreds of nanoseconds.
- Reading 1 KB from an SSD: tens to hundreds of microseconds.
- Requesting data over the Internet: several to hundreds of milliseconds.




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.

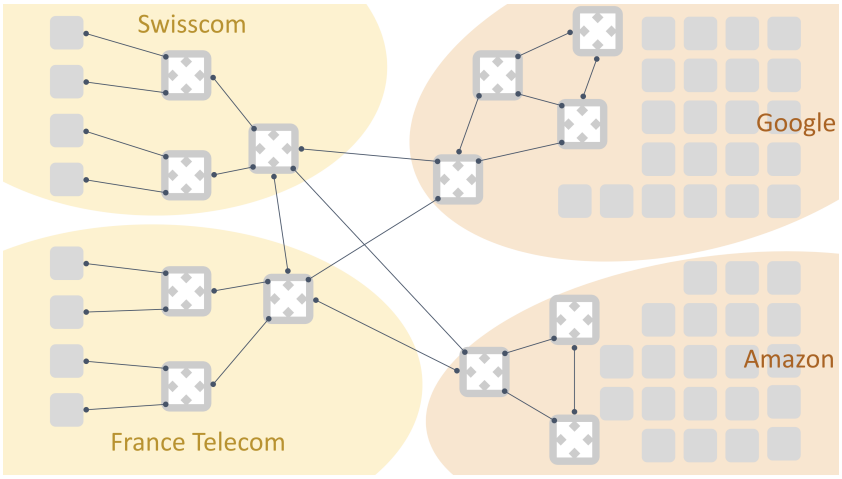


### 1.6.2 Packet Switches and Network Links

In addition to end systems, the Internet relies on:

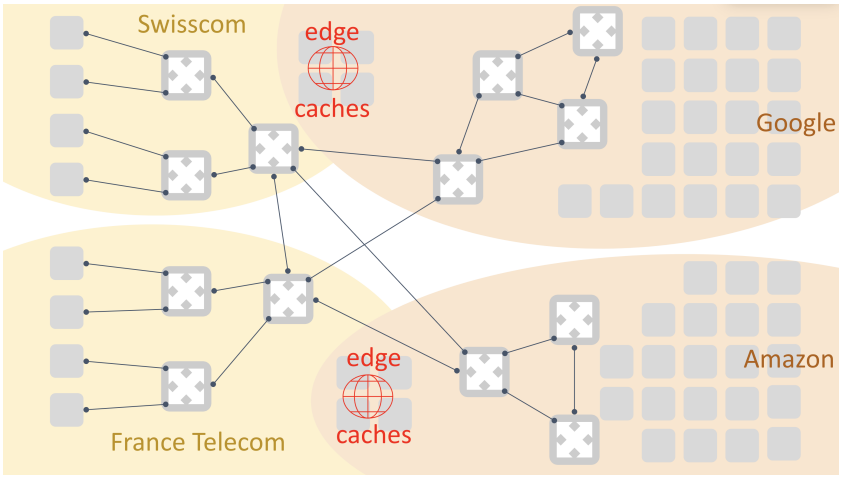
- **Packet Switches**: Devices that route data between end systems.
- **Network Links**: Physical connections that interconnect packet switches and end systems.

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.

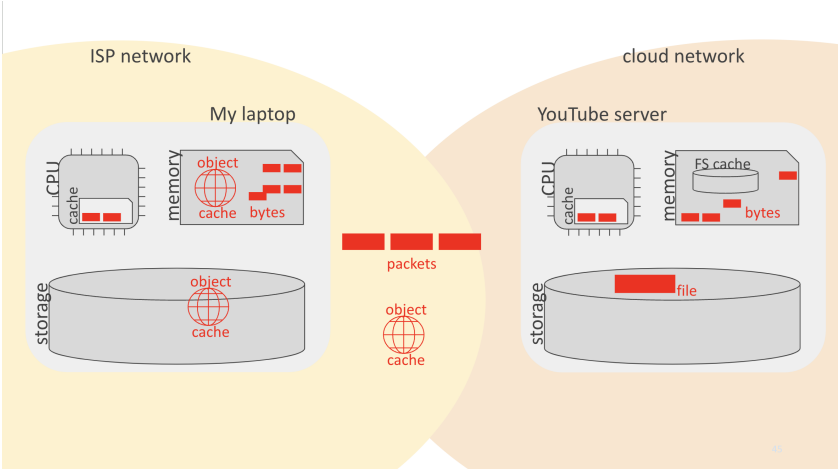



## 1.7 Summary

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).



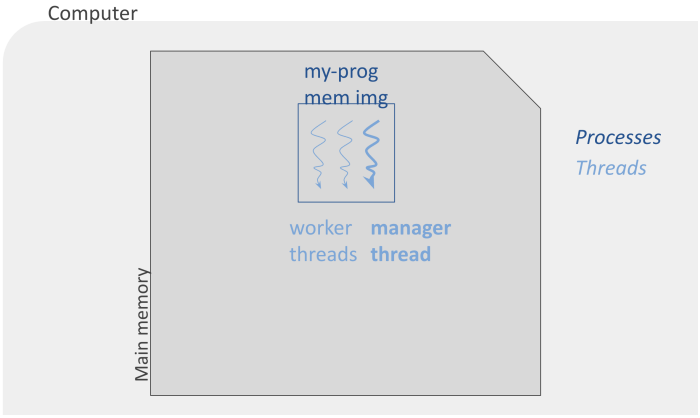
# Chapter 2

# 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

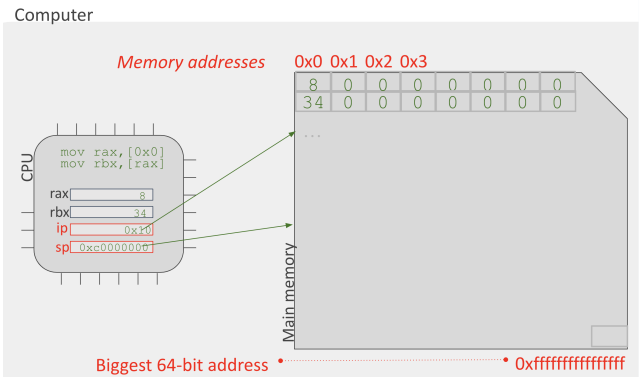
When a program runs, the processor loads it into main memory and creates a thread. In a multithreaded program, there is typically one *manager* thread that delegates work to several *worker* threads. For instance, when computing the sum of a large set of numbers, the workload can be divided into subsets,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

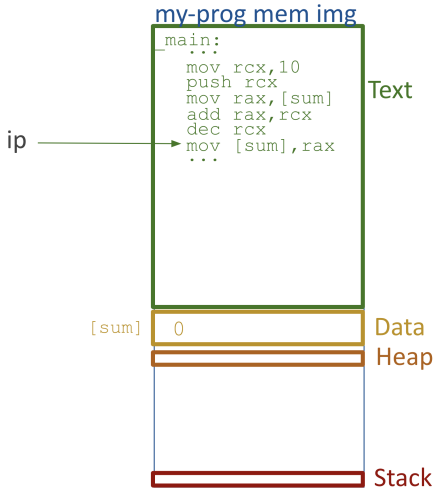
### 2.3.1 The Stack

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.

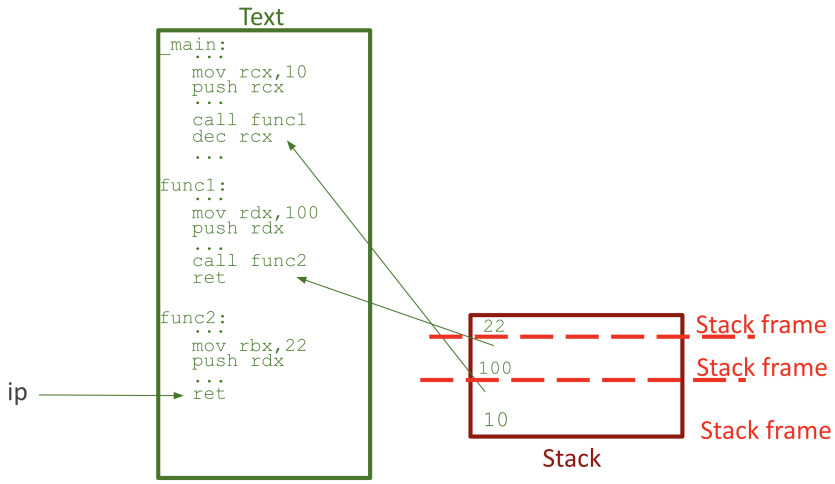
**Key Characteristics of the Stack:**

- **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.*

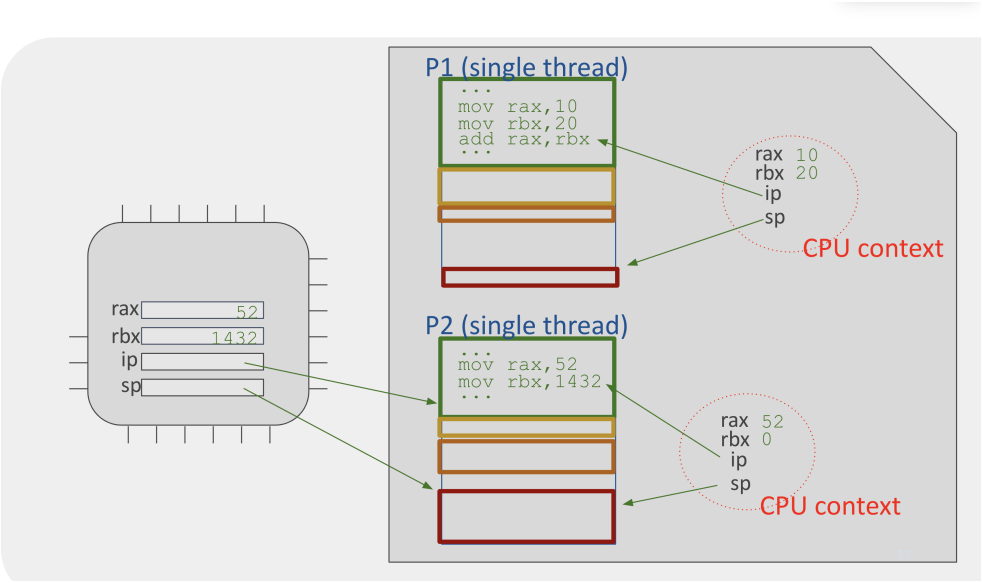
- ***Process ID (PID):*** *A unique identifier assigned to each process.*
- ***Thread ID (TID):*** *A unique identifier for each thread, which may be unique within a process or across the entire system, depending on the operating system.*

