Why Systems Software?

2025 Winter ECE353: Systems Software
Jon Eyolfson

Lecture 1
2.0.3

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

I'm Jon, Your Instructor

Eyolfson

I'm Jon, Your Instructor

Eyolfson

I'm Jon, Your Instructor

Elf

I'm Jon, Your Instructor

Elf son

I'm Jon, Your Instructor

E lfson

I'm Jon, Your Instructor

Eyolfson

Why Operating Systems?

Understanding the operating system will make you a better programmer

You will either write software that:

- Interacts with the operating system
- Is the operating system

Important URLs for Course Resources

Very Important: Sign into https://compeng.gg

- Lectures: https://eyolfson.com/courses/ece353/
- Labs: https://compeng-gg.github.io/2025-winter-ece353-docs/

These links and others are on: https://q.utoronto.ca/ (Quercus)

Labs on GitHub, Discussion on Discord, Streams on YouTube

Connect your Discord and GitHub on `https://compeng.gg`

Anonymous Discord Messages

Some students don't want to ask questions on Discord
because it's not anonymous, **we fixed that**

Use the command:
`/anon <message>`

The command sends your message anonymously in the current channel

Lecture Attendance is Still Important

It's much faster to get feedback from you and clarify if anything is unclear

We'll have live coding, I'll be able to explain any happy accidents

If there's anything else I can do to make attending a better experience let me know!

Evaluation for this Course

Assessment	Weight	Due Date
Lab 0	1%	January 13
Lab 1	4%	January 20
Lab 2	4%	February 3
Test 1	12.5%	February 12 @ 9 AM
Lab 3	4%	February 24
Lab 4	4%	March 10
Test 2	12.5%	March 12 @ 9 AM
Lab 5	4%	March 24
Lab 6	4%	April 7
Final Exam	50%	TBD

Academic Honesty Policy

You can study together, discuss concepts on Discord

Don’t post lab code on Discord, any other code is okay

Any cheating is not tolerated, and will only hurt you

The Recommended Books Complement Lectures

"Operating Systems: Three Easy Pieces"
by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau

"The C Programming Language"
by Brian Kernighan and Dennis Ritchie

Skills You Should Practice Again If Needed

C programming and debugging

Being able to convert between binary, hex, and decimal

Little-endian and big-endian

Memory being byte-addressable, memory addresses (pointers)

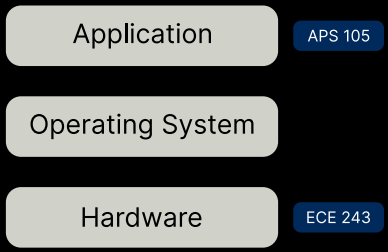
Please Provide Feedback!

This course is challenging, please let me know if anything is unclear

You can ask interesting questions, all programs interact with the OS

By the end of the course you'll be a better programmer

An Operating System Manages Resources



There's 3 Core Operating System Concepts

Virtualization: share one resource by mimicking multiple independent copies

Concurrency: handle multiple things happening at the same time

Persistence: retain data consistency even without power

“All problems in computer science can be solved by another level of indirection”

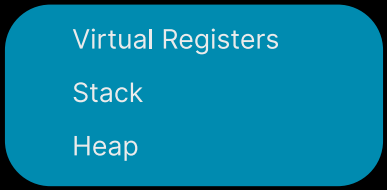
- David Wheeler

Our First Abstraction is a Process

Program: a file containing all the instructions and data required to run

Process: an instance of running a program

The Basic Requirements for a Process



My First Question to You

How are you able to run two different programs at the same time?

For example, a “hello world” program and another that counts up one every second

Does the OS Allocate Different Stacks For Each Process?

The stacks for each process need to be in physical memory

One option is the operating system just allocates any unused memory for the stack

Would there be any issues with this?

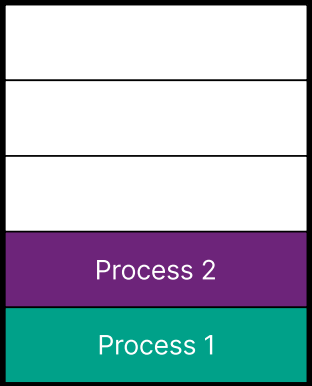
What About Global Variables?

