Introduction to Operating Systems

This chapter serves as a foundational overview of Operating Systems (OS), defining their purpose, core functionalities, and the fundamental problems they solve. It introduces the three major themes that will be explored throughout the book: **Virtualization**, **Concurrency**, and **Persistence**.

1. What is an Operating System?

At its core, a running program simply executes instructions. However, a lot more happens behind the scenes to make a computer system usable and efficient. This is where the Operating System comes in.

1.1. The Role of the OS

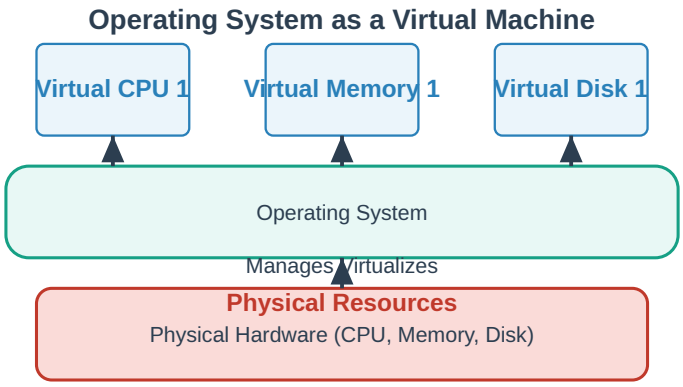
The OS is a body of software responsible for:

- Making it easy to run programs, even seemingly many at once.
- Allowing programs to share system resources like memory.
- Enabling programs to interact with hardware devices.
- Ensuring the system operates correctly and efficiently.

1.2. OS as a Virtual Machine (Virtualization)

The primary technique the OS uses to achieve its goals is **virtualization**.

- **Concept:** The OS takes a physical resource (e.g., processor, memory, disk) and transforms it into a more general, powerful, and easy-to-use **virtual form**.
- **Analogy:** The OS creates an illusion, making it seem as though each program has its own dedicated resources, even when they are shared.
- **Example:** A single CPU can appear as many "virtual CPUs," allowing multiple programs to run concurrently.



1.3. OS as a Resource Manager

Beyond virtualization, the OS also acts as a **resource manager**.

- **Resources:** The CPU, memory, and disk are all shared resources of the system.
- **Management:** The OS's role is to manage these resources, allocating them among competing programs based on various goals (e.g., efficiency, fairness).

1.4. OS Interfaces (APIs / System Calls)

To allow users and applications to interact with the OS and utilize its features (like running programs, allocating memory, or accessing files), the OS provides **interfaces** or **APIs** (Application Programming Interfaces).

- **System Calls:** These are special functions that applications can call to request services from the OS kernel.
- **Standard Library:** The OS can be seen as providing a standard library of services to applications through these system calls.

---

2. Core OS Themes: Virtualization, Concurrency, Persistence

The book organizes the study of operating systems around three major conceptual themes, each addressing a fundamental problem.

2.1. Virtualizing the CPU (Time Sharing)

The OS creates the illusion that each program has its own CPU, even on a system with only one physical processor.

- **The Illusion:** Many programs appear to run "at the same time."
- **Mechanism:** The OS achieves this through **time sharing**, rapidly switching the CPU between different running programs. Each program runs for a short period, then the OS switches to another, creating the appearance of simultaneous execution.
- **Example Program (`cpu.c`):** This simple program spins for a second, then prints a character, repeating forever.

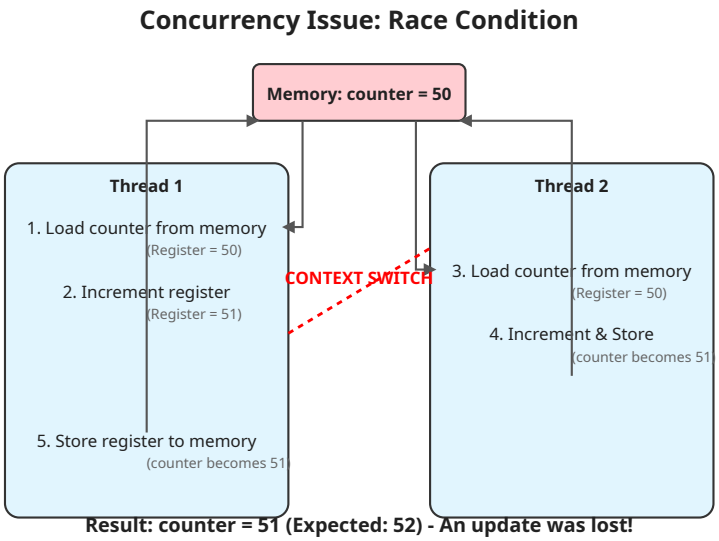
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h" // Assumed to contain Spin(1)