**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

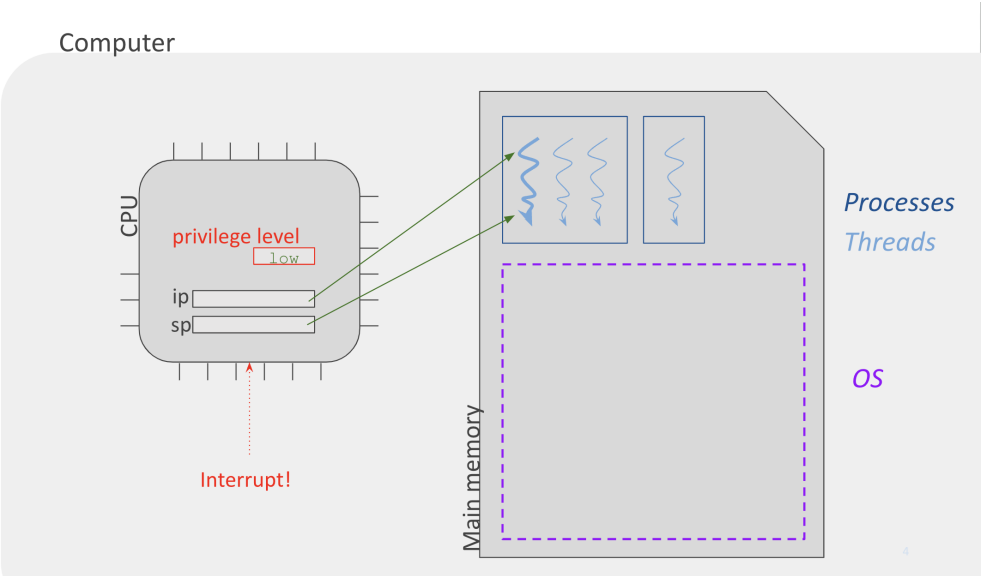
### 3.1.2 CPU Privilege Levels and Execution Modes

**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.

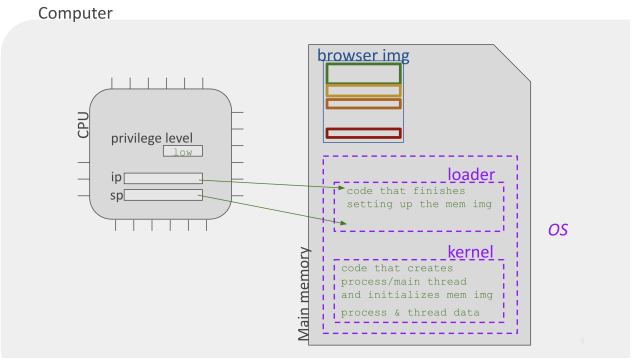
- By switching between kernel mode and user mode, the OS ensures that user applications run securely and do not directly manipulate hardware resources.

#### 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.

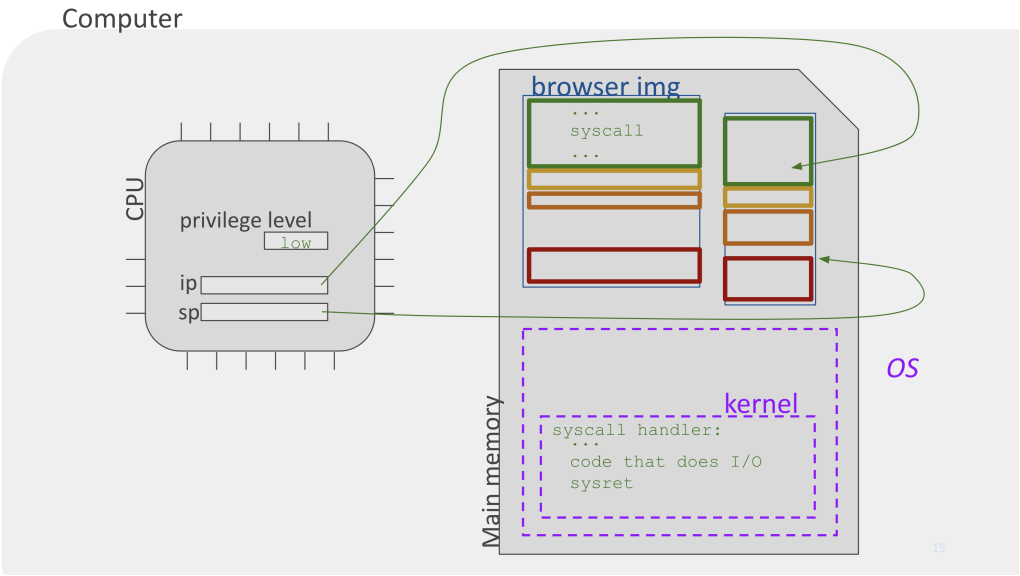



This layered privilege model ensures a secure and efficient execution environment, maintaining stability and protection across all processes within the system.

### 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:

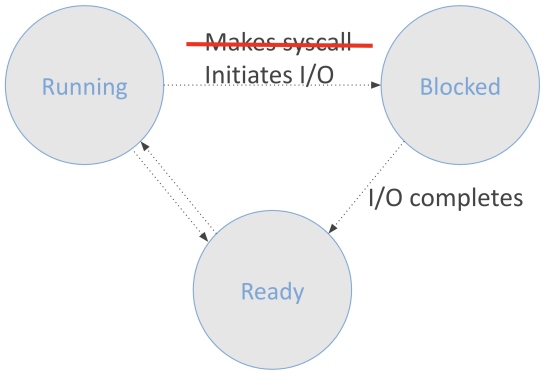
1. The CPU temporarily elevates the privilege level.
2. The control is transferred to the OS to execute the privileged code.
3. If the syscall involves an I/O operation, the process may be blocked while waiting for a response, and the CPU may perform a context switch to another thread.
4. Once the operation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



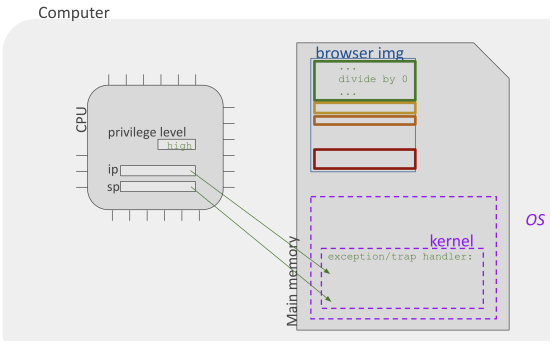

## 3.2 The Kernel's Job cont.

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.

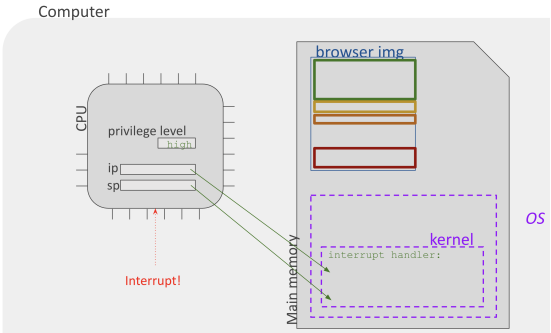
#### Exception Handling

*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.

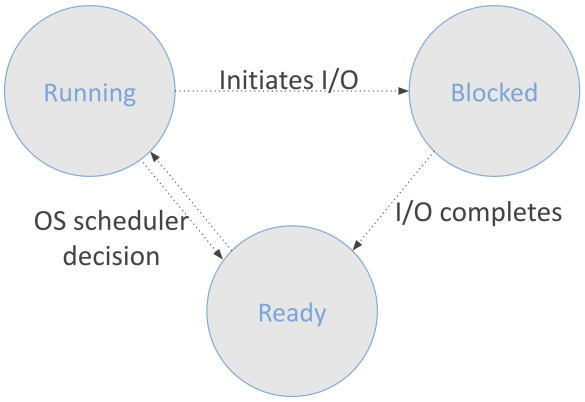



#### The Timer Interrupt

**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 .
```

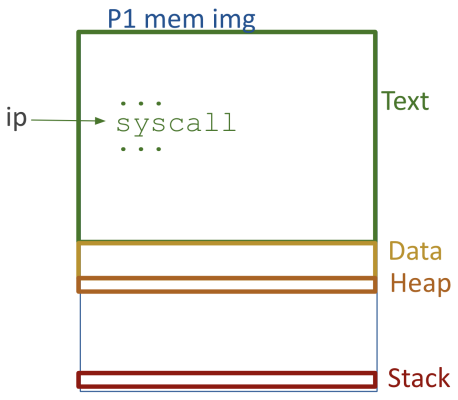



Figure 3.1: Exit Syscall: Code snippet and stack visualization.

**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);
```

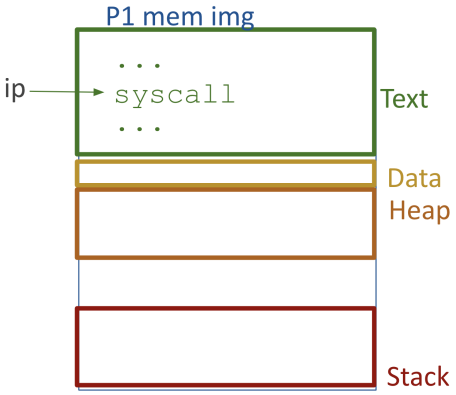



Figure 3.2: Exec Syscall: Code snippet and process image.