The compiler needs to pick an address for each variable when you compile

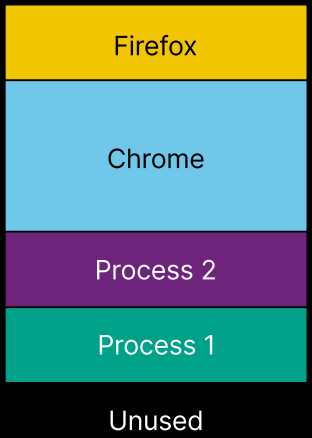
What if we had a global registry of addresses?

Would there be any issues with this?

Potential Memory Layout for Multiple Processes



Potential Memory Layout for Multiple Processes



What Happens If Two Processes Run the Same Program?

```c
#include <stdio.h>
#include <unistd.h>

static int global = 0;

int main(void) {
  int local = 0;
  while (1) {
    ++local;
    ++global;
    printf("local = %d, global = %d\n", local, global);
    sleep(1);
  }
  return 0;
}
```

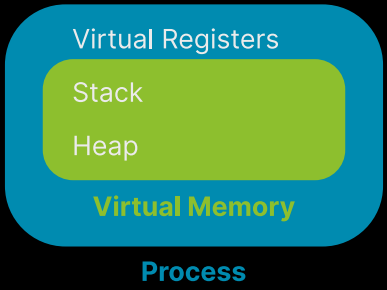
What Did We Find?

Was the address of `local` the same between the two processes?

Was the address of `global` the same between the two processes?

What else may be needed for a process?

A Process Has Its Own Virtual Memory



Example Code from This Class

All code will be in the “materials” repository located:
https://github.com/compeng-gg/2025-winter-ece353-materials

Compile the code:

```
cd lectures/01-why-systems-software
meson setup build
meson compile -C build
```

Execute the code:

```
build/read-four-bytes <FILE>
```

Source: materials/lectures/01-why-systems-software/read-four-bytes.c

Believe It or Not, This Is “Hello world”

```
0x7F 0x45 0x4C 0x46 0x02 0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x02 0x00 0xB7 0x00 0x01 0x00 0x00 0x00 0x78 0x00 0x01 0x00 0x00 0x00 0x00 0x00
0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x40 0x00 0x38 0x00 0x01 0x00 0x40 0x00 0x00 0x00 0x00 0x00
0x01 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00
0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x10 0x00 0x00 0x00 0x00 0x00 0x00 0x08 0x08 0x80 0xD2 0x20 0x00 0x80 0xD2
0x81 0x13 0x80 0xD2 0x21 0x00 0xA0 0xF2 0x82 0x01 0x80 0xD2 0x01 0x00 0x00 0xD4
0xC8 0x0B 0x80 0xD2 0x00 0x00 0x80 0xD2 0x01 0x00 0x00 0xD4 0x48 0x65 0x6C 0x6C
0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A
```

Let's Execute This Program and Verify It's "Hello world"

```
0x7F 0x45 0x4C 0x46 0x02 0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x02 0x00 0xB7 0x00 0x01 0x00 0x00 0x00 0x78 0x00 0x01 0x00 0x00 0x00 0x00 0x00
0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x40 0x00 0x38 0x00 0x01 0x00 0x40 0x00 0x00 0x00 0x00 0x00
0x01 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00
0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x10 0x00 0x00 0x00 0x00 0x00 0x00 0x08 0x08 0x80 0xD2 0x20 0x00 0x80 0xD2
0x81 0x13 0x80 0xD2 0x21 0x00 0xA0 0xF2 0x82 0x01 0x80 0xD2 0x01 0x00 0x00 0xD4
0xC8 0x0B 0x80 0xD2 0x00 0x00 0x80 0xD2 0x01 0x00 0x00 0xD4 0x48 0x65 0x6C 0x6C
0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A
```

Execute using: `./hello-world-linux-aarch64`

Aside: There's 3 Major ISAs in Use Today

ISA stands for the *instruction set architecture*
It's the machine code, or numbers the CPU understands

`x86-64` (aka `amd64`): for desktops, non-Apple laptops, servers

`aarch64` (aka `arm64`): for phones, tablets, Apple laptops

`riscv` (aka `rv64gc`): open-source implementation, similar to ARM

We'll touch on all of them in this course

Our Next Abstraction is a File Descriptor

Since our processes are independent, we need an explicit way to transfer data

IPC: inter-process communication is transferring data between two processes

File descriptor: a resource that users may either read bytes from or write bytes to
(identified by an index stored in a process)

