



Topic Notes: Introduction and Overview

Welcome to Operating Systems!

What do you think of when you talk about an operating system? (“I installed a new operating system”, “Windows is my least favorite operating system”, “That must be a bug in the operating system”)

What do you expect to learn in a course about operating systems?

Operating system topics are always in the news – there are daily developments in the operating system world. Things change quickly. This course is partially reinvented each time around, though the concepts remain similar.

Where This Fits In

You learned high-level language programming in your introductory and data structures courses.

You learned about hardware and assembly language in 220 or an equivalent. You have a notion of how to get from circuits to CPUs and memory.

A course in compilers and/or programming languages would teach you about how high-level languages let you program the hardware in a more convenient way.

Many of the things that fit between those (compiled) high-level language programs and the hardware are topics for this course.

So what do you need to do to go from the basic hardware you studied in those courses to the multi-user systems we are used to on modern computers?

A computer system is made up of a collection of resources, such as a processor, memory, disks, a keyboard, printers, network interfaces.

The operating system attempts to regulate the use of these resources for efficiency, fairness when multiple users or processes want to use them, and safety to make sure multiple users don't interfere with each other.

We will consider the operating system from two points of view: users and systems.

To a user, the operating system provides a more convenient interface. This allows the user to log in, manipulate files and run programs in a reasonably intuitive and convenient manner. Meanwhile, it provides protection of the user's data from unauthorized access, and ensures that the user is allocated a fair share of the computer's resources.

The user would like to do things like run programs and read and write files and communicate over the network without worrying about the details of what goes on at the lower levels. The overriding

theme here (as in so much of computer science) is *abstraction!*

To a system, the operating system provides safe and efficient access to the actual hardware. The operating system tries to share resources when safe to do so and restrict access when necessary.

We can think of the operating system as a big resource manager.

Examples of Problems

Many important ideas in Computer Science arise in the study of Operating Systems:

- There are 3 users, each wishing to use the computer at the same time. Each has a program that needs to run for 5 minutes. Is it better for the system to run the first to completion, then the second to completion, then the third? Should it switch among them once a minute? Once a second? Once a millisecond? After every instruction? Which makes the most efficient use of the system's resources? Which makes the users happiest? Does the answer depend on what the program is doing for those 5 minutes?
- Suppose we have two programs, one that generates output values that are used as inputs to the other. We either do not want to or cannot have the second program wait for completion of the first before it starts to work: the programs must execute concurrently. How can we manage this situation if values may be generated by the first more quickly than they can be processed by the second? Or vice versa? Or if the situation changes over time?
- Suppose we have a one-lane bridge. How can you most efficiently manage traffic across the bridge? This sounds simple enough, but the best answer is not always clear. Some ideas include: just let people take turns, have a traffic light that alternates turns, have a pair of flaggers, give one direction precedence. But there are many potential problems: what if cars come in on both sides and meet in the middle? Someone's going to have to back up. The traffic light can be pretty annoying if you're stuck at the red and you wait and wait and don't see anyone come the other way. This is an unnecessary wait.
- Suppose we have a shared printer. If multiple people want to print at the same time something has to make sure the jobs don't get intermingled.
- The one-lane bridge example is one example where a deadlock can arise. It can come up in more subtle ways. Think of this like gridlock. Everyone is waiting for someone else to do something before they can proceed. No one gets anywhere.

In a computer system, this could be a situation where two users need exclusive access to two resources.

A simple example is two users who need to copy tapes. The system has two tape drives, and a tape drive is necessarily granted to one user at a time. User 1 requests a drive and gets it. User 2 requests a drive and gets it. User 1 requests a second drive, but must wait until User 2 finishes with the one he has. User 2 requests a second drive, but must wait until User 1 finished with the one he has. Uh oh. We can think of this as two antagonistic users, but

even “friendly” users may not be aware that they are holding a resource that is preventing the other resource they need from ever becoming available.

- Suppose we have a collection of processes that are cooperating on a task. They need to coordinate. We’ll look in some detail at process synchronization both from the point of view of algorithms that use it and what hardware and operating system support is needed to make these kinds of programs correct and efficient.
- If you have a disk attached to the computer, and several users of the computer, how do we organize data on the disk so it
 - is convenient to write and read,
 - is organized efficiently (quick to access, not a lot of wasted space), and
 - enforces appropriate protection on the files, to make sure users can’t read or worse yet modify or delete the files that belong to some other user, but preserving the ability to share data effectively when appropriate?
- Suppose we have a network of computers – like the School of Science Labs.