**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
}
```

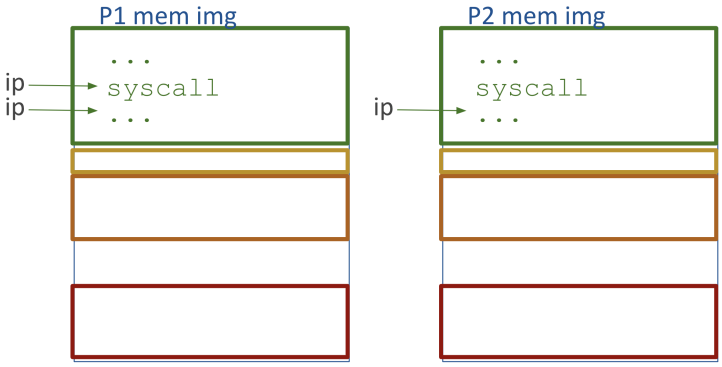



Figure 3.3: Fork Syscall: Example code and the resulting stack layout.

**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

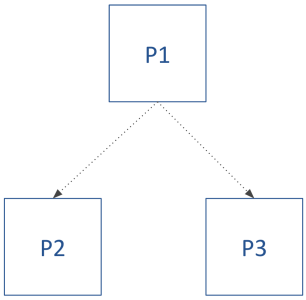
Processes are typically created by combining the `fork` and `exec` syscalls. For example:

```
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

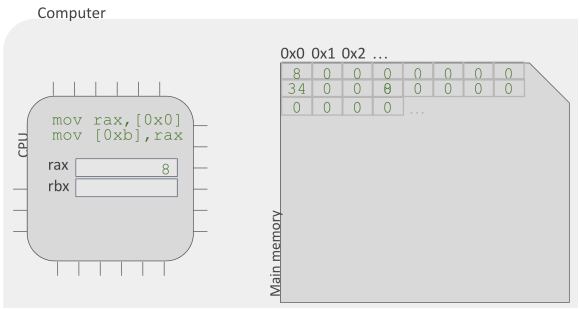
Main memory is conceptualized as a linear array of bytes, where each byte has a unique memory address (e.g., `0x0`, `0x1`, `0x2`, etc.). Each byte can store any value that fits within its 8-bit capacity, and importantly, the value stored in a given byte is independent of its memory address. For instance,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

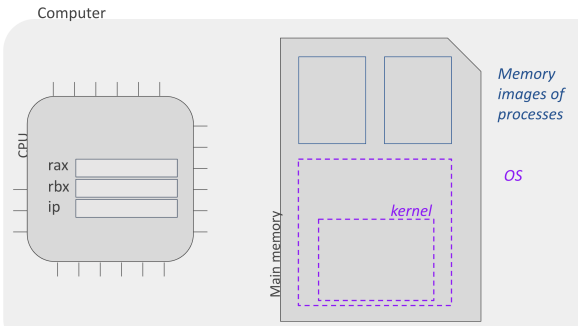
*The **instruction pointer** is a CPU register that holds the memory address of the next instruction to be executed.*

### 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).**

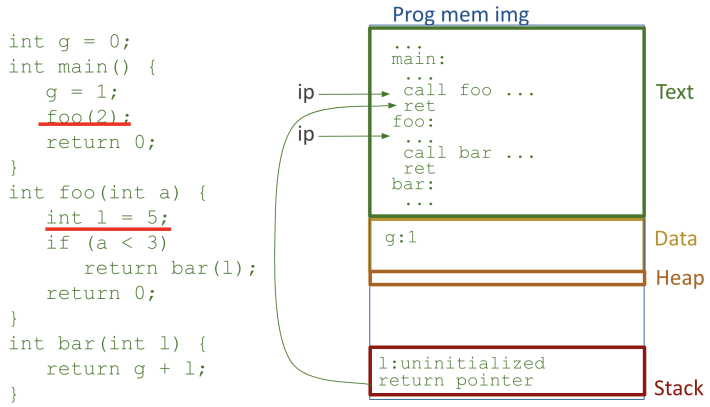
*A **process memory image** is the complete layout of a process's memory, comprising:*

- *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.

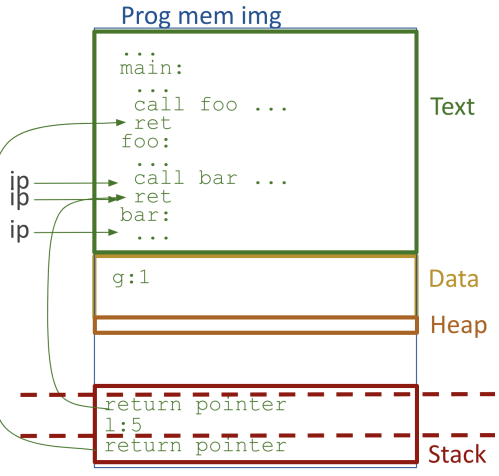
1. Mark each segment, even if it is empty.
2. Draw a schema of the code in the **Text Segment**, including only function names and calls in assembly.
3. Identify global variables and place them in the **Data Segment** (e.g., `g:0`).
4. Simulate each step of the program's execution to fill the **Heap**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




