Intro to Operating Systems

Andrew Quinn

Learning Objectives:

1. Why do we need operating systems?
2. What do Operating Systems do?

Announcements:

1. Bring your laptop to class.
2. About you survey due Friday.

HISTORICAL CONTEXT

Mainframes were the first computers. They were:

- Extremely large.
- Extremely expensive.
- Managed by people, called operators, to make sure that everything went smoothly.
- Batch processing systems:
  - One program runs at a time.
  - Programs run to completion.
- Note: Mainframes are still around today! Although, they look very different.

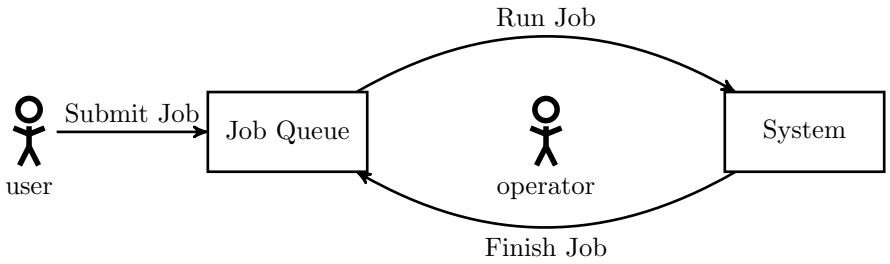


Figure 1: General workflow of an operator-managed batch processing system

Writing code on a mainframe was just as hard as writing code is today. So, the community developed libraries to make programming a mainframe *easier*. In essence, they applied *modularity* to make their lives easier. What are some things you might want to modularize to make programming a system easier?

- File Input and output (`fopen`, `fclose`, `read`, etc.).
- Memory management (`malloc`, `free`, etc.).
- Concurrency primitives (`locks`, `semaphores`, etc.).
- Networking code.

Figure 2a provides a high-level system diagram for these earliest systems.

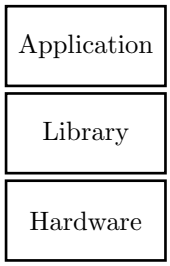
However, there are a few major issues that arise in this setup. First, executing a single job at a time is very inefficient from a utilization stand-point. Recall that a computer system is up of many different components—CPUs, Memory, secondary storage (e.g., Disk, Hard-drives, etc.), keyboard, etc. When an application uses a *peripheral device*, such as secondary storage or a keyboard, the CPU and memory of the system are idle. Remember, these machines are *very* expensive, so we do not want to waste their precious computing time. Note that the shift towards interactivity in computing, where a user interacts with a computer during a program, exacerbates this issue. Imagine the entire system waiting for each keystroke of the user—how inefficient!

OPERATING SYSTEMS

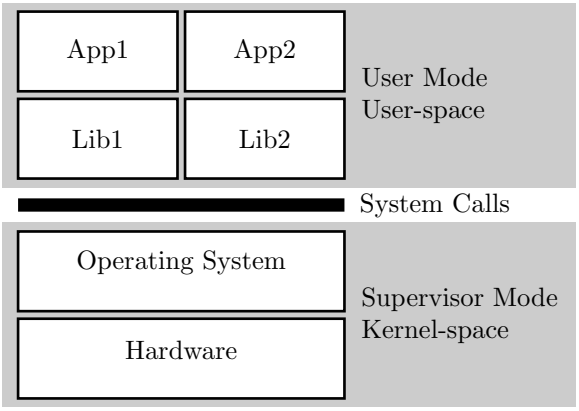
The idea that designers came up with in the late 60s and 70s was *Multi-programming*—what if we let multiple applications execute concurrently on the machine! This would solve the efficiency issue since the

Aside: Batch Processing

Batch processing remains common in computing today, although human operators that manually manipulate job queues are a relic of the past. Today's batch processing usually involves very large computing tasks that span multiple machines. Researchers and practitioners have developed many computing frameworks (e.g., Map-Reduce, Spark, etc.) to help manage these large-scale tasks. Are these distributed batch-processing frameworks *distributed operating systems*? Many in the research community would say so.



(a) System Diagram for Early Mainframes.



(b) System Diagram for Today's Operating Systems.

system could overlap the use of peripheral devices with the use of the CPU and memory. The issue is that this would be *way* to complex for a human operator to manage.

And thus, we have operating systems. Like the libraries from before, operating systems aim to make systems *easier to use*. Unlike libraries from before, they ensure that multiple applications can *safely* run over shared hardware resources by acting as a *resource manager*.

MAKING SYSTEMS EASIER TO USE

Operating Systems provide *system calls*: APIs that an application can use to request that the OS do something on the application's behalf. Sometimes called a standard library. System calls provide many services:

- Concurrency: provide interfaces for safe shared access to resources. We will review these; think back to CSE 130.
- Memory: provide interfaces for an application to allocate memory resources.
- Persistency: implement operations for saving data persistently. We will discuss persistency in detail in this course. Specifically, how does an operating system:
  - Mechanism: Ensure that a process's data is not lost during a crash.
  - Mechanism: Efficiently use device space even while a file grows and shrinks.
  - Policy: use device space so that accesses are as fast as possible.

In addition to providing system calls, operating systems make a system easier to use by *Virtualizing* resources (remember CSE 130?). The idea is to provide the illusion that a system has more resources than it actually has. The end-goal is to provide each process with the illusion of isolated and independent CPUs/memory, even though all processes share the same resources. This is brings us to the operating system's other role...

A RESOURCE MANAGER

Operating systems are often seen as the manager of all resources in a system. Doing this requires *mechanisms* that ensure that resources can only be allocated by the operating system and *policies* that ensure that the

Aside: Mechanism vs. Policy

You'll note that we've drawn a distinction between mechanisms and policies here. Separating mechanism from policy is a *design principle* that you'll see throughout operating systems; we'll be sure to call it out when applicable throughout the course.

operating system provides resources fairly to all applications.

Let's briefly discuss the mechanism that OSes use to prevent applications from directly accessing resources. Today's hardware provides multiple privilege domains, called different *modes* or *rings*. Certain instructions can only be executed when the system is in *supervisor mode* (ring 0), which is the mode in which the operating system executes. Executing these instructions while in *user mode* (ring 3) will cause an exception. In addition to separating the mode of execution, operating systems also separate the memory of the system into different *spaces*. Memory that is related to the operating system is said to reside in "*kernel-space*", while memory that is related to a user program is in "*user-space*". Colloquially, people often conflate user-space and user mode and say things like "this code runs in users-pace (or, user-land)".

An operating system's system calls (see above) are not like the procedure calls that you've used in the past. Instead, an operating system's system calls are actually implemented as special interrupt instructions (called *traps*), which cause the program to enter into supervisor mode and execute a previous configured *trap handler* routine. We'll talk more about how this works soon.

One interesting note: when past professors told you that you were executing system calls in past classes (e.g., 13S or 130) *they were lying to you*. You have almost certainly only ever executed the libc function call that wraps underlying system calls. It is a subtle difference; but, an important one.

DESIGN GOALS

Operating systems have many **design goals**:

- Protection: prevent applications from impacting each other, intentionally or unintentionally.
- Security: prevent applications from subverting OS-provided protections.
- Fairness: ensure that all applications receive a fair share of resources
- Performance: ensure low overhead.
- Reliability: do not fail. Maybe even make it so programs fail less often?
- Energy efficiency: Require little energy to operate.
- Support diverse environments (e.g., distributed systems, mobile systems, real-time systems, etc.)