Introduction to operating systems

Kevin Wang

UNIVERSITY OF MICHIGAN

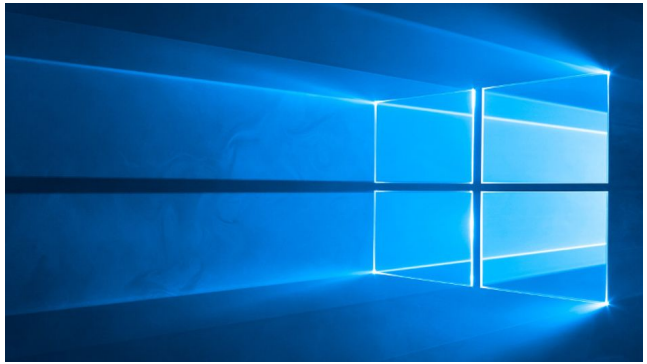
8 December 2023

About operating systems at my home institution – EECS 482

- Full course on OS is taught after intro to computer architecture
- More software-focused
- Very rigorous!
- We broadly cover:
  - Threading
  - Memory management
  - File systems
  - Networking/distributed systems
- Feel free to Google the course if you are curious about it

What is an operating system?

Windows, Mac, Linux...



What should I do if I want to make my own operating system?

e.g. an OS for an industrial control system?

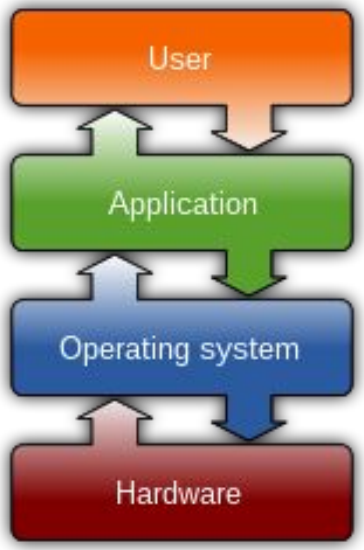
What kind of code should I be writing?

What responsibilities does an operating system have?

How does your computer know what to do...

- ...when the computer starts?
- ...when you want to run multiple applications at once?
- ...when you want to create / edit a file?
- ...when you plug in a USB drive?
- ...when you want to access the internet?

Operating systems are the layer between hardware and software

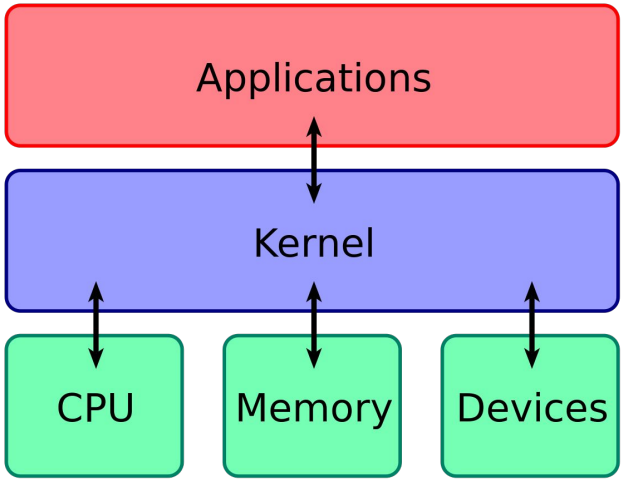


https://upload.wikimedia.org/wikipedia/commons/thumb/e/e1/Operating_system_placement.svg/165px-Operating_system_placement.svg.png

The kernel

The kernel is the “master program” at the heart of the OS

Controls basically every part of the OS



https://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Kernel_Layout.svg/1200px-Kernel_Layout.svg.png

Some of the things the kernel does:

- Starting up the computer
- File operations
- Integrating peripheral devices (mouse, keyboard, USB drives, monitors…)
- Lots more that we won't talk about