**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 + l;
}
```



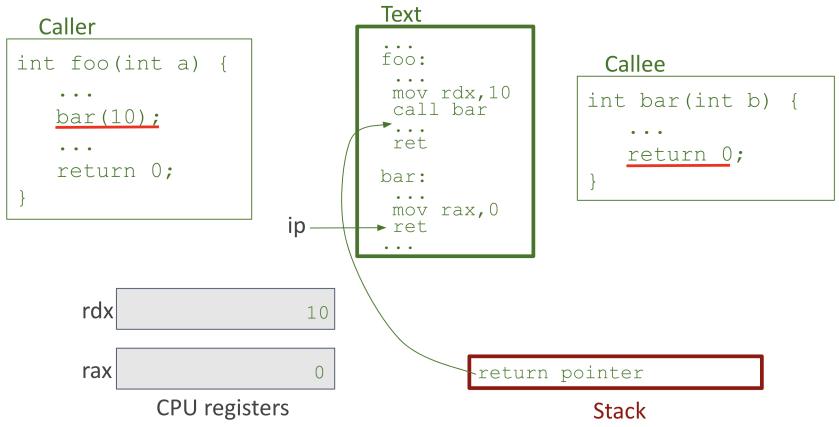
### 4.2.1 Optional - Stack and Register Functioning

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:

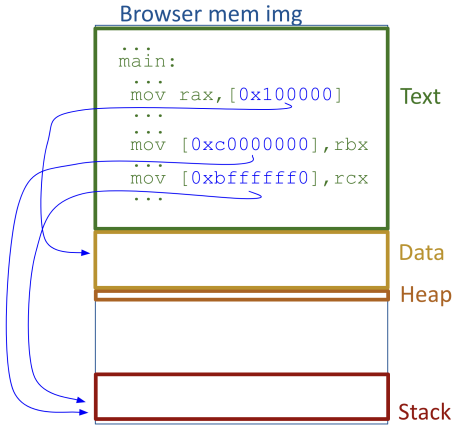
- **Caller-Saved Registers:** The caller saves certain registers to the stack before the call and restores them after the call returns.
- **Callee-Saved Registers:** The callee saves its registers at the beginning of the function and restores them before returning.

*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

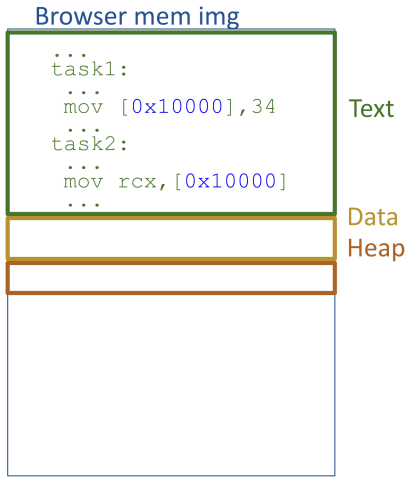
In modern operating systems, each process references memory using virtual addresses. Underneath, these virtual addresses are translated to physical addresses. Importantly, each process has its own virtual address space, which means that two processes may use the same virtual address while referring to 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).



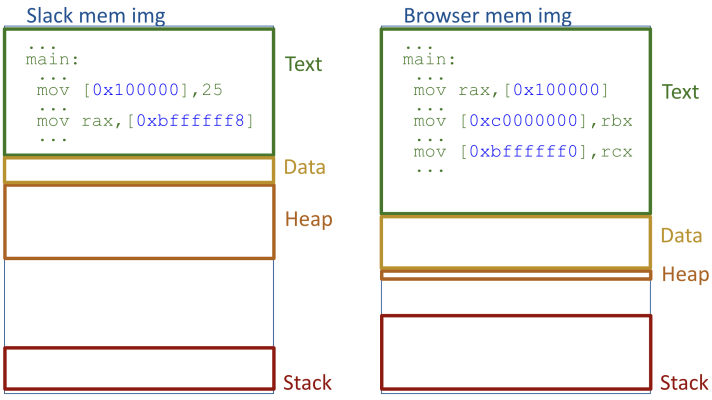

**Exam Question:** If two memory instructions read the same virtual address, is it the same physical address?

**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.

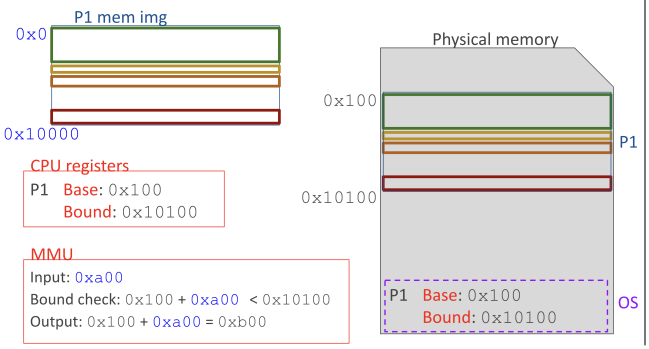
**Definition (Contiguous Memory).**

*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.*

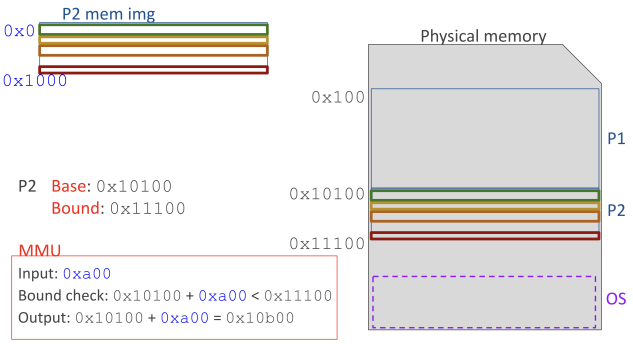
### 4.3.1 Memory Management Unit — Simple Implementation

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.



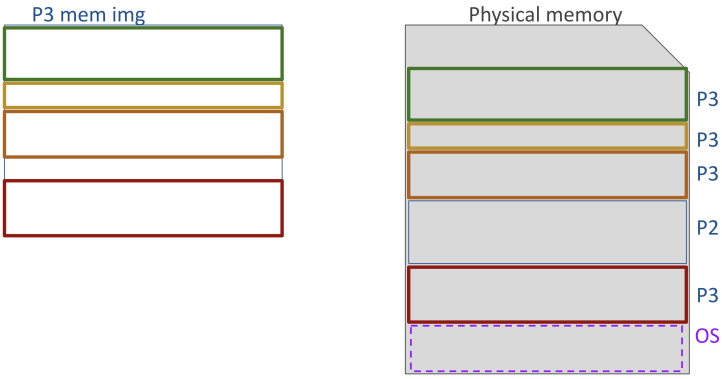

When a second process ($P_2$) is introduced, the MMU checks its corresponding base and bound registers to determine the physical memory range in which $P_2$ should be placed.




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.

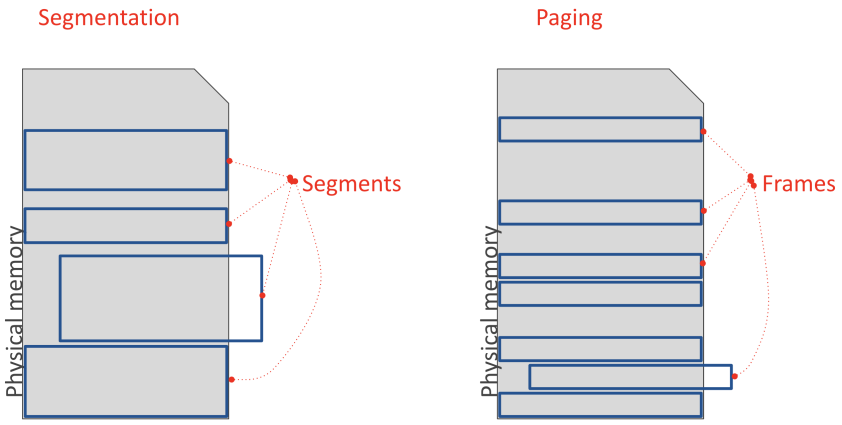
However, because each process must reside in one contiguous memory block, **fragmentation** can occur.

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.

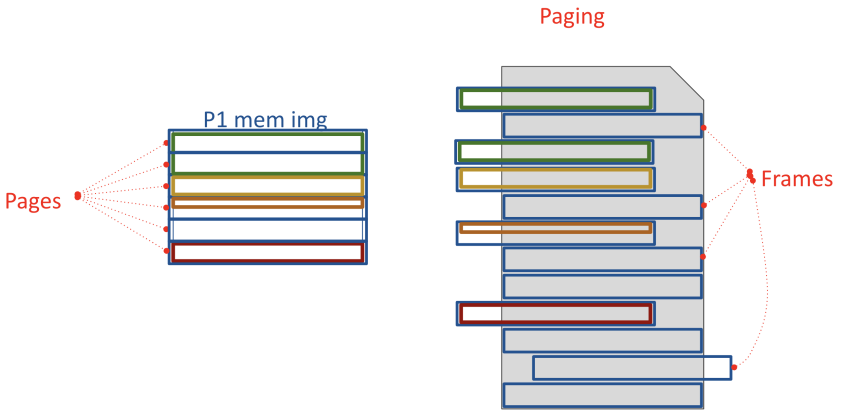


To address the inefficiency of requiring each process to occupy a single contiguous block of memory, an effective strategy is to *divide the process's address space into smaller chunks*, allowing noncontiguous allocation. Two primary techniques for accomplishing this are:

- **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.



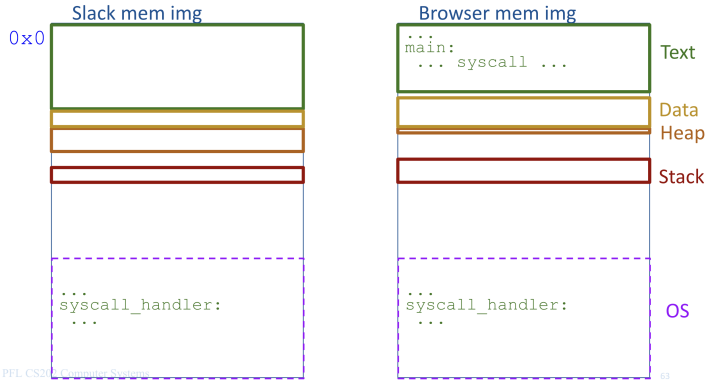

In **paging**, the MMU maintains a *page table* to translate from virtual pages to fixed-size physical frames:




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.

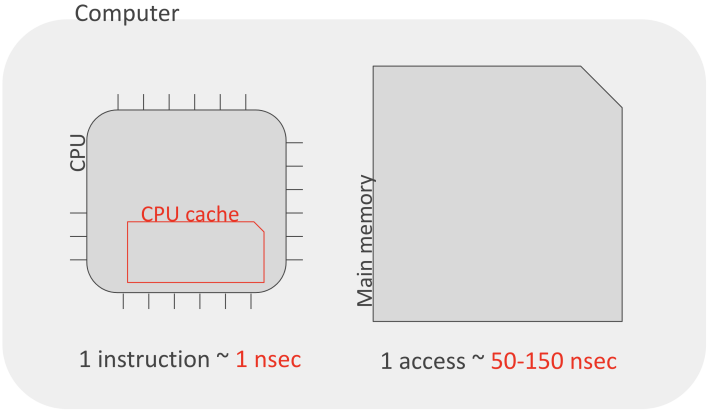



## 4.5 CPU Caching and Memory Hierarchy

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.