A file descriptor could represent a file, or your terminal

System Calls Make Requests to the Operating System

We can represent system calls like regular C functions

Here are two system calls we need for a basic “Hello world” program:

```
ssize_t write(int fd, const void *buf, size_t count);
```
Description: writes bytes from a byte array to a file descriptor
- `fd` - the file descriptor
- `buf` - the address of the start of the byte array (called a buffer)
- `count` - how many bytes to write from the buffer

```
void exit_group(int status);
```
Description: exits the current process and sets an exit status code
- `status` - the exit status code (0-255)

A Hypothetical “Hello world” Program

By convention there’s some expected file descriptors:

- `0` - standard input (read)
- `1` - standard output (write)
- `2` - standard error (write)

The most basic “Hello world” program would start executing the following:

```
void _start(void) {
  write(1, "Hello world\n", 12);
  exit_group(0);
}
```

Another Aside: API Tells You What and ABI Tells You How

Application Programming Interface (API) abstracts the details and describes the arguments and return value of a function

e.g. A function takes 2 integer arguments

Application Binary Interface (ABI) specifies the details, specifically how to pass arguments and where the return value is

e.g. The same function using the C calling convention (arguments on the stack)

System Call ABI for Linux AArch64

The operating system “functions” do not have an address,
instead we can generate an interrupt for the OS

Generate an interrupt with a `svc` instruction, using registers for arguments:

- `x8` — System call number
- `x0` — 1st argument
- `x1` — 2nd argument
- `x2` — 3rd argument
- `x3` — 4th argument
- `x4` — 5th argument
- `x5` — 6th argument

What are the limitations of this?

Last Aside: C ABI, or Calling Convention for x86-64

System calls use registers, while C is stack based:

- Arguments pushed on the stack from right-to-left order
- `rax`, `rcx`, `rdx` are caller saved
- Remaining registers are callee saved
- Some arguments may be passed in registers instead of the stack

See Wikipedia for more details (there's lots of conventions, think ECE 243)

What advantages does this give us vs system call ABI? Disadvantages?

Programs on Linux Use the ELF File Format

Executable and Linkable Format (ELF) specifies both executables and libraries

Always starts with the 4 bytes: `0x7F 0x45 0x4C 0x46`
or with ASCII encoding: `DEL 'E' 'L' 'F'`

These 4 bytes are called “magic”, and that’s how you know what kind of file this is (other file formats may have a different number of bytes)

See: https://en.wikipedia.org/wiki/List_of_file_signatures
e.g., PDF files start with `%PDF-`

Our Bytes Represent an ELF File

Tells the OS to load the entire executable file into memory at address `0x10000`

The file header is 64 bytes, and the “program header” is 56 bytes
(120 bytes total)

The next 36 bytes are instructions, then 12 bytes for the string
`"Hello world\n"`
- Instructions start at `0x10078` (`0x78` is 120)
- The string (data) starts at `0x1009C` (`0x9C` is 156)

You can use: `readelf -a <FILE>` to see the gory details

Visually How Our ELF File Gets Divided

File Header

```
0x7F 0x45 0x4C 0x46 0x02 0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x02 0x00 0xB7 0x00 0x01 0x00 0x00 0x00 0x78 0x00 0x01 0x00 0x00 0x00 0x00 0x00
0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x40 0x00 0x38 0x00 0x01 0x00 0x40 0x00 0x00 0x00 0x00 0x00
```

```
0x01 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00
0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x10 0x00 0x00 0x00 0x00 0x00 0x00
```

Program Header

Instructions

```
0x08 0x08 0x80 0xD2 0x20 0x00 0x80 0xD2
0x81 0x13 0x80 0xD2 0x21 0x00 0xA0 0xF2 0x82 0x01 0x80 0xD2 0x01 0x00 0x00 0xD4
0xC8 0x0B 0x80 0xD2 0x00 0x00 0x80 0xD2 0x01 0x00 0x00 0xD4
```

Data

```
0x48 0x65 0x6C 0x6C
0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A
```

Instructions for "Hello world", Using the Linux AArch64 ABI

Plug in the 36 bytes for instructions into a disassembler, such as:
https://onlinedisassembler.com/