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1); // Spins for 1 second
        printf("%s\n", str);
    }
    return 0;
}
```

- **Demonstration:** When multiple instances of `cpu.c` are run concurrently (e.g., `./cpu A & ./cpu B & ./cpu C & ./cpu D &`), their output interleaves, showing the illusion of parallel execution on a single CPU.

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
...
```

- **Crux of the Problem:** How does the operating system virtualize re-

sources? What mechanisms and policies are implemented to achieve this efficiently, and what hardware support is needed?



2.2. Virtualizing Memory

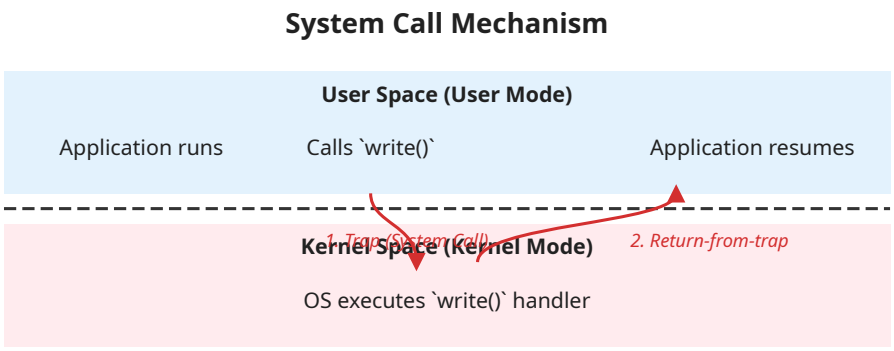
The OS creates the illusion that each program has its own private memory, even though physical memory is a shared resource.

- **Physical Memory Model:** Memory is a simple array of bytes, accessed by specifying an address.
- **The Illusion:** Each running program appears to have its own **private virtual address space**.
- **Mechanism:** The OS maps these virtual address spaces onto the physical memory of the machine. A memory reference by one program does not affect the address space of others.
- **Example Program (`mem.c`):** This program allocates memory using `malloc()`, prints its address, and then continuously increments the value stored at that address.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h" // Assumed to contain Spin(1)

int main(int argc, char *argv[]) {
    int *p = malloc(sizeof(int)); // a1
    assert(p != NULL);
```