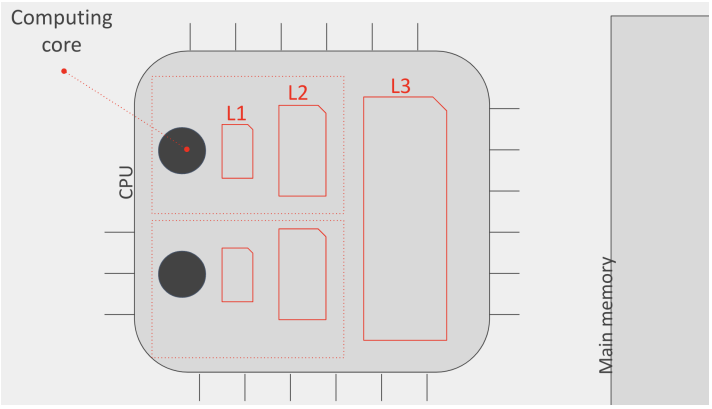

### 4.5.2 Multi-Level Cache Architecture

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.*

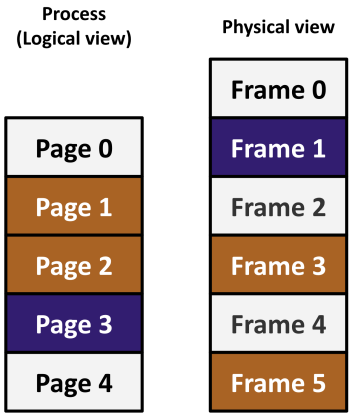
## 5.1 Page-based Memory Management Unit (MMU)

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.

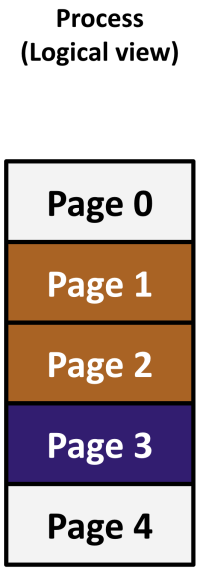


### 5.1.2 Size of a Page

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.

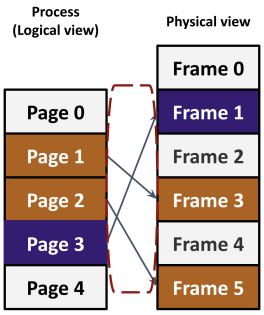
**Super Pages:** These are larger blocks made up of multiple contiguous pages (e.g., 2 MB or 1 GB). They reduce the overhead associated with page translation.




### 5.1.3 Memory Management Scheme

The operating system manages memory using the following scheme:

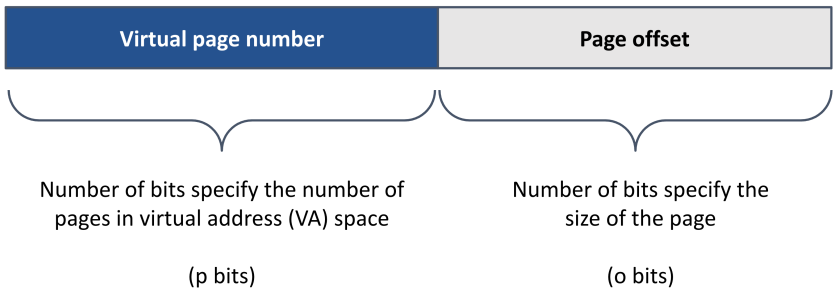
1. **Frame Allocation:** Logical pages are mapped to available physical frames based on the OS's allocation strategy.
2. **Page Table:** The OS maintains a data structure called the page table, which stores the mapping between logical pages and physical frames.
3. **Per-Process Management:** Each process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



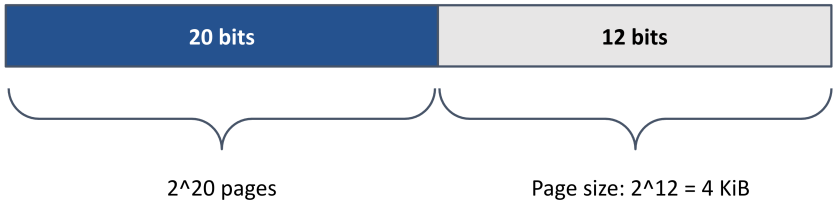

### 5.1.4 Address Representation

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.



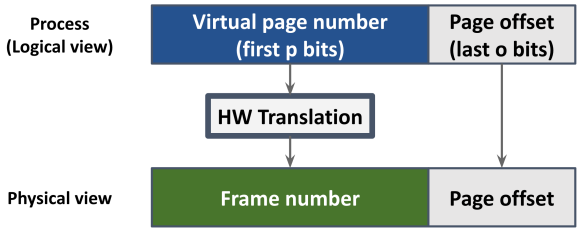
**Example 5.1.4.1** (Virtual Address Example (32-bit Architecture))**.** *In a 32-bit system, the virtual address is divided into a virtual page number and a page offset, as illustrated below.*




### 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.

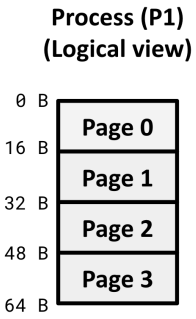
*Personal Remark:* This systematic approach to address translation is fundamental to the operation of virtual memory systems, ensuring efficient and secure memory access.

### 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.*



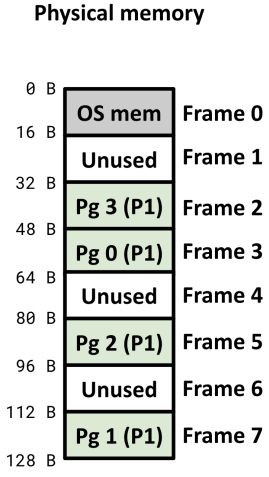

### 5.1.7 Physical Memory

**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*



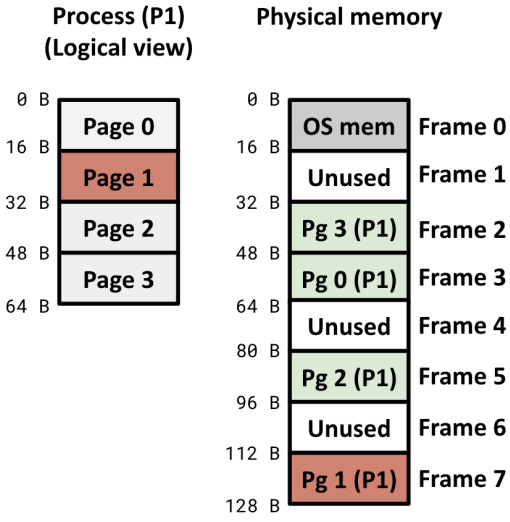

### 5.1.8 Virtual Address Translation

**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:*

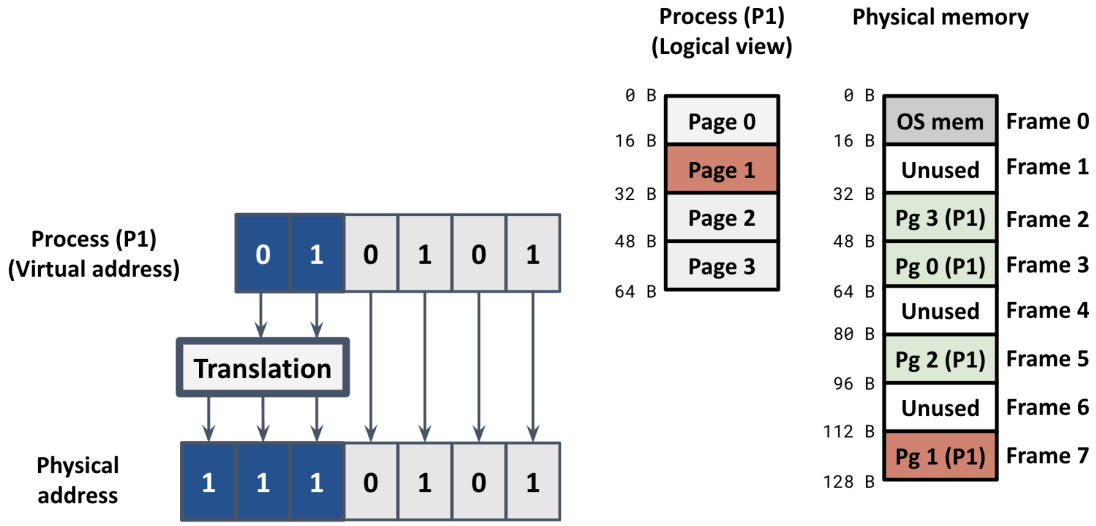
- *Virtual address space: 64 bytes (6 bits)*
- *Page size: 16 bytes (4 bits for offset)*

*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).*

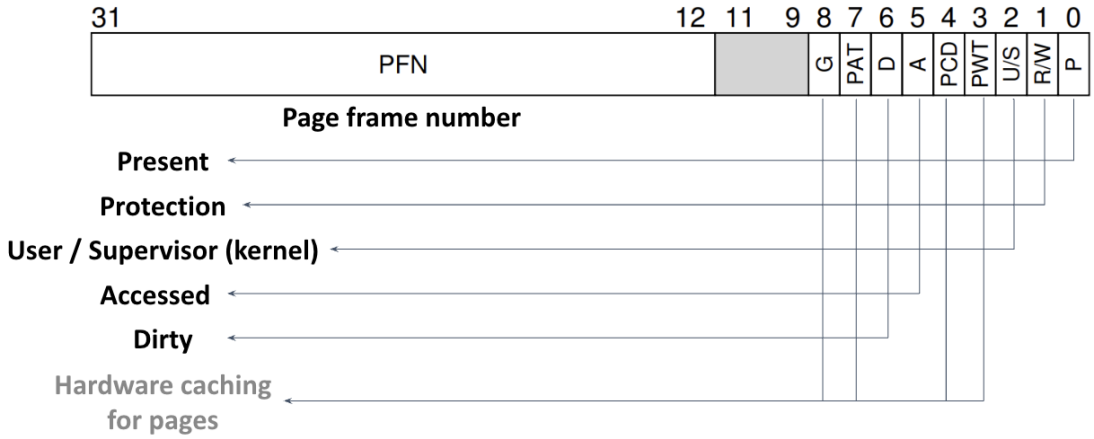
### 5.2.1 Structure of Page Table Entries

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



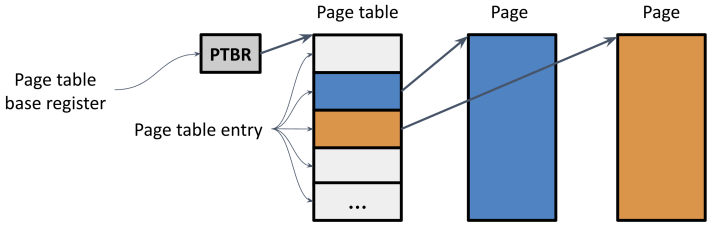

Additionally, each page table entry is always aligned to the page size for efficient access by the hardware.