If we have a collection of computers shared among a collection of users, how can we devise a system where the resources are efficient and easy to use, yet secure?

An approach that works well in our lab, where the systems typically have only one user at a time – the one who is sitting in front of a given computer, might not work well in a lab where the computers are used for long, CPU intensive jobs, such as graphics rendering or scientific computation.

Most of the problems that come up are not specific to a given operating system or type of computer. In fact, many of these same problems have been evident from the earliest historical systems right up to current systems.

An interesting thing about operating systems, and in fact much of computer science, is that an important “historical example” is often no more than a few decades old.

Some of the historical examples will likely be very familiar to you. What old systems were in your childhood? Some from mine include the Commodore 64/128.

And even when you might think some of the issues that were important on your (or your parents’?) old Commodore 64 or Apple][keep coming back in your tablets and smartphones or other smaller-scale special purpose computers.

The course is not about which is the best operating system (though I’ll make my opinions known from time to time and you can do the same). One thing we’ll see as we go is that different systems have strengths and weaknesses that make them appropriate in different situations.

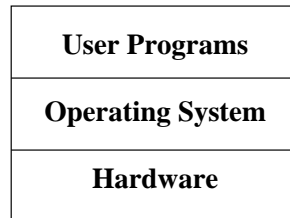
When we compare approaches to a particular problem, trying to determine which approach is better, the answer will often be “it depends.”

We use Unix-like operating systems as our model, but modern Windows systems have most of the same ideas underneath. Macs are really just Unix systems with a shiny user interface.

What is an Operating System?

One possible definition of an operating system (from our textbook): “a program that acts as an intermediary between a user of a computer and the computer hardware.”

You can find similar definitions elsewhere.



The term *user* is (obviously) the person using a computer or and *user programs* are the programs being run on their behalf. It is the user program that makes use of the resources of the computer as managed by the operating system.

At its most basic level, the operating system is a low-level program, which talks directly to computer hardware on behalf of user programs. The operating system *kernel* is the program that stays running on the computer at all times. The kernel decides what user programs can do and when they can do it, and provides facilities to make it more convenient for them to do it.

We often think of the “operating system” as including a lot more than the kernel – system programs and application programs as well. These utility programs, editors, office suites, etc., are not part of the kernel. We will consider mostly things at the kernel level, but we will look at some system programs.

What is the hardware? It could be a small single-user PC, a workstation, mainframe, or a super-computer. It will contain one or more CPUs, main memory, disk resources, and I/O devices. It could also now be something much smaller – an embedded system or a handheld computer.

Much of operating system theory focuses on large, multiprogramming systems – multiple users, multiple programs, with time sharing.

As desktop and portable computers have become more powerful, the issues that were formerly only the concern of larger systems have become important on the smaller scale. And issues that were formerly the concern of single-user desktops arise in tablet computers and smartphones.

We consider three main goals and functions of an operating system:

- to facilitate the use of the hardware by user programs (by providing convenience, efficiency, and flexibility)
- to allocate system resources (CPU, memory, I/O, file storage)
- to enforce security (controlled access to files, hardware resources, authentication)

These goals are often competing!

- The user of a system wants: convenience, ease of use, reliability, safety, speed
- The system wants: ease of design, implementation, maintenance, also flexibility and efficiency

Common Operating System Components

Many of these components involve all three of the goals/functions we listed.

- *Command-line interpreter (CLI)*
 - Think: UNIX command line or an MS-DOS prompt.
 - Often referred to as the *shell*.
 - A CLI provides a convenient and efficient way to access files and run programs on the system.
 - Note that a graphical windowing system is often just a way to issue the same commands without having to type their names or perhaps without even knowing what they are.
- *Process Management*
 - A “process” is a program in execution.
 - The operating system is responsible for creation, deletion, scheduling, communication.
- *Main-memory Management*
 - allocation: who gets to use which memory and when.
 - protection: make sure only those processes that are supposed to have access to a certain part of memory are granted access.
- *File Management*
 - creation, deletion, reading, writing.
 - file system and directory structures.
 - mapping files to hardware: convenience, efficiency, protection are key functions of the operating system.
- *I/O System Management*
 - safe, efficient, and convenient access to the wide array of devices.
 - device driver interface.

- buffering.
- *Secondary Storage Management*
 - storage allocation, including free space management.
 - disk scheduling: efficiency improvements.
 - caching: efficiency.
- *Networking*
 - another device to manage, but one with high-speed information flow.
- *Protection System*
 - specification and enforcement of access controls.
- *Error Detection*
 - hardware or user program errors.
 - deal with them gracefully: terminate only one program rather than the whole system if possible.
 - detect serious hardware errors

Mechanism vs. Policy: mechanisms are provided to perform tasks, policy determines what will actually be done. Separation of mechanism from policy is an important principle. Allowing policy to be changed later allows maximum flexibility.

