

# CS 162: Operating Systems and Systems Programming

## Lecture Notes

Mokhalad Aljuboori

Fall 2024 — Ion Stoica

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is an Operating System? . . . . .	1
1.2	Operating System Evaluation . . . . .	2
<b>2</b>	<b>Four Fundamental OS Concepts</b>	<b>3</b>
2.1	The Process Abstraction . . . . .	3
2.2	Dual-Mode Operation . . . . .	3
2.2.1	Privileged Instructions . . . . .	4
2.2.2	Memory Protection . . . . .	4
2.2.3	Timer Interrupts . . . . .	4
2.3	Types of Mode Transfers . . . . .	5
2.3.1	User to Kernel Mode Transfer . . . . .	5
2.3.2	Kernel to User Mode Transfer . . . . .	5
<b>3</b>	<b>Abstractions 1: Threads and Processes A quick, programmer's viewpoint</b>	<b>6</b>
<b>4</b>	<b>Abstractions 2: Processes, Files and I/O</b>	<b>7</b>
4.0.1	Creating Processes . . . . .	7

08/29/24

## Lecture 1

### Introduction

## 1.1 What is an Operating System?

An operating system is the layer of software that manages a computer's resources for its users and their applications. In other words, the OS is a special layer of software that enables the applications to run and share the hardware of a system.

OS provides:

1. Convenient abstraction of complex hardware devices.

2. Protected access to shared resources
3. Security and authentication
4. Communication amongst logical entities

How can an operating system run multiple applications? For this, operating systems need to play three roles:

1. **Referee:** Since the applications share physical resources, the operating system needs to decide which application gets what resources and when. This role is somewhat akin to that of a patient kindergarten teacher. It balances needs, separates conflicts, and facilitates sharing.
2. **Illusionist:** OS's needs to provide an abstraction of physical hardware to simplify application design. They need to provide the illusion of nearly infinite memory. Also provide the illusion that each program has the computer's processors entirely to itself. These illusions let you write applications independently of the amount of physical memory on the system or the physical number of processors.
3. **Glue:** OS's needs to provide a set of common services that facilitate sharing among applications. Many OS's provide a common user interface routines so applications can have the same "look and feel". Most importantly, the OS provides a layer separating applications from hardware I/O devices like monitors, keyboards, and mice so applications can be written independently of the specific hardware.

#### Definition 1.1: Process

A process is an execution environment provided by the OS to run a program. A process has its own memory space, its own file descriptors, and its own CPU time. The OS can run multiple processes at the same time, switching between them to give the illusion that they are all running concurrently.

A process consists of:

- Address space
- One or more threads of control executing in that address space.
- Additional system states.

The OS provides access protection between processes. Process 2 cannot access the resources and memory used by Process 1 and vice versa.

Note that a processor can only run one process at a time. So how could it possibly run multiple processes? The OS uses a technique called context switching, switching between executing different processes very fast.

## 1.2 Operating System Evaluation

Having defined what an operating system does, how should we choose among alternative designs? We discuss several desirable criteria for operating systems:

1. **Reliability and Availability:** Does the operating system do what you want?
2. **Security:** Can the operating system be corrupted by an attacker?
3. **Portability:** Is the operating system easy to move to new hardware platforms?
4. **Performance:** Is the user interface responsive, or does the operating system impose too much overhead?
5. **Adoption:** How many other users are there for this operating system?

In many cases, tradeoffs between these criteria are inevitable — improving a system along one dimension may hurt it along another.

09/03/24

## Lecture 2

### Four Fundamental OS Concepts

#### Definition 2.1: Kernel

Implementing protection is the job of the *operating system kernel*. The kernel is the lowest level of software running on the system, and has full access to all of the machine hardware. Therefore the kernel is necessarily *trusted* to do anything with the hardware. Everything else — that is, the untrusted software running on the system — is run in a restricted environment with less than complete access to the full power of the hardware.

## 2.1 The Process Abstraction

#### Definition 2.2: Process

You can think of a process as an instance of a program, same way that an object is an instance of a class in OOP. Each process has its own memory space for program's data, stack, and heap. The process is the abstraction for protected execution provided by the operating system kernel.

A process needs permission from the kernel before accessing the memory of any other process, before reading or writing to the disk, before changing hardware settings, and so forth.

Applications/programs need to run on the processor with all potentially dangerous operations **disabled**. To make this work, hardware needs to support **dual-mode operation**. See next section.

The OS keeps track of the various process on the computer using a data structure called the *process control block*.

#### Definition 2.3: Process Control Block (PCB)

The PCB is a data structure that stores all the information the OS needs about a particular process, including:

- Status (Running, ready, blocked, ...)
- Where its stored in memory,
- Where its executable image resides on disk,
- Which user asked it to start executing,
- What privileges the process has,
- and so forth.

## 2.2 Dual-Mode Operation

How does the operating system kernel prevent a process from harming other processes or the operating system itself? What prevents a process from overwriting another process's data structures, or even overwriting the operating system image stored on disk?

It's done in hardware, and it's called **dual-mode operation**. Most instructions are perfectly safe, such as adding two registers together and storing the result in a third register. But some instructions are dangerous, such as reading from or writing to disk, or changing the memory protection settings. The hardware must be able to distinguish between these two types of instructions.