## 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**

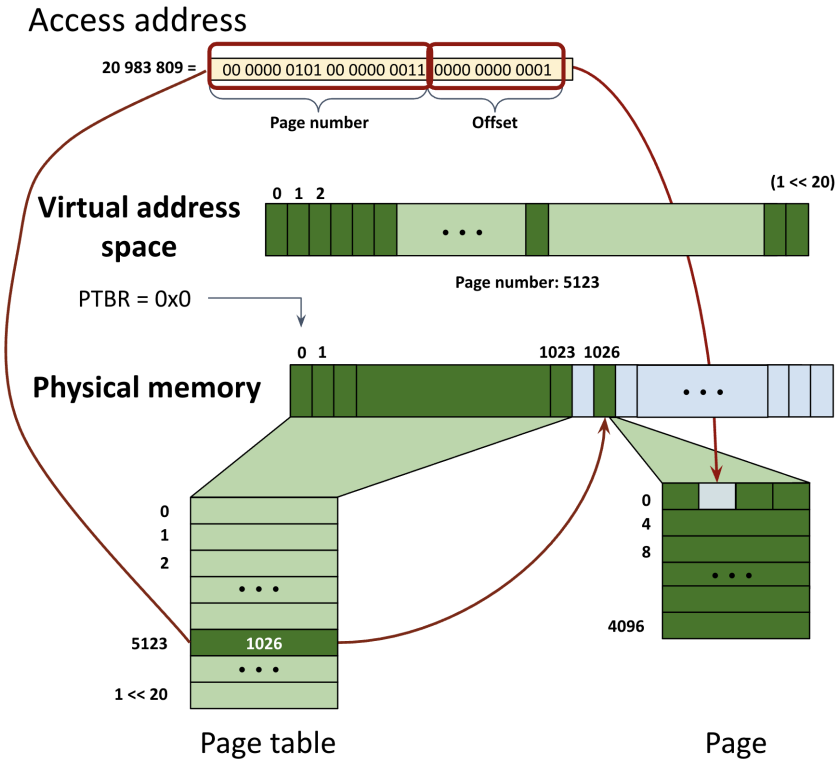
   - The Page Table Base Register (PTBR) points to the beginning of the page table in physical memory (in our example, `PTBR = 0x0`).
   - The page number extracted from the virtual address (`5123`) is used as an index into the linear page table.

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.

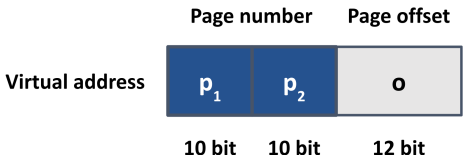
**Example 5.3.3.1** (Two-Level Paging Example)**.** *A virtual address on a 32-bit machine with a 4 KiB page size is divided as follows:*

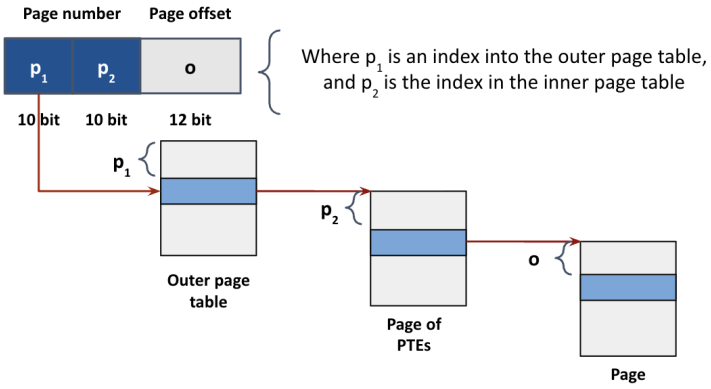
- ***Page offset:*** *12 bits.*
- ***Page number:*** *20 bits.*

*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:

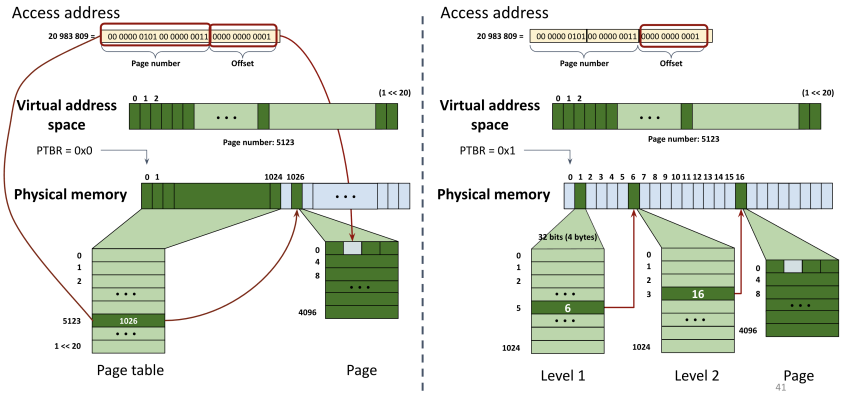
$$\underbrace{\text{Page Number}}_{20\text{ bits}} \quad \underbrace{\text{Offset}}_{12\text{ bits}}.$$

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 |

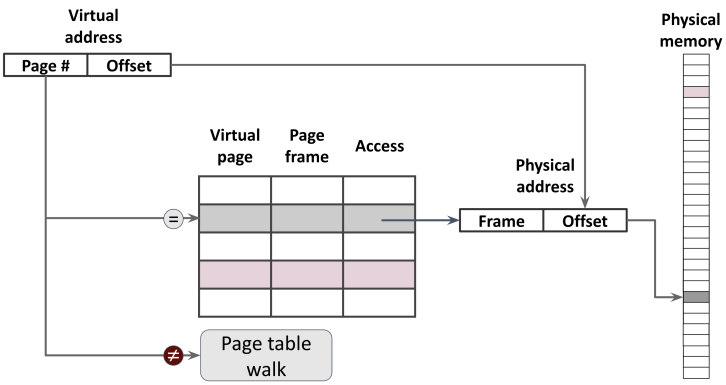
**Key:** The TLB is implemented as a dedicated circuit, separate from main memory, enabling rapid address translation.

### 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

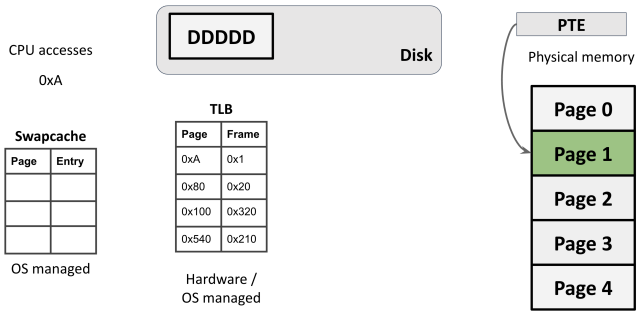
When a process accesses a page that is not present in main memory, the Memory Management Unit (MMU) cannot translate the virtual address because the corresponding page table entry indicates that the page is absent. This situation results in a *page fault*, prompting the OS to bring the page back from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

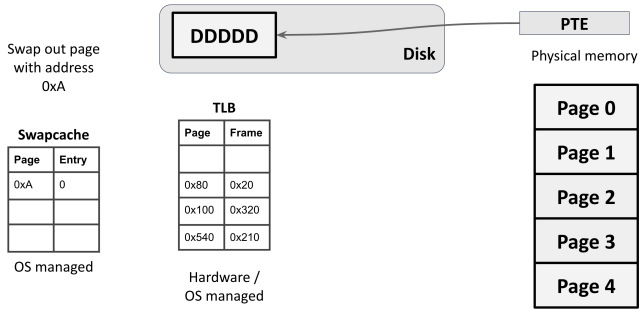


### 5.5.3 Swapping Out: Freeing Up Memory

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.

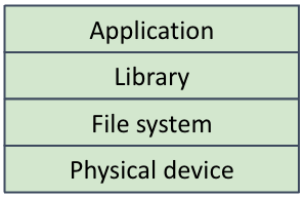
## 6.2 I/O Operations and File System Layers

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).**

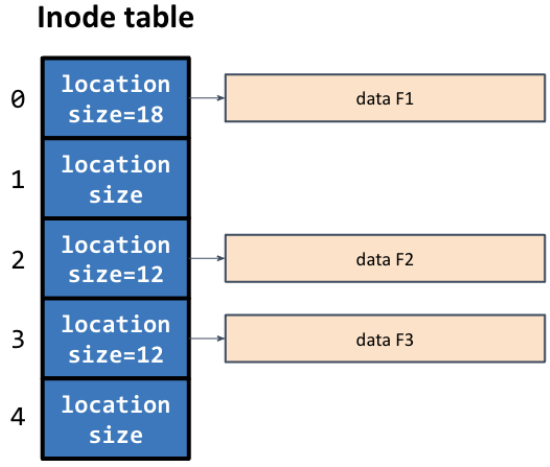
*A directory is a special type of file containing an array of mappings between filenames and inode numbers.*

**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.*



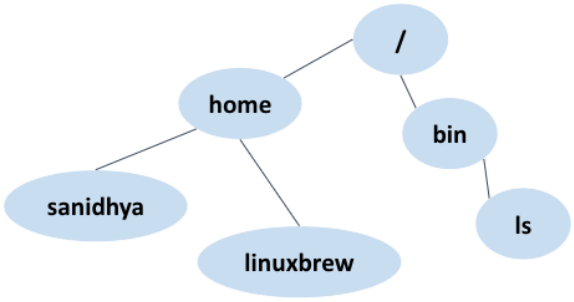

**Relationship Between Inodes and Directories**

Important distinctions regarding directories and inodes:

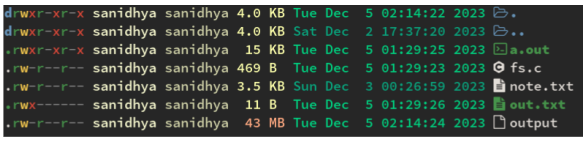
- Directories themselves are stored as regular files but with a special identifying flag.
- This special flag restricts operations (e.g., writing directly to directory files is prohibited).
- Directories contain arrays of pairs {`filename, inode number`}.
- An inode does **not** store its filename; filenames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

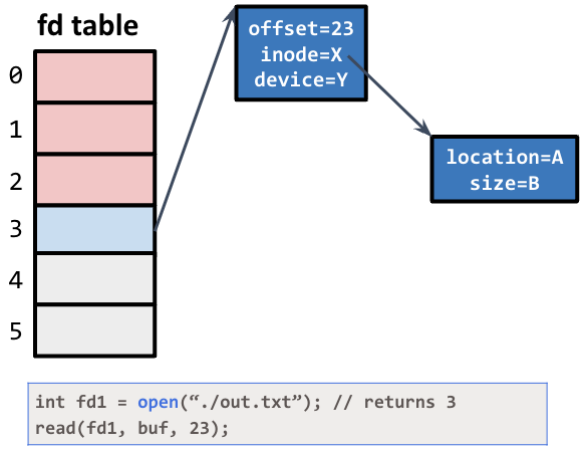
