Computer Science 322
Operating Systems
Mount Holyoke College
Spring 2008

Topic Notes: Introduction and Overview

Welcome to CS 322!

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?

OS topics are always in the news – lots of current developments in the OS 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 211. How to get from circuits to CPUs and memory.

Compilers and programming languages 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.

In 211 (and maybe 324 for some of you), you learned about things like interrupts, traps, exceptions, caches, virtual memory. These will all be important here, and we'll revisit those as we go along.

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 the point of view of users and systems.

To a user, the OS 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 running programs and reading and writing files and communicating over the network without worrying about the details of what goes on at the lower levels. *Abstraction!*

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

We can think of the OS 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?

- Suppose we have two programs, one that generates output values that are used as inputs to the other. How can we manage the situation where 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? Sounds simple enough. Ideas: just let people take turns, have a traffic light that alternates turns, have a pair of flaggers, give one direction precedence. Potential problems: 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 OS 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 for people to make use of it
  - have it be an efficient organization (quick to access, not a lot of wasted space)
  - enforce 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 can share data effectively when appropriate?
- Network of computers – like the CS Lab.

  If we have a collection of computers shared among a collection of users, how do we set things up so again things 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 OS or type of computer. In fact, many of them come up from the eariest 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 PDAs or cell phones or other smaller-scale special purpose computers.

The course is not about which is the best OS (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're comparing approaches to a particular problem, and the question comes up as to 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.

Lab 0

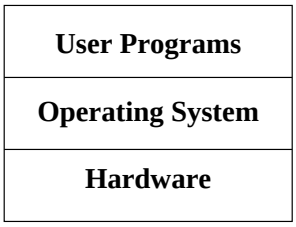
We begin with a lab assignment...

What is an Operating System?

I am going to go quickly through highlights of the things in the first two chapters of the text. You'll need to read them – a lot of it should be familiar. I will end up talking about many of the topics there today and as they become important for upcoming topics.

Possible definition (from our book): "a program that acts as an intermediary between a user of a computer and the computer hardware."

You can find similar definitions elsewhere.



At its most basic level, the OS 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.

We may think of the OS as including a lot more than the kernel – system programs and application programs as well.

What is this hardware? Depends... Could be a small single-user PC, could be a minicomputer, mainframe, supercomputer. One or more CPUs, main memory, disk resources, and I/O devices. Could even be something smaller – an embedded system or a PDA.

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

As desktop and portable computers get more powerful, the issues that were formerly only the concern of larger systems become important on the smaller scale.

Goals/Functions of an OS:

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

These goals are often competing!

- Competing design goals:

  - User wants – convenience, ease of use, reliability, safety, speed
  - System wants – ease of design, implementation, maintenance, also flexibilty and efficiency

The definition of the **user** depends. It may be a **user program**, which is the user of the resources of the computer as managed by the operating system, or the user of the computer, who runs those programs.

Many additional programs, often lumped in as part of the "operating system" such as utility programs, editors, office suites, etc., are not part of the kernel. They may or may not be part of the "operating system" depending on how you define it.

Common Operating System Components

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

- **Command-interpreter** - think UNIX command line or MS-DOS prompt. Deals with all of the above. Really, a windowing system is just a way to issue the same commands without knowing what they are.
- **Process Management** "process" is a program in execution. OS responsible for creation, deletion, scheduling, communication
- **Main-memory Management** allocation, protection
- **File Management** creation, deletion, directory structures, mapping files to hardware
- **I/O System Management** device driver interface, buffering
- **Secondary Storage Management** free space management, storage allocation, disk scheduling, caching
- **Networking** another device to manage – high-speed information flow
- **Protection System** specification and enforcement of access controls
- **Error Detection** hardware or user program 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.

Examples of Operating Systems

- Unix and friends
  - BSD (Sun OS 4.x, DEC Ultrix, FreeBSD, OpenBSD, Darwin/OSX)

  - SysV (Solaris, Irix)
  - OSF/1 (DEC/Compaq Unix/Tru64, AIX, HP/UX)
  - Linux
- PC systems
  - DOS (MS,PC), Win 3.x
  - Win 95, Win 98, Win ME
  - Windows NT, Win2K, XP, Vista
  - OS/2
  - Mac OS N, N$\leq$9
- Multiprogramming systems for Mainframes
  - VMS (DEC VAX/Alpha)
  - IBM MVS
  - IBM VM
  - IBM OS/360, IBM OS/390
  - MULTICS
- PDA
  - PalmOS
  - Windows CE
- Virtual Machines
  - JVM
  - Video game console emulators
- Real-Time Systems
  - VxWorks
  - QNX
- Others
  - Mach
  - CP/M
  - BeOS