**Definition 2.4: Dual-Mode Operation**

A single bit in the processor status register is used to signify the current mode of the processor. The mode bit is modified by some instructions. When set to 1, the processor is in **kernel mode**. When set to 0, the processor is in **user mode**.

In **user mode**, the processor checks each instruction before executing it to verify that it is permitted to be performed by that process. We'll describe the specific checks soon.

In **kernel mode**, the OS executes the instruction with protection checks disabled.

What hardware is needed to let the operating system kernel protect applications and users from one another, yet also let user code run directly on the processor? At a minimum, the hardware must support the following three things:

1. **Privileged Instructions.** All potentially unsafe instructions are prohibited when executing in user mode.
2. **Memory Protection.** All memory accesses outside of a process's valid memory region are prohibited when executing in user mode.
3. **Timer Interrupts.** Regardless of what the process does, the kernel must have a way to periodically regain control from the current process.

In addition, the hardware must also provide a way to safely transfer control from user mode to kernel mode and back. We'll discuss how this is done in a later section.

**2.2.1 Privileged Instructions**

Instructions available only in kernel mode are called **privileged instructions**. The OS kernel must be able to execute these instructions to do its work. Thus, while application programs can use only a subset of the full instruction set, the OS executes in kernel mode with the full power of the hardware.

If an application were to attempt to access restricted memory or attempts to change its privilege level, such actions would cause a *processor exception* to occur. Unlike exceptions in a high-level language, a processor exception causes the processor to transfer control to an exception handler in the OS kernel. Usually, the kernel simply halts the process after a privilege violation.

**2.2.2 Memory Protection**

To run an application process, both the operating system and the application must be resident in memory at the same time. Further, other application processes may also be stored in memory. They share the same physical memory.

To make memory sharing safe, the OS must be able to configure the hardware so that each application process **can read and write only its own memory**, not the memory of the OS or any other application. Otherwise, an application could modify the operating system kernel's code or data to take control of the system or corrupt other applications.

While it may seem that read-only access to memory is harmless, recall that the OS needs to provide both **security and privacy**. Kernel data structures may contain private user data, and kernel memory may contain sensitive information such as passwords while they are being verified.

**2.2.3 Timer Interrupts**

Process isolation also requires to provide a way for the operating system kernel to periodically regain control of the processor. This is done using a **timer interrupt**. If the application enters an infinite loop, or if the user simply becomes impatient and wants the system to stop the application, then the operating system must be able to regain control.

The operating system also needs to regain control of the processor in normal operation. To respond to user input in a timely manner while also running other applications, the OS must be able to regain control to switch to a new task.

A **hardware timer** device is used to generate periodic interrupts after a specified delay (either in time or after some number of instructions have been executed).

When the timer interrupt occurs, the processor stops executing the current process and transfers control from the user process to the kernel running in kernel mode. The kernel can then decide what to do next, such as switching to a new process or continuing to run the current process.

Resetting the timer is a privileged operation, so only the kernel can reset the timer. This prevents an application from disabling the timer and running forever. A timer or other interrupt does not imply that the program has an error.

## 2.3 Types of Mode Transfers

The question is now, how to safely transition between executing a user process to executing the kernel and vice versa. This mechanism must be both fast and safe, as it is used frequently (a high performance web server could switch between user and kernel thousands of times per second), leaving no room for error.

We'll discuss first transfers from user to kernel mode, as transitioning in the other direction works by "undo"-ing the transition from user process into the kernel.

### 2.3.1 User to Kernel Mode Transfer

There are three reasons for a user process to transfer control to the kernel. Interrupts, processor exceptions, and system calls.

1. **Interrupts:** An interrupt is an **asynchronous** signal to the processor that some external event has occurred that may require the processor's attention. As the processor is executing instructions, it checks for whether an interrupt has arrived. If so, it completes or stalls any instructions in progress, and instead of fetching the next instruction, the processor hardware saves the current execution state and starts executing at a specially designated interrupt handler in the kernel. Each different type of interrupt requires its own handler and there are many types of interrupts.
2. **Processor Exceptions:**
3. **System Calls:** User processes can transition into the operating system kernel voluntarily to request that the kernel perform an operation on the user's behalf. A **system call** is any procedure provided by the kernel that can be called from user level.

### 2.3.2 Kernel to User Mode Transfer

---

Today, the four Fundamental OS Concepts are

- Thread: Execution context
  - Fully describe program state
- Address Space (with or w/o translation)
- Process: an instance of a running program
- Dual Mode operation / Protection

The bottom line of an OS is **to run programs**. Here is how they are run:

1. Write and compile them
2. Load instruction and data segments of executable file into memory
3. Create stack and heap
4. "Transfer control to program"
5. Provide services to program

09/05/24

## Lecture 3

*Abstractions 1: Threads and Processes A quick, programmer's viewpoint***Definition 3.1: Thread**

A single unique execution context. It provides the abstraction of: a single execution sequence that represents a separately scheduled task. Threads are a mechanism for concurrency.

Concurrency is about handling multiple things at once, it's not parallelism. Parallelism is about doing multiple things at once.

09/10/24

**Lecture 4***Abstractions 2: Processes, Files and I/O*

There are 3 types of Kernel Mode Transfers.

1. Syscall.
2. Interrupts
3. Trap or Exception, eg. Seg fault.

**4.0.1 Creating Processes**