File system operations can be implemented using file names along with inode and device IDs. However, performing a lookup from a file name to its corresponding inode/device ID for every operation can be inefficient.
To overcome this, most systems perform the expensive directory traversal once and then 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

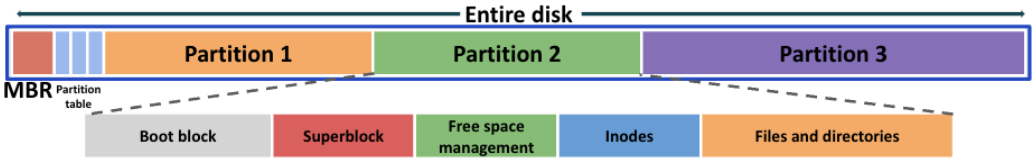
### 6.6.1 File System Implementation

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:

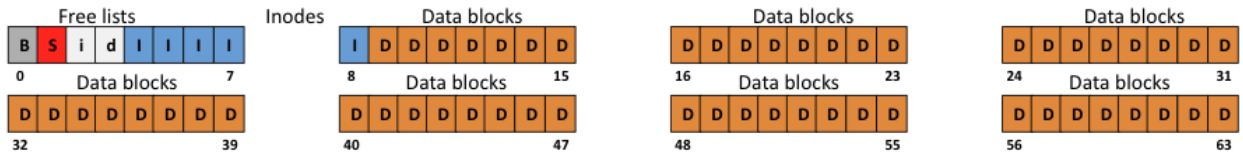



**Disk Structure**

- **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:

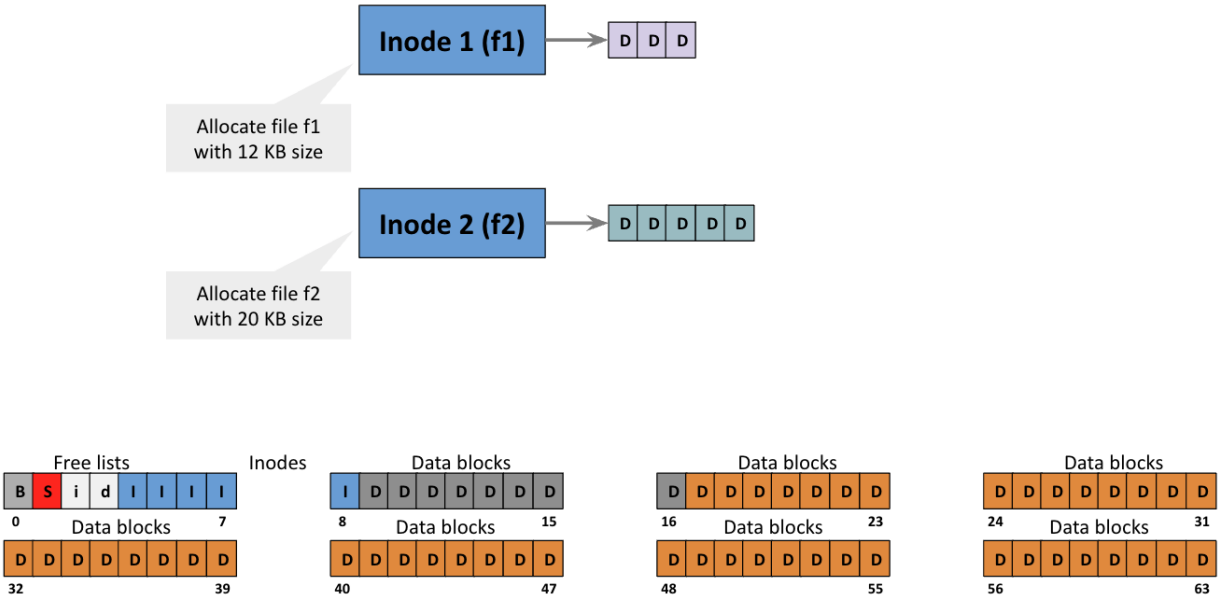
- File type (regular file, directory, symbolic link)
- User ID of the owner
- Permissions (Read/Write/Execute)
- File size in bytes
- Block addresses containing the file's data
- Creation timestamp
- Number of linked paths (hard links)
- Counts of direct and indirect data blocks

### 6.6.6 File Allocation Methods

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

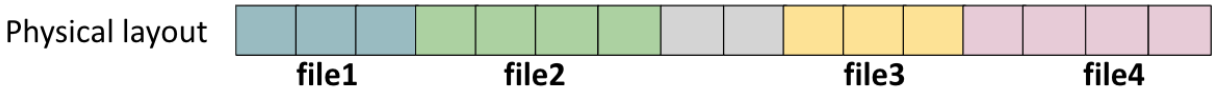
- **Contiguous Allocation**: Files stored in sequential blocks.
- **Linked Allocation**: Files stored as linked lists of blocks.
- **File Allocation Table (FAT)**: Uses a central table to manage file blocks.
- **Multi-level Indexed Allocation**: Files managed through hierarchical index pointers.

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

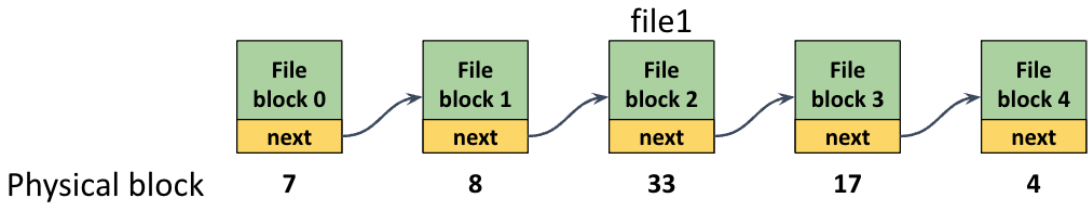
In contiguous allocation, all blocks of a file are stored consecutively on disk.




- **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

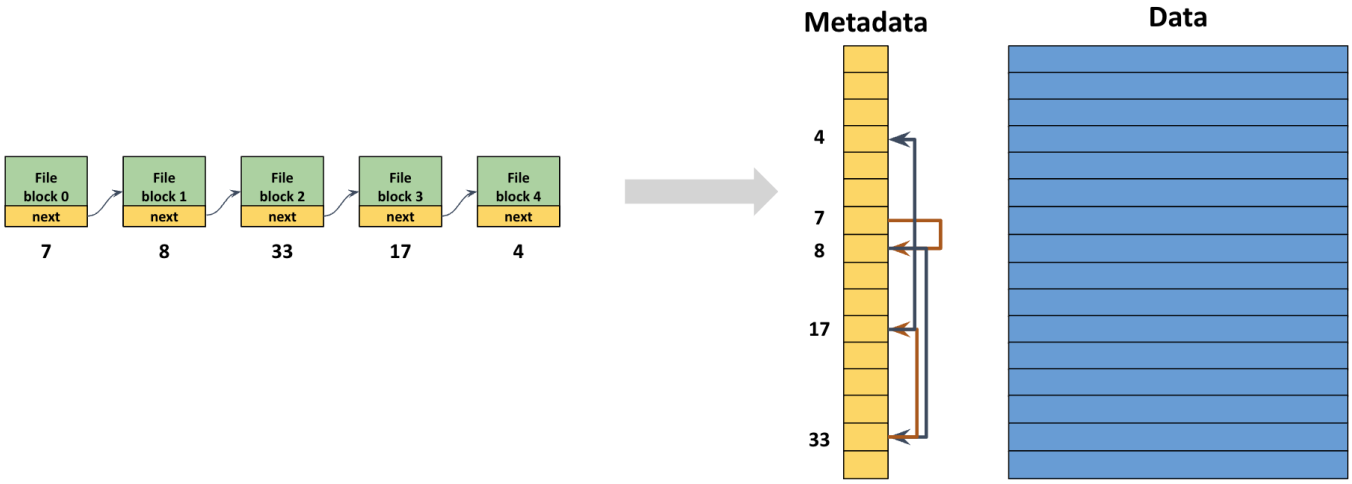
In linked allocation, each file is stored as a linked list of disk blocks. Each block contains data and a pointer to the next block.




- **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 $\mu s$, reading 5 blocks requires 500 $\mu s$.
- **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 $\mu$s

**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):**

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

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

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

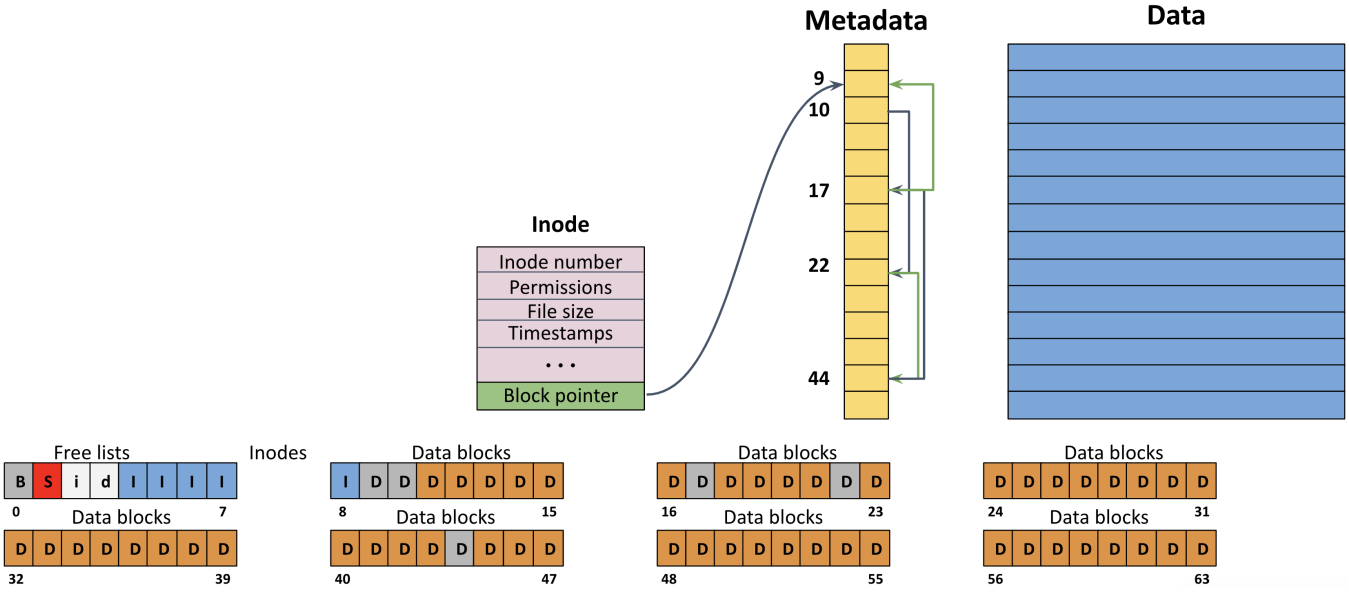
**Time to access all metadata (worst case):**

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

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

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

**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

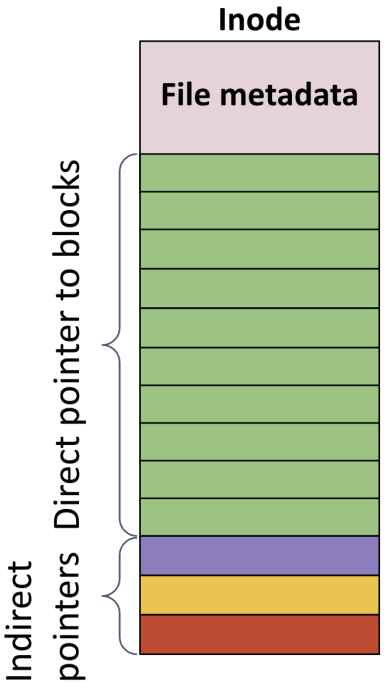
A well-designed block allocation strategy should balance several competing requirements:

- 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.