Some History

Very Early Systems

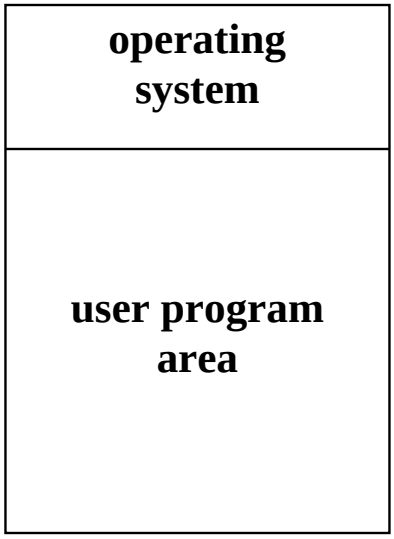
One user at a time. Everyone involved is an expert. Sign up for a block of time to go program the computer (possibly involving plugboards) and run the program.

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

Early Mainframe/Batch Systems

One job in the system at a time. A "batch process" is a non-interactive process. You set everything up beforehand, it runs, you get your output.

System memory:



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

Big problem here - 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.

When disks provided direct-access, 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). CPU can stay busier – better **CPU utilization**.

But, we still have some idle time for the CPU - disk is still much slower than 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

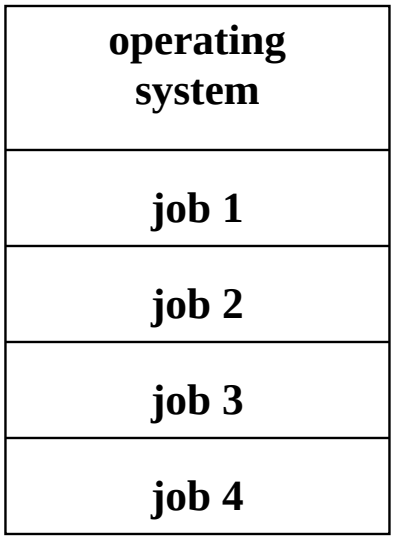
loops. If some user's program goes into an infinite loop, it would probably have to be stopped manually.

So we move on to...

Multiprogramming Batch Systems

Have multiple jobs in the system. The CPU can service any job that is in memory.

System memory:



When one 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:

- Some resident monitor program needed to be there 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 OS – monitor vs. user.
- I/O device 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.

  Information hiding, encapsulation, all that good stuff from OOP.

  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 actually 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.

- Need to choose a job to run next when one job makes an I/O request or terminates. Need for **CPU scheduling**.
- Need 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 and things that only the system can do.

This requires a **mode bit** or something similar. 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 be done 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 OS.

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 OS.

OS can do something appropriate by printing an error or killing the process, or maybe just fixing up whatever caused the trap.

But.. there are still significant limits...

We can still have infinite loops in programs and that's not good. And since the users are probably not watching as closely, it might not be noticed as 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.

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 **multitasking**.

Users can run **interactively** - I/O can include 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 do this all the time. We work on multiple things at once. The book mentions a lawyers who take multiple cases to keep themselves occupied.

This is done by switching user tasks or **processes** transparently. This changes what it important for CPU scheduling. We want each interactive user to get a turn on the CPU quickly – good **response time**. Need to switch among processes quickly – **context switching**.

Such systems depend on the idea of **interrupts**, which allow devices or the operating system to get the attention of the CPU from a user process.

You can think of each of these ideas in your own multitasking. 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 (that e-mail inbox, for example)?

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

Personal Computers

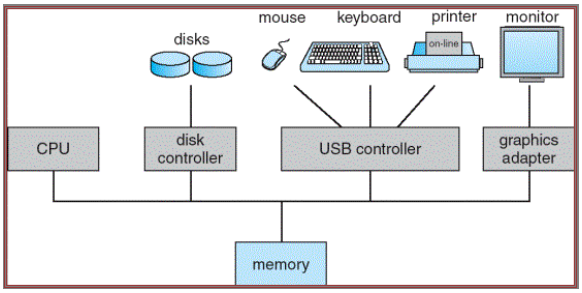
PC's appeared when computers became cheap enough to be affordable for a single user to have one dedicated.

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. One user means protection and security are not important.

But, as the desktop computer gets more powerful, many of the concepts of the multiprogramming OS's are working their way down into the PC world.

The time-sharing and personal computer categories have merged into the modern workstation idea.

As you know from CS 221, 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

What happens when we start having 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.

We have a number of examples here. There is a small cluster here that we will use from time to

time.

Modern supercomputers can include thousands of processors, with combinations of shared and distributed memory.

Many OS issues come up in such systems, and we'll talk about those as we go forward.

Real-time systems

Used for things like reading critical sensor values or controlling some device. The devices could range from kitchen appliance controls to the Mars explorer robot.

Hard real-time systems for critical applications - automated vehicle (car, airplane, spacecraft) control

Soft real-time for less critical - visualization, robotics, multimedia.

Handheld systems

This is a relatively new category. PDAs, cell phones. 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.