File operations are done by the kernel

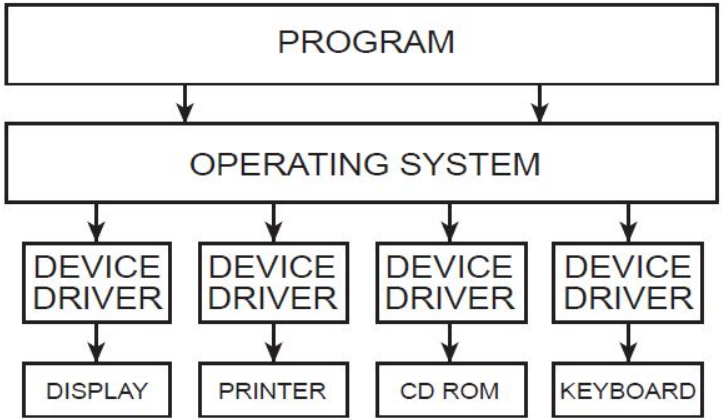
- The kernel has control over the file system
- The file system itself is a separate part of the OS

I/O is controlled by the kernel

- This is your printf, scanf, etc…
- …but also your keyboard, monitor, printer, and other inputs / outputs

OS lets you plug in external devices

- Ethernet / USB / optical drives work because the OS can interface with them
- As well as “internal” devices (graphics cards, wi-fi adapters…)
- **Driver:** software that lets the OS interface with an external device



Question: what kind of access do you think the programmer has to the kernel?

- Because the kernel is so powerful, access is restricted
  - Some mistakes with the kernel can only be solved by reinstalling the OS / a new kernel
  - Has your code segfaulted before? It might've been trying to access memory reserved for the kernel
- Becomes very interesting when connected to computer security!
  - Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges
    - "Hammer" a row of memory to make adjacent transistors flip
  - Stuxnet
    - NSA hacks using Windows zero-days to destroy 20% of Iranian nuclear centrifuges

Multithreading

Multithreading: utilizing the OS with multiple cores

- Core = CPU processor
  - Many modern CPUs will have multiple cores nowadays
  - Why? Moore's law no longer holds
- Thread = one independent sequence of instructions
  - You can have multiple threads per core, but it won't be faster unless you have multiple cores
- One form of parallelism / concurrent programming
  - Parallelism shows up outside of OS contexts too

What does this look like in code?

To use, `#include <pthread.h>` ('p' stands for IEEE POSIX standard)

```c
void *print_one()
{
    printf("1 ");
}

int main()
{
    pthread_t thread1, thread2;
    int i = 0;
    int iret1 = pthread_create( &thread1, NULL, print_one, (void*)(&i) );
    int iret2 = pthread_create( &thread2, NULL, print_one, (void*)(&i) );
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
}

```

Does the extra speed come for free?

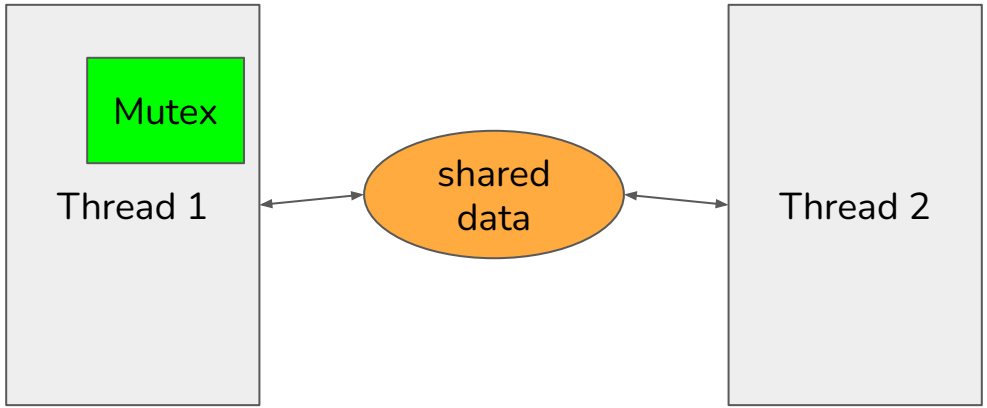
No – need to worry about concurrency problems

