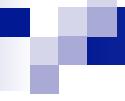


What is an OS?

ICS332
Operating Systems



What is an Operating System?

- What would you say to your non-CS-major friend asking this? Anybody?

What is an Operating System?

- What would you say to your non-CS-major friend asking this? Anybody?
- Typical “ok” answer: It is the software layer between the applications and the hardware because the hardware would be too difficult for users to use
- Typical “sort of ok” answer: It is “all the code that I don’t have to write” when I develop software (not quite right since there are tons of non-OS libraries whose code you don’t write either)
- Typical **wrong** “Big Brother / Eye of Sauron” answer: It is the one program that runs at all times and watches everything
 - This is a pervasive but **misled** view: **the OS is not a running program**
 - Although it starts programs
- Better answer: it’s code that gets invoked time and again
 - And there is no need to “reserve” one CPU/core for it (something you will hear once in a while)

What is an Operating System?

- This is not such a simple question
- An OS is a complete software system that manages access to hardware and makes it possible to run software applications on that hardware
- A core component of the OS is called the **kernel**, which is **code and data structures** in charge of managing hardware resources
 - And is not a running program
- The kernel acts as layer between application software and the hardware, and it performs virtualization...

The OS Virtualizes

- Conceptually, the main role of the OS is **virtualization**
 - The first of the “three easy pieces” of our textbook
- The term “virtualization” is used in many contexts
 - The Java Virtual Machine (JVM)
 - Virtual Machines that one would use in the cloud
 - More on this much later in the semester
- In the context of OSes we mean two things:
 - Resource abstraction
 - Resource allocation

Virtualization: Abstraction

- The OS is a **Resource Abstractor**
- It defines a set of **logical resources** that correspond to **hardware/physical resources**
- It defines **operations** on these logical resources
- Typical examples:

Physical	Logical	Operations
CPU	Running Programs	start, terminate, pause, ...
Memory (SRAM, DRAM)	Data	allocate, free, read, write, ...
Storage (SSD, HDD, Tapes,...)	Files	create, delete, open, read, write, ...

Virtualization: Allocation

- The OS is a **Resource Allocator**
- It decides **who** (i.e., which running program) gets **how much** (e.g., CPU cycles, bytes of RAM, bytes on disk) and **when/where**

Resource	Example resource allocation decisions
CPU	Should the currently running program keep going? Which program should run next?
Memory	Where in RAM should a running program's data be? Should a program be allowed to use more
Storage	Where on disk should pieces of files be stored?

Virtualization: Why and How?

- Why virtualization?
 - **Reason #1:** To make the computer **easier to program**
 - There was a time “before OSes” in which the programmer had to know a lot about the insides of the computer
 - Think how easy it is today to write code without understanding/knowing anything about the hardware
 - **Reason #2:** To provide each program with the **illusion** that it is alone on the computer, going through its fetch-decode-execute cycle
 - When you develop a program, you don’t think of what other programs will be running when your program will run!
 - And yet many programs run at once
 - **Reason #3:** Make sure that **bad things don’t happen**
 - A running program cannot corrupt another running program’s data
 - A running program cannot crash the system
- How does the OS do it? That’s what a lot of ICS332 is about!
- Looking at early OSes, we can see what happens when there is no or not enough virtualization...

No Memory Virtualization?

- Example: Blue Screen of Death (BSOD) in Win 95/98
 - We've all seen it, and nowadays it means there is a bug in the Windows kernel or in a device driver (i.e., NOT in your code)
 - But in the “old” days (Windows 95/98), any application could bring about the BSOD
 - This is because memory was not (sufficiently) virtualized, and your code could, due to a bug, write to pretty much any RAM address, and thus overwrite/corrupt kernel data structures
 - Due to the difficulty of building on top of DOS, of being backward compatible
 - All of this changed with Windows NT (2001 for home users)
 - Which is a turning point when the Windows OS became more “respectable”
 - Note that, at that time, UNIX systems had had strong memory protection for years!
 - And this is why, thanks to virtualization, nowadays our Windows computers don't crash as much as they used to

No CPU Virtualization?