```
    printf("(%d) address pointed to by p: %p\n", getpid(), p); // a2

    *p = 0; // a3
    while (1) {
        Spin(1);
        *p = *p + 1; // a4
        printf("(%d) p: %d\n", getpid(), *p);
    }
    return 0;
}
```

- **Demonstration:** When multiple instances of `mem.c` are run, they might all report allocating memory at the *same virtual address* (e.g., `0x200000`), yet they increment their respective counters independently. This shows that each process has its own isolated view of memory.



2.3. Concurrency

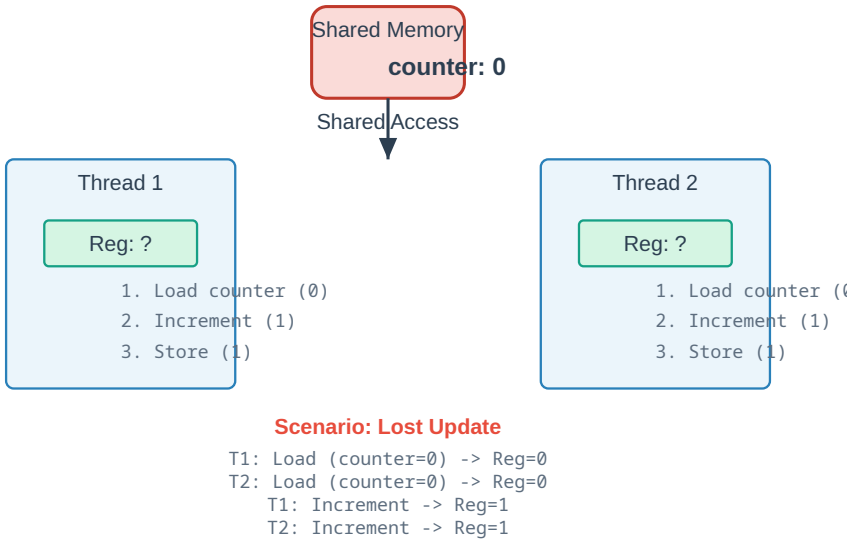
Concurrency refers to the challenges that arise when multiple tasks (or parts of the same program) are working on "many things at once" within the same shared environment, particularly the same memory space.

- **The Problem:** When multiple threads or processes access and modify shared data, the interleaving of their instructions can lead to non-deterministic and incorrect results.
- **Example Program (`threads.c`):** This program creates two threads that both increment a shared global `counter` variable in a loop.

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h" // Assumed to contain Pthread_create, Pthread_join

volatile int counter = 0; // Shared global variable
```

```c
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++; // Increment the shared counter
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL); // Create thread 1
    Pthread_create(&p2, NULL, worker, NULL); // Create thread 2
    Pthread_join(p1, NULL); // Wait for thread 1 to finish
    Pthread_join(p2, NULL); // Wait for thread 2 to finish
    printf("Final value   : %d\n", counter);
    return 0;
}
```

- **Demonstration (Race Condition):**
  - Expected result: If `loops` is 1000, `counter` should be 2000 (1000 from each thread).
  - Actual result: For large `loops` values (e.g., 100,000), the final `counter` value is often incorrect and varies between runs (e.g., 143012, 137298).
  - **Reason:** The `counter++` operation is not atomic. It involves three machine instructions:
    1. Load `counter` value from memory into a CPU register.
    2. Increment the value in the register.
    3. Store the new value from the register back to memory. If threads are interrupted and interleaved during these three steps, updates can be lost.
- **Crux of the Problem:** How to build correctly working programs when there are many concurrently executing threads within the same memory space? What primitives and mechanisms are needed from the OS and hardware?



2.4. Persistence

Persistence deals with the challenge of storing data reliably and durably, ensuring it survives even if the system loses power or crashes.

- **The Problem:** System memory (DRAM) is volatile; data is lost on power loss. Users care about their data.
- **Hardware:** I/O devices like hard drives and SSDs provide persistent storage.
- **Software:** The **file system** is the part of the OS responsible for managing persistent data on these devices, storing files reliably and efficiently.
- **Shared Files:** Unlike CPU and memory virtualization, the OS does *not* create private, virtualized disks for each application. Files are often shared between different programs (e.g., an editor, a compiler, and an executable all interact with the same source code file).
- **Example Program (`io.c`):** This program demonstrates basic file I/O by opening, writing to, and closing a file.

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h> // For O_WRONLY, O_CREAT, O_TRUNC
#include <sys/types.h> // For mode_t (S_IRWXU)
```

```c
int main(int argc, char *argv[]) {
    // Open/create file /tmp/file with write-only, create, truncate modes
    // S_IRWXU gives read/write/execute permissions to owner
    int fd = open("/tmp/file", O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU);
    assert(fd > -1); // Ensure file opened successfully

    // Write "hello world\n" (13 bytes) to the file
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13); // Ensure all bytes were written

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

- **OS Role in I/O:** The OS handles the complex, low-level details of interacting with storage devices (e.g., figuring out where data resides on disk, managing internal file system structures, issuing I/O requests). It provides a simplified, standard interface (system calls like `open()`, `write()`, `close()`) to applications.
- **Performance & Reliability:** File systems often delay writes (buffering) for performance and use intricate protocols (journaling, copy-on-write) to ensure data integrity even during system failures.
- **Crux of the Problem:** How to store data persistently? What techniques, mechanisms, and policies are needed for correctness, high performance, and reliability in the face of hardware and software failures?