Without some form of control, many other thread **interleavings** are possible

Thread 3 – `printf("has ")`
Thread 5 – `printf("problems ")`
Thread 1 – `printf("Now ")`
Thread 6 – `printf(". ")`
Thread 4 – `printf("two ")`
Thread 2 – `printf("he ")`

For consistency, multithreading requires a system of locks

- **Mutex: mut**ual **ex**clusion mechanism that prevents multiple threads from modifying shared data at once
  - Thread must acquire a mutex **lock** before running



Thread 2 must wait for thread 1 to unlock (release the mutex) before continuing to modify shared data

There’s more we have to skip for now

Highly recommend learning this -
significantly changes your way of thinking
about software

Parallelism is good if you have a few big tasks that can mostly be done separately

- For example:
  - Quickly processing large files / chunks of data
  - Writing information to different files at once
- Outside of OS:
  - Loading webpage elements at the same time
  - Loading webpage elements whenever they become available over network (e.g. JS promises)
  - Anything with graphics / rendering / GPU programming
- Fun skill to pick up!

This also gets into processes

- A **process** is a program running on your computer
  - You can have multiple processes running, just like how you can have multiple threads
- Common question in systems engineering: parallelize with threads or processes?
  - Google Chrome tabs are all separate processes – same with Edge and other Chromium-based browsers
  - Internet Explorer was also multiprocess but in practice, not all tabs were separate processes
    - Part of the reason IE was bad – had the complications of multiprocess browsers without gaining many of the benefits

Virtual memory

The OS lets programs think they have access to all possible memory addresses

This is the easiest interface for developers writing their own applications

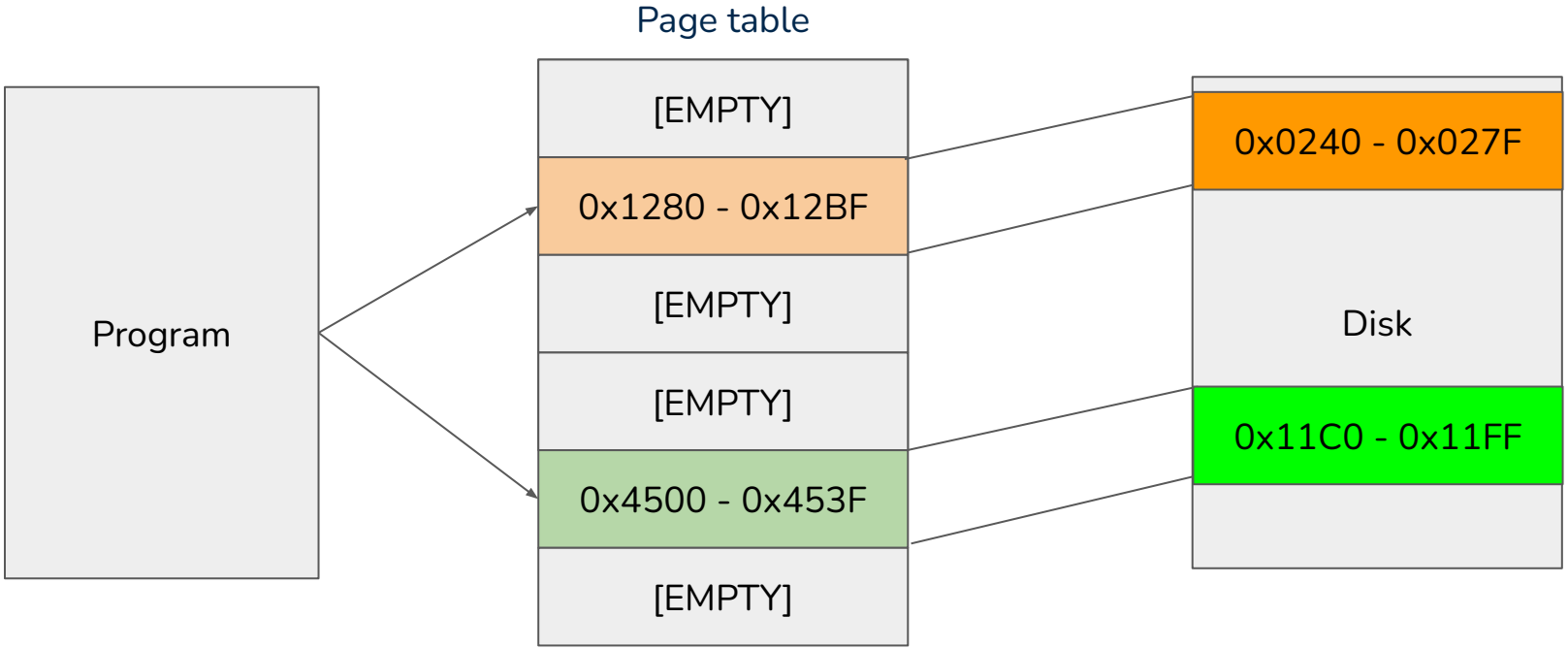
However, programs don't *actually* have access to all of memory

