EECS 482: Intro to Operating Systems

Noah Peters

April 21, 2023

Abstract

Notes for EECS 482 at the University of Michigan, Winter 2018. Note template by Pingbang Hu.

Contents

Chapter 1

Introduction

Lecture 1: Introduction

1.1 What is an Operating System?

What would happen if we ran applications directly on hardware? We would quickly run into the problems of **portability** and **resource sharing**. What can we do to resolve these problems? We can use an **operating system**:

Definition 1.1.1 (Operating system)**.** The operating system is the software layer between user applications and the hardware.

We define particular roles for the OS:

- **Illusionist**: it creates abstractions (i.e. CPU → Threads; Memory → Address space)
- **Government**: it manages shared hardware resources (but at a cost)

1.2 OS and Apps: 2 Perspectives

We can try two different approaches to our operating system:

1.2.1 App as the Main Program

Our first perspective consists of the *application as the main program*. Under this perspective, the application gets services by calling the kernel (OS). However it has its problems:

i. How does the application start?

ii. How do tasks occurring outside any program get done?

iii. How do multiple programs run simultaneously without messing each other up?

1.2.2 OS as the Main Program

We try another perspective where the operating system is the main program. Under this perspective, the operating system *calls applications as subroutines*. The *illusion* is that every app runs on its own computer. We have the lower layer OS invoking higher layer applications. The app or processor returns control to the OS.

1.3 History of Operating Systems

Beyond the most basic hardware operator, a human, we observe the most basic forms of operating systems and work our way forward.

Definition 1.3.1 (Batch Processing). Batch processing was a punchcard method of improving CPU and I/O utilization by removing user interaction. The operating system is the batch monitor and a library of standard services.

However, in batch processing (i.e. where a human processes punchcards in batches), new problems arise such as protection and uninteractivity. We can improve our batching of punchcards by using a **multi-programmed batch**.

Definition 1.3.2 (Multi-Programmed Batch). An improved version of batch processing where utilization is improved by overlapping CPU and I/O on different (concurrent) programs.

This operating system is more complex, as it can:

i. Run multiple processes or I/Os concurrently

ii. Enable simultaneous CPU and I/O

iii. Protect processes from each other

However, it is still **not interactive**. We can address this by adding user access to the system via **time sharing**.

Definition 1.3.3 (Time Sharing). Model user as a (very slow) I/O device, allowing people to interact with programs as they run. In other words, *share* some runtime with the User.

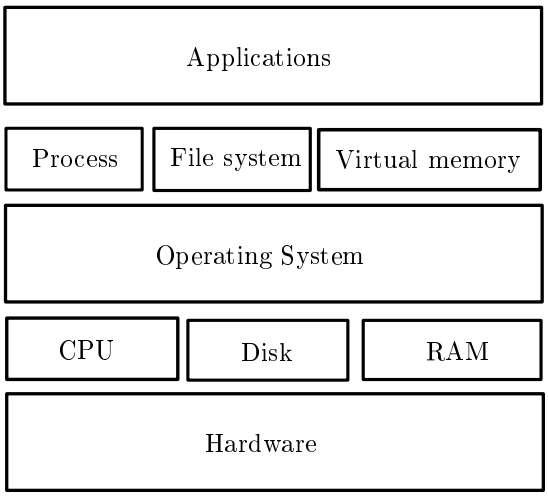
While the user is interacting with the device, we can switch between processes. However, this continues to add complexity, as there are now lots of simultaneous jobs with multiple sources of new jobs (i.e. people can start new jobs), yet, importantly, **interactivity is restored**.

Chapter 2

The Kernel Abstraction

Lecture 2: Threads

What does an OS do? It creates **abstractions** to make hardware easier to use. Further, it manages **shared** hardware resources. Operating systems use a variety of abstractions:



Recall the complexity issues we've faced while designing an OS: multiple users, programs, I/O devices, etc. We can manage this complexity via:

- **Divide and conquer**
- **Modularity and abstraction**

Definition 2.0.1 (Process). A **process** is the OS abstraction for **execution**. It's also known as a **job** or **task**.

This is advantageous simply due to the alternative if we were using strictly hardware.

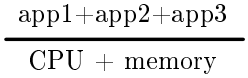


Figure 2.1: The interface hardware provides.

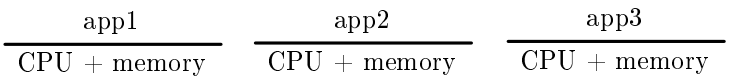


Figure 2.2: The interface that OS provides.

A process is simply a *running program* and is named using its **process ID (PID)**. It contains all the state for a running program.

Definition 2.0.2 (Program). Programs are static entities with *potential* for execution.

Appendix

Appendix A

Additional Proofs

A.1 Proof of ??

We can now prove **??**.

Proof of ??. See here. ■