---

3. Design Goals of an Operating System

When building an OS, several key goals guide its design and implementation:

- **Abstractions (Convenience):** Creating higher-level, simpler interfaces to complex underlying hardware. This makes the system easier to use and program.
- **Performance (Efficiency):** Minimizing the overhead introduced by the OS itself (e.g., extra time for operations, extra memory/disk space). Virtualization is valuable, but not at any cost.
- **Protection (Isolation):** Ensuring that the malicious or accidental bad behavior of one application does not harm other applications or the OS itself. This is achieved by isolating processes from one another.
- **Reliability:** Striving for a high degree of dependability, as an OS failure impacts all applications. Building reliable OSes is a significant challenge due to their complexity.

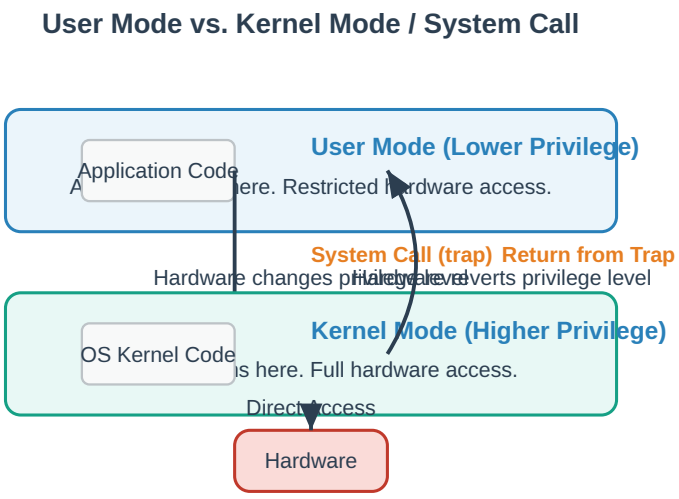
- **Other Goals:** Energy-efficiency, security (an extension of protection), and mobility are increasingly important in modern systems.

---

4. A Brief History of Operating Systems

The evolution of OSes reflects accumulated good ideas and lessons learned over time.

- **Early Operating Systems (Just Libraries):**
  - Initially, OSes were simple collections of commonly used functions (e.g., I/O handling) provided as libraries.
  - One program ran at a time, controlled by human operators (batch processing). Interactive use was too expensive.
- **Beyond Libraries (Protection & System Calls):**
  - The need for **protection** emerged: OS code (with hardware control) needed to be special and isolated from application code.
  - **System Calls** were invented (pioneered by Atlas computing system).
    - Unlike procedure calls, system calls transfer control to the OS while simultaneously **raising the hardware privilege level** (from **user mode** to **kernel mode**).
    - In kernel mode, the OS has full hardware access.
    - A special hardware instruction (trap) initiates the system call.
    - After servicing the request, the OS returns control to the user application via a "return-from-trap" instruction, lowering the privilege level back to user mode.



- **The Era of Multiprogramming:**

  - Driven by the desire to better utilize expensive machine resources (minicomputers).
  - OSes would load multiple jobs into memory and rapidly switch between them, improving CPU utilization (especially when programs were waiting for slow I/O).
  - This introduced new challenges: memory protection between programs and handling concurrency issues (e.g., interrupts).
  - **UNIX:** A major practical advance, developed by Ken Thompson and Dennis Ritchie at Bell Labs. It simplified and integrated many good ideas from earlier systems. Its design principles (small, powerful programs connected by pipes) and open distribution made it immensely popular.
- **The Modern Era:**
  - **Personal Computers (PCs):** Early PC OSes (like DOS, early Mac OS) initially regressed, often lacking features like memory protection and robust multitasking.
  - **Re-adoption of Good Ideas:** Over time, features from minicomputer OSes (like UNIX's core principles) were re-integrated into modern desktop OSes (e.g., macOS/OS X, Windows NT).
  - **Linux:** Linus Torvalds's independent re-implementation of UNIX principles, combined with GNU tools, led to Linux, which became dominant in servers, mobile (Android), and even desktops.
  - Modern OSes (including those on smartphones) now incorporate sophisticated features developed in earlier eras, making systems more robust and powerful.

---

5. Summary

The operating system is a complex but essential piece of software that makes computer systems usable, efficient, and reliable. It achieves this primarily through **virtualization** of resources (CPU, memory), managing **concurrency** (handling multiple tasks safely), and ensuring **persistence** of data (via file systems). Modern OSes are built upon decades of accumulated good ideas, many of which originated from systems like UNIX.