- Because…
  - …some memory is reserved for kernel / other processes
  - …different processes might try to write to the same memory
  - …memory might not be large enough to accommodate all possible 64-bit addresses
- The operating system does a lot of work to make it seem like they do
  - Virtual memory management!

Basic idea: store mappings of virtual **pages** ⇒ physical **pages** in a data structure called a **page table**

- A **page** is a contiguous block of memory – similar to a cache block
  - Pages are generally much larger
- Page table stores data from physical memory
  - Based on a virtual address, we can figure out where in the page table we need to look for a matching physical page
  - Maintained as software, not hardware
- If we need something not in the page table, we have a **page fault** and need to fetch the data from disk

Very simplified view of a page table



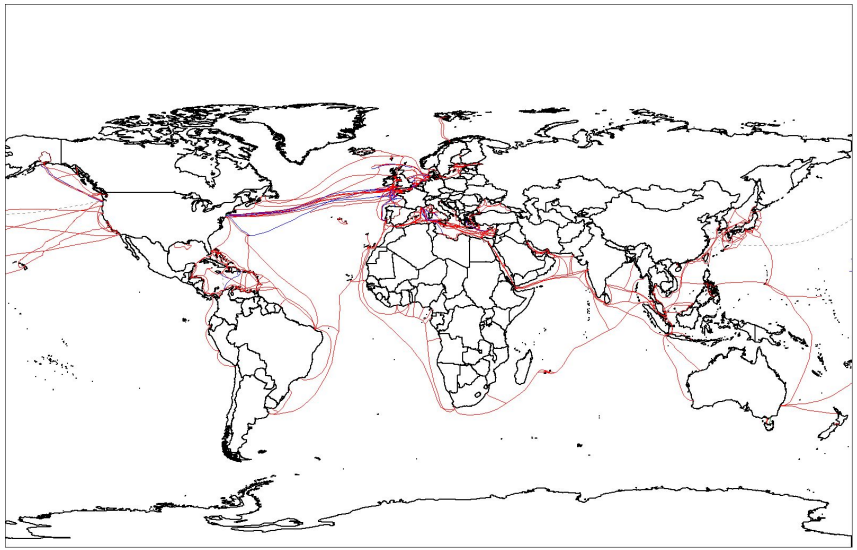
We won't go into too much detail today on virtual memory...

- ...but if you're curious, many of the mechanisms behind it are similar to caching
  - Caching was making RAM accesses more efficient, whereas page tables make disk accesses more efficient

(a little bit of) Networking and sockets

Computer networking fundamentally starts at the OS

The nuances of networking are a separate topic at our university



OS provides **ports** for computers to talk with each other

- If you are hosting a website at `localhost:3000`, you are on port 3000
  - Port 80 is for HTTP, 443 for HTTPS
- **Sockets** are a way to communicate between ports
  - `<sys/socket.h>` is part of the standard C library
- I will probably be out of time here, but feel free to investigate the `socket.h` library if you are curious

Questions?