- Example: CPU hogging in Mac OS 9
 - Nowadays, we expect that our machines will never be frozen because some application is running
 - But in older OSes, like Mac OS 9, there was no virtualization of the CPU
 - The idea is that applications will be nice and would call some “yield control back to the kernel” function every now and then
 - Called “cooperative multi-tasking”
 - As a result, an application could hog the CPU indefinitely
 - Either because it wasn’t “nice”
 - Or just because it had a bug
 - This led to the infamous “spinning beach ball” for the whole system (not just for an application)
 - This was fixed in Mac OS X (2001), which was a turning point for Mac OS
 - And here again, UNIX had CPU virtualization for years at that point!

No Device Virtualization?

- Example: Game development for DOS (80's 90's)
 - Back then, if you had to write a game for DOS, you had to interact directly with the graphics and sound hardware
 - As a game developer, you had to write different code for different hardware components (so that many users could play the game)
 - This was a lot of work for everyone
 - e.g., developers had to read and follow the hardware spec of many possible graphics cards on the market
 - e.g., users had to tweak complicated config files
 - This was really bug-prone, meant that a game may not work on the most recent hardware, meant that games could conflict with each other, etc.
 - Device virtualization appeared in Win 95/98, which provided a device API that game developers could use
 - And again, UNIX had had that for decades at that point!
 - We'll look at reasons why older OSes (UNIX) were objectively better than subsequent ones (DOS, MacOS9)

The Three Easy Pieces

- Our textbook is called OSTEP: Operating Systems: Three Easy Pieces
- The three pieces are:
 - Virtualization
 - Concurrency
 - Persistence
- Let's talk briefly about concurrency and persistence....

Concurrency: Multi-Programming

- **Concurrency**: doing multiple things “at once”
- **Multi-programming** is the name of the OS’s capability to execute multiple programs concurrently
 - This is only feasible because the OS provides virtualization
- We take multi-programming completely for granted
 - which is why many of you likely had never even heard of the term
- Computers used to be used in “single-user mode”, where a program is truly alone until completion, and then another program is started, and so on ...
- This had several productivity drawbacks:
 - Your computer can do only one thing at a time
 - If the program is idle for a while (e.g., waiting for keyboard input, waiting for any I/O), then the CPU cycles are completely wasted
- OS advances made multi-programming possible, and we never looked back!

Concurrency Issues

- Due to multi-programming, a big issue has been **concurrency**, since the OS has to juggle many things “at the same time”
- It leads to deep/difficult/interesting issues within the OS
- Furthermore, nowadays most programs are also concurrent
 - e.g., for a single program to use multiple cores using multi-threading (ICS 432 is all about that)
- Therefore, concurrency is everywhere and is a constant theme in any OS course
 - Section 2.3 in our reading assignment talks about the main concurrency problem
 - If you find it a bit confusing, don’t fear, we’ll come back to this...

Persistence

- Persistence: the ability to store data that survives a program termination / a computer shutdown
- This is done by the file system
 - Typically considered part of the OS (which provides “file stuff” system calls)
 - Even though it is often developed independently from the core OS code



Conclusion

- Reading Assignment: 2.1-3. Section 2.4 starts with:

OSTEP

So now you have some idea of what an OS actually does: it takes physical **resources**, such as a CPU, memory, or disk, and **virtualizes** them. It handles tough and tricky issues related to **concurrency**. And it stores files **persistently**, thus making them safe over the long-term.

- Sections 2.1 and 2.2 show examples programs to illustrate virtualization, which I didn't discuss
 - We'll look at similar programs in future modules
- Section 2.3 is about concurrency and will likely be confusing for most of you
 - That's ok, we'll talk about concurrency in a future module
- Coming up next: the kernel