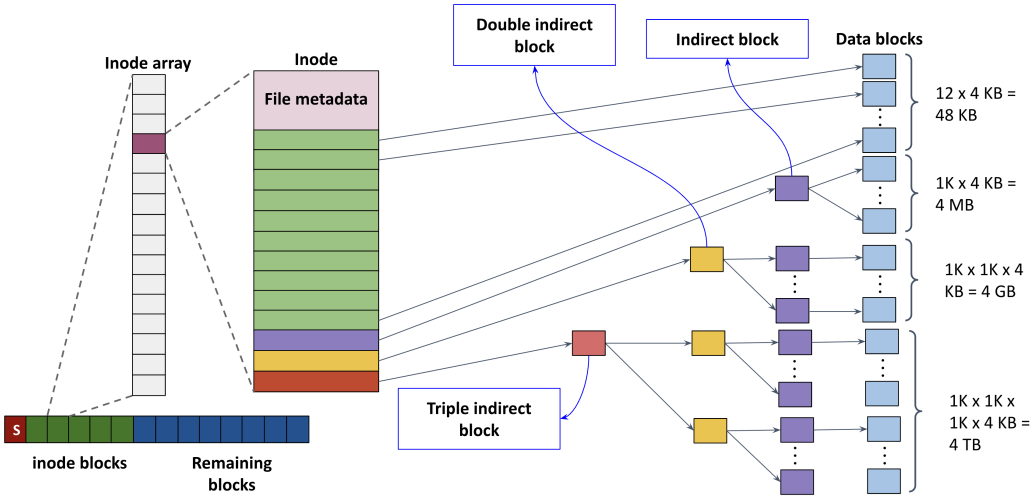
### Key Features and Advantages

- **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:

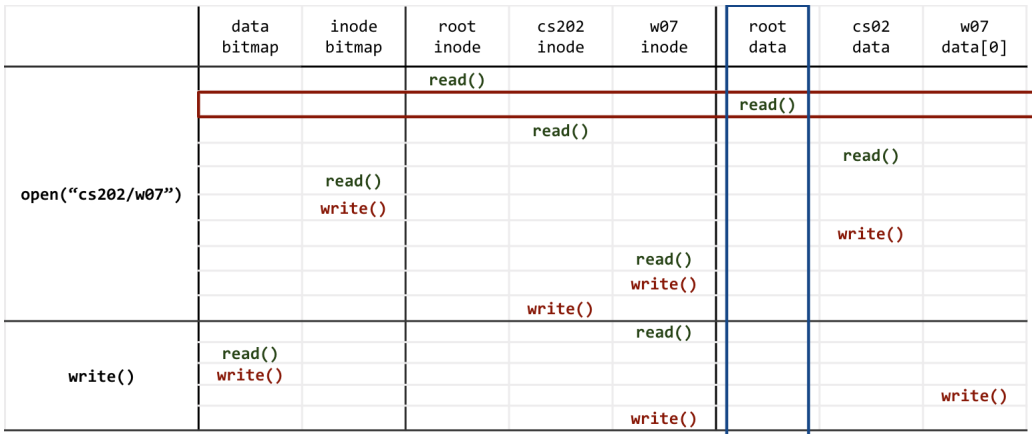
1. Read the free data block bitmap to locate available space
2. Write to the data block bitmap to mark the block as allocated
3. Read the file's inode to access its metadata
4. Write to the file's inode to update its block pointers
5. Write the actual data to the newly allocated block

#### 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:

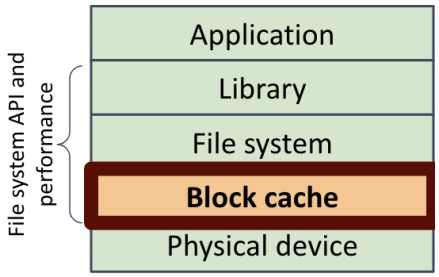
- Maintaining abstractions that decouple logical operations from physical ones
- Allowing the system to reorder or coalesce operations
- Providing flexibility in how and when operations are physically executed

### 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.*

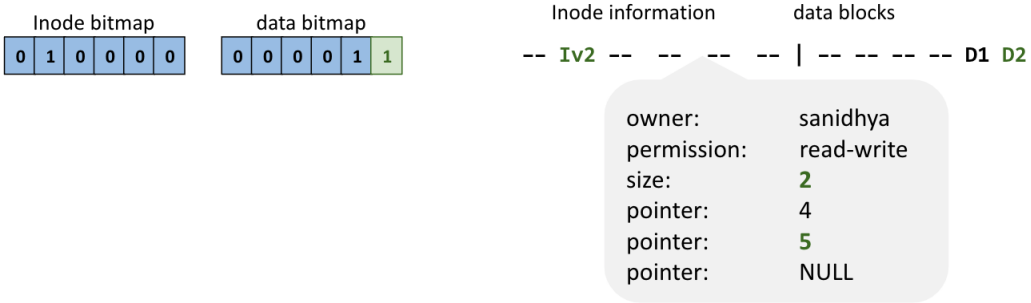
**Definition (Write-Through Cache).** *A caching policy where modifications are immediately written to both cache and disk, ensuring consistency at the cost of performance.*

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

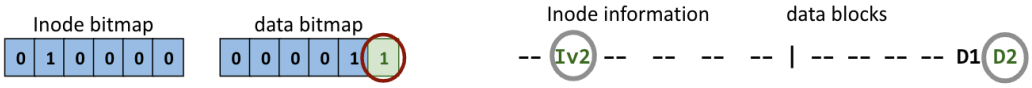



What happens if a crash or power outage occurs during these writes? The key issue is that file system operations often involve multiple write operations, and a failure between these operations can lead to an inconsistent state.

### 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)

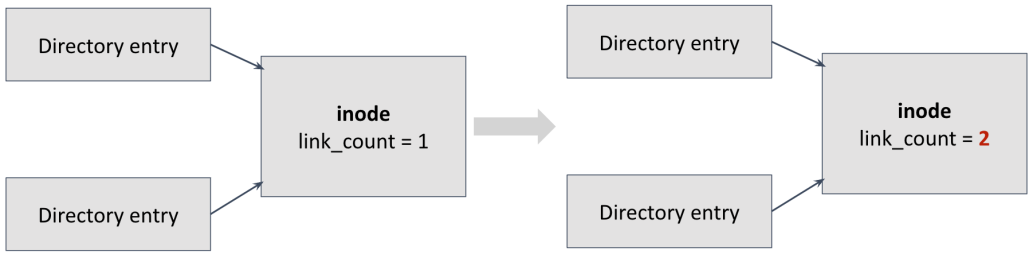
One approach to address file system inconsistencies is using a file system checker (FSCK). FSCK is a utility that checks the consistency of the file system after a certain number of mount operations or after a crash. It performs hundreds of consistency checks across different fields, such as:

- 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

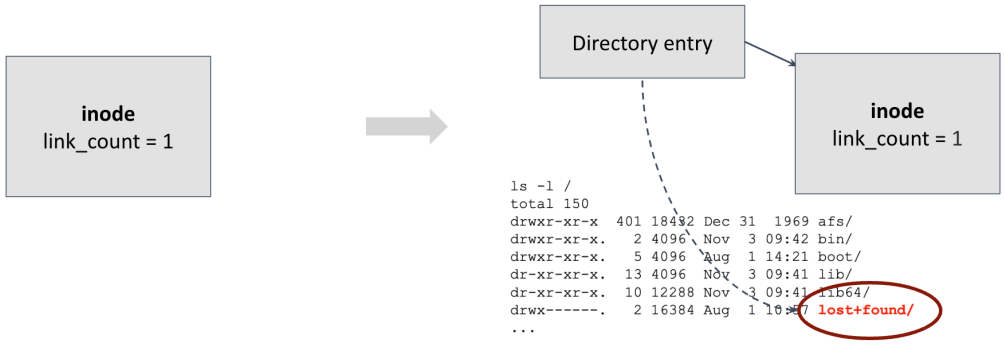
The file system checker (`fsck`) is a crucial utility for maintaining file system integrity. It is automatically invoked after a specific number of mount operations or following a system crash to ensure the file system's consistency. The `fsck` performs numerous checks across various file system components, 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.



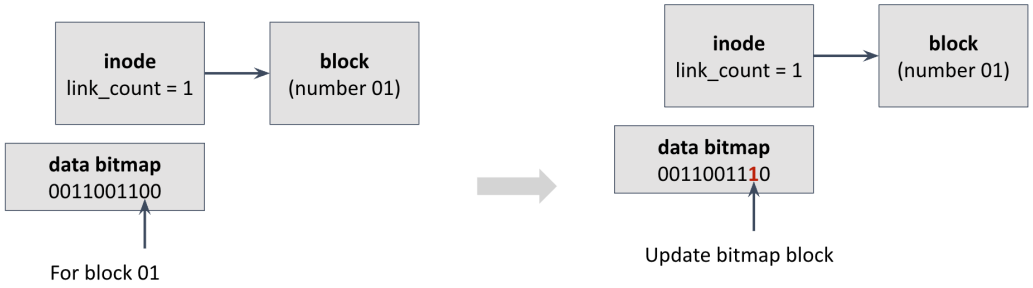

Fix the link count by increasing to 2

- **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.



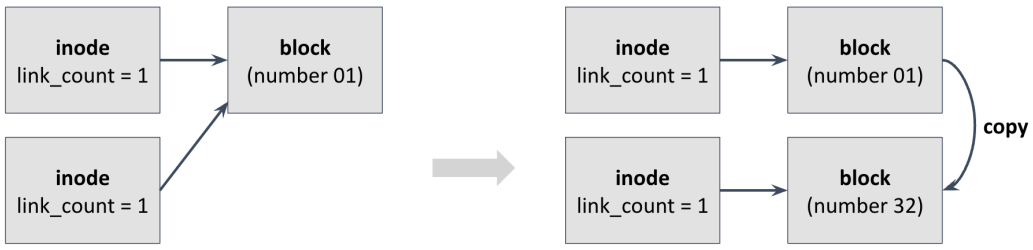

Link the file in lost+found directory

- **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.



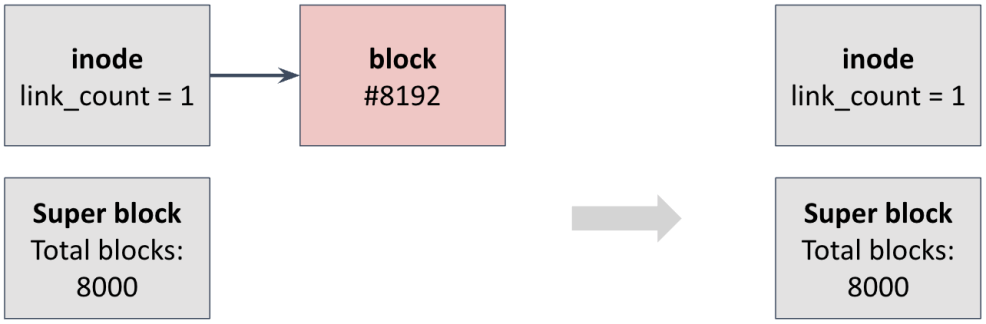

Update a reference block in the bitmap

- **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.