Our disassembled instructions:

```
mov  x8, 0x40          #64
mov  x0, 0x01          #1
mov  x1, 0x9C          #156
movk x1, 0x01, lsl 16  #0x10000
mov  x2, 0x0C          #12
svc  0x0
mov  x8, 0x5E          # 94
mov  x0, 0x0           # 0
svc  0x0
```

Data for Our “Hello world” Example

The 12 bytes of data is the `"Hello world"` string itself, ASCII encoded:

```
0x48 0x65 0x6C 0x6C 0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A
```

Low level ASCII tip: bit 5 is `0`/1 for upper case/lower case (values differ by 32)

This accounts for every single byte of our 168 byte program, let’s see what C does...

Can you already spot a difference between strings in our example compared to C?

The Kernel is a Core Part of Your Operating System

Kernel mode is a privilege level on your CPU that gives access to more instructions

Different architectures have a different name for this mode
  e.g., this is S-mode on RISC-V

The Kernel is a Core Part of Your Operating System

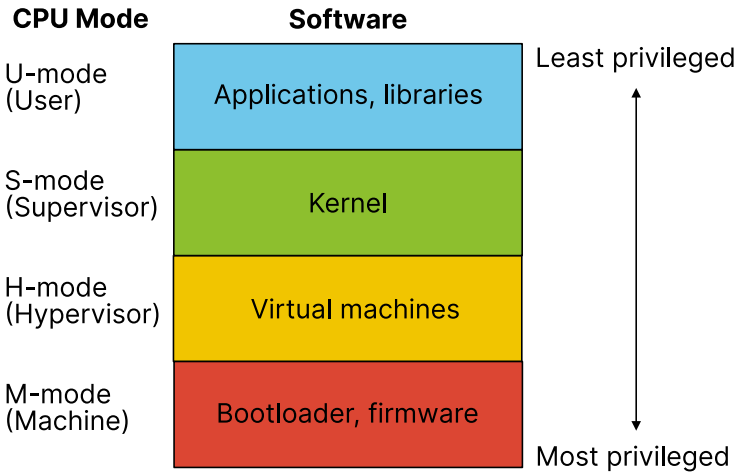
Kernel mode is a privilege level on your CPU that gives access to more instructions

Different architectures have a different name for this mode
e.g., this is S-mode on RISC-V

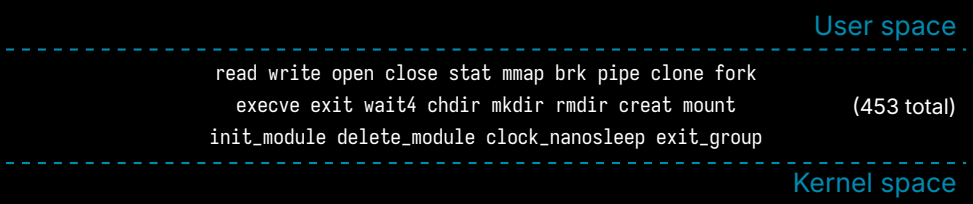
The kernel is the part of your operating system that runs in kernel mode

These instructions allow only trusted software to interact with hardware
e.g., only the kernel can manage virtual memory for processes

More Privileged CPU Modes Can Access More Instructions



System Calls Transition Between User and Kernel Mode



System Calls Are Traceable

We can trace all the system calls a process makes on Linux using the command:

```
strace <PROGRAM>
```

We can see all the system calls our “Hello world” program makes:

```
execve("./hello_world", ["./hello_world"], 0x7ffd0489de40 /* 46 vars */) = 0
write(1, "Hello world\n", 12)           = 12
exit_group(0)                           = ?
+++ exited with 0 +++
```

Now, let’s really see what C does...