Some History

We take a brief look at the historical development of computer systems from the point of view of operating systems.

Very Early Systems

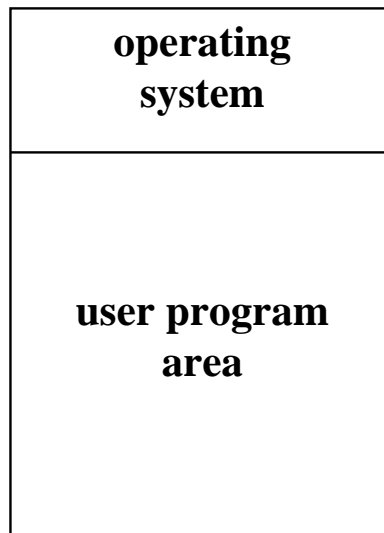
Here, we had just one user at a time. Everyone involved is an expert. A user would sign up for a block of time to go program the computer (possibly involving plugboards) and run the program.

This was very expensive. The machines were huge and expensive and they would sit there idle quite a bit while waiting for people or card readers or other very slow things.

Early Mainframe/Batch Systems

We still have one job in the system at a time. But the jobs are “batch processes” – non-interactive process. You set everything up beforehand, wait your turn, your program runs, and you get your output.

System memory is organized very simply:



Card Reader (input) → Memory/CPU (computation) → Line Printer (output)

The big problem here is that card readers and line printers are slow - what is this expensive CPU doing while the card reader is loading a program or while the output is being printed? It's idle. Not good.

With disks able to provide direct access to information, the operating system of batch computers was able to use the faster disk (in relation to the card readers and printers anyway) to *spool* upcoming jobs and output. (Spool means Simultaneous Peripheral Operation On-Line). This means that the CPU can stay busier – still better *CPU utilization*.

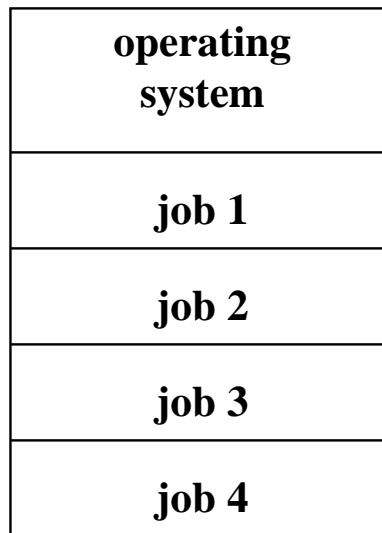
But, we still have some idle time for the CPU. Disk is much slower than the CPU, both then and today. When a job needs access to the disk (or any other I/O) while starting up, during the run, or when writing its output, the CPU is still idle or nearly idle. There's also the potential for infinite loops in programs. If some user's program goes into an infinite loop, it would probably have to be stopped manually.

So we move on to...

Multiprogramming Batch Systems

Here, we have multiple jobs in the system. The CPU can execute any job that is in memory.

System memory now needs a bit more complicated organization:



When one job needs to access I/O or anything else that would cause the CPU to be idle, another job is selected to run while the I/O request is serviced.

This brings up some new issues that we will discuss later in the semester:

- A resident monitor program needs to coordinate all of this.
- The monitor program is **in charge**. It's allowed to do things that regular user programs can't do.
- This is the start of the dual nature of operating systems – monitor vs. user.
- I/O devices must be able to operate without the CPU, as the CPU would be busy with another job when I/O is taking place.
- I/O request must be made through *system calls* - not direct to hardware. Imagine two jobs both sending lines of output to the printer any time they wanted.

System calls have access to the hardware, whereas the user processes should not.

The user process doesn't know how a system call works, just how to call it and what it's supposed to do.

To open a file, for example, a user mode program makes a system call which runs in monitor/kernel mode, which does the actual I/O.

The monitor program can then ensure safety of the I/O request as well as hand off the CPU to another job while the I/O request is processed.

- The monitor needs to choose which job to run next when one job makes an I/O request or terminates. This is *CPU scheduling*.
- The system needs to make sure that job 1 can't read or interfere with job 2's memory. *Memory management and protection*.

The dual mode operation requires some hardware support. The system needs to be able to distinguish things that users are allowed to do (*unprivileged* operations) and things that only the system can do (*privileged* operations).