System Calls for “Hello world” in C, Finding Standard Library

```
execve("./hello_world_c", ["./hello_world_c"], 0x7ffcb3444f60 /* 46 vars */) = 0
brk(NULL)                               = 0x5636ab9ea000
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=149337, ...}) = 0
mmap(NULL, 149337, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f4d43846000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000C"..., 832) = 832
lseek(3, 792, SEEK_SET)                 = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2136840, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f4d43844000
lseek(3, 792, SEEK_SET)                 = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
lseek(3, 864, SEEK_SET)                 = 864
read(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0", 32) = 32
```

System Calls for “Hello world” in C, Loading Standard Library

```
mmap(NULL, 1848896, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f4d43680000
mprotect(0x7f4d436a2000, 1671168, PROT_NONE) = 0
mmap(0x7f4d436a2000, 1355776, PROT_READ|PROT_EXEC,
     MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7f4d436a2000
mmap(0x7f4d437ed000, 311296, PROT_READ,
     MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16d000) = 0x7f4d437ed000
mmap(0x7f4d4383a000, 24576, PROT_READ|PROT_WRITE,
     MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b9000) = 0x7f4d4383a000
mmap(0x7f4d43840000, 13888, PROT_READ|PROT_WRITE,
     MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f4d43840000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f4d43845500) = 0
mprotect(0x7f4d4383a000, 16384, PROT_READ) = 0
mprotect(0x5636a9abd000, 4096, PROT_READ) = 0
mprotect(0x7f4d43894000, 4096, PROT_READ) = 0
munmap(0x7f4d43846000, 149337)          = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}) = 0
```

System Calls for “Hello world” in C, Setting Up Heap and Printing

```
brk(NULL)                               = 0x5636ab9ea000
brk(0x5636aba0b000)                     = 0x5636aba0b000
write(1, "Hello world\n", 12)           = 12
exit_group(0)                           = ?
+++ exited with 0 +++
```

The C version of “Hello world” ends with the exact same system calls we made

You Can Think of the Kernel as a Long Running Process

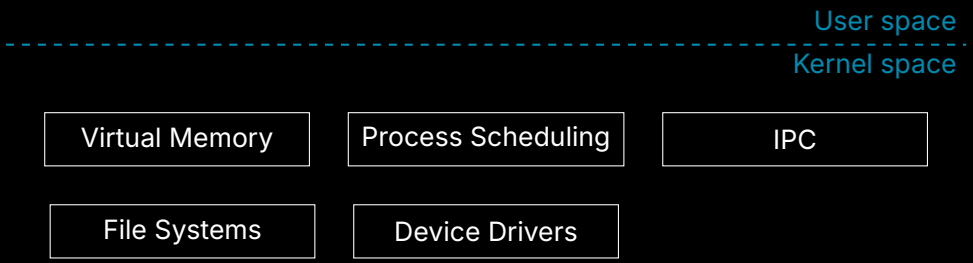
Writing kernel code is more like writing library code (there's no `main`)

The kernel lets you load code (called modules)

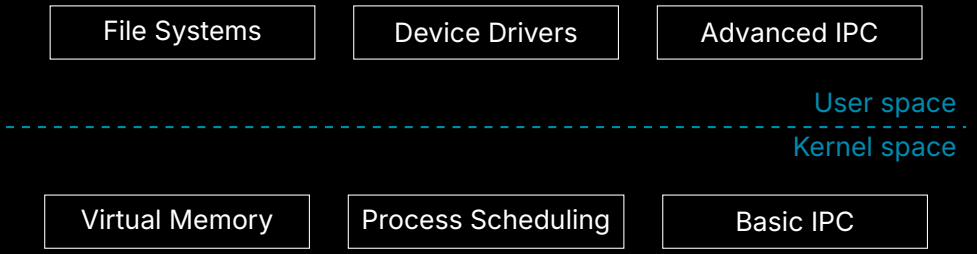
Your code executes on-demand
e.g. when it's loaded manually, new hardware, or accessing a certain file

If you write a kernel module, you can execute privileged instructions
and access any kernel data, so you could do anything

A Monolithic Kernel Runs Operating System Services in Kernel Mode



A Microkernel Runs the Minimum Amount of Services in Kernel Mode



Other Types of Kernels

"Hybrid" kernels are between monolithic and microkernels
- Emulation services to user mode (Windows)
- Device drivers to user mode (macOS)

Nanokernels and picokernels
- Move even more into user mode than traditional microkernels

There's different architectural lines you can draw with different trade-offs

Kernel Interfaces Operate Between CPU Mode Boundaries

The lessons from the lecture:

- The kernel is the part of the OS that interacts with hardware (it runs in kernel mode)
- System calls are the interface between user and kernel mode
  - Every program must use this interface!
- File format and instructions to define a simple “Hello world” (in 168 bytes)
  - Difference between API and ABI
  - How to explore system calls
- Different kernel architectures shift how much code runs in kernel mode