This requires a *mode bit* or something similar to control whether the CPU should be allowed to perform privileged operations. The kernel needs to set this to “user mode” before calling user code and user code that needs to do anything that requires “system mode” must do so through a system call.

If a user program tries to perform a privileged instruction when in user mode, it will not be allowed – will *trap* to the operating system.

We’ll see more on this as we continue.

The ideas of traps and *interrupts* become important as well. If a user program does something illegal (regardless of whether it is malicious) it should not crash the system – just “trap” to the operating system.

The operating system can do something appropriate by printing an error or killing the process, or maybe fixing a problem that caused the trap and allowing the process to continue.

But.. there are still significant limits...

Programs are not guaranteed to be perfect and could have infinite loops. And since the users are probably not closely watching their programs execute in a batch environment, it might not be noticed quickly.

We also would need to make sure that a severe program error (bad pointers, division by 0) would halt the user program but not the entire system, as other programs would also be in progress.

But.. what about interactive processes?

Time-sharing Systems

The batch systems do not allow user interaction with the program. This is obviously not sufficient in all cases, so operating systems evolved to allow a more general *multitasking*.

Users can run *interactively* using a keyboard and a terminal display (or windowing system in a modern equivalent). A user typing at the keyboard is *much* slower than a computer.

People multitask all the time (for better or for worse). Consider for example lawyers who take multiple cases to keep themselves occupied.

This is done by switching among user processes automatically and transparently. The goal for a CPU scheduler with interactive processes is to ensure that each interactive user gets a turn on the CPU quickly to achieve a good *response time*. We also need to be able to switch among processes quickly with an efficient *context switching* mechanism.

Such systems depend heavily on the idea of interrupts, which allow devices or the operating system to take control of the CPU even when it is executing a user process.

The considerations are similar to what happens when you multitask. How many tasks can you

switch among before getting overwhelmed? Is it better to work on each for a few seconds at a time, a few minutes, or a whole day? How often do your tasks get interrupted (by texts, tweets, facebook, or emails, for example)?

Many of the concepts we'll talk about this term are present in multiprogramming and time-sharing systems.

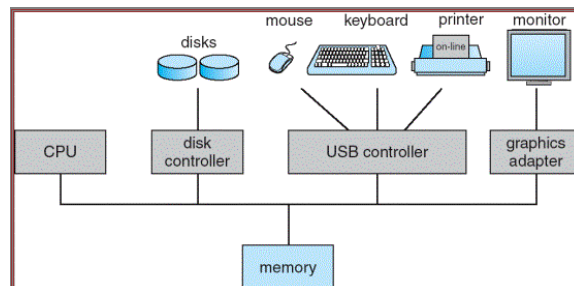
Personal Computers

PC's appeared when computers became cheap enough that a single user to have one dedicated for his or her own use.

Such a system has different needs. CPU utilization is generally not the biggest concern, since there are no other jobs waiting to execute. User convenience and responsiveness are the top concerns. Having just one user means protection and security are not important.

As desktop computers became more and more powerful, the way people used them evolved into a *workstation* model. There are multiple processes, remote access, meaning many of the concerns originally only dealt with by multiprogrammed and time-shared systems are now addressed by PC operating systems.

As you know from previous courses, modern computers can be viewed as a collection of components connected by a bus.



This is the kind of system we will spend most of our time considering.

Parallel and Distributed Systems

Things really get interesting when we introduce multiple CPUs. They might be in the same system, or they might be distributed across a number of systems. Or perhaps we have a whole collection of uniprocessor systems that might make sense to use or manage as a group.

Modern supercomputers can include hundreds of thousands of processors, with combinations of shared and distributed memory. Modern desktop and portable computers contain multiple *cores*, which are basically multiple CPUs from the perspective of the operating system.

Many operating systems issues come up in such systems, and we'll talk about those throughout the semester.

Real-time systems

We will not focus on real-time systems, but will mention them from time to time. These are used for systems that perform tasks such as reading critical sensor values or controlling some device. The devices could range from kitchen appliance controls to Mars explorer robots.

Hard real-time systems for critical applications have very rigid restrictions: e.g, automated vehicle (car, airplane, spacecraft) control. Missing a deadline is potentially disastrous.

Soft real-time systems are less critical - visualization, robotics, multimedia.

Handheld systems

This is a relatively new and quickly evolving category of computer. It started with PDAs, and cell phones with limited capabilities but now includes advanced smartphones and tablet computers. Many of the issues that have trickled down from the multiprogramming and time-shared systems to the personal computer and workstation world are now starting to get down to this level. These have relatively slower processors, smaller displays, limited memory and non-volatile storage.