

ECE 344: Operating Systems  
Lecture 2

# Introduction

1.0.2

Jon Eyolfson  
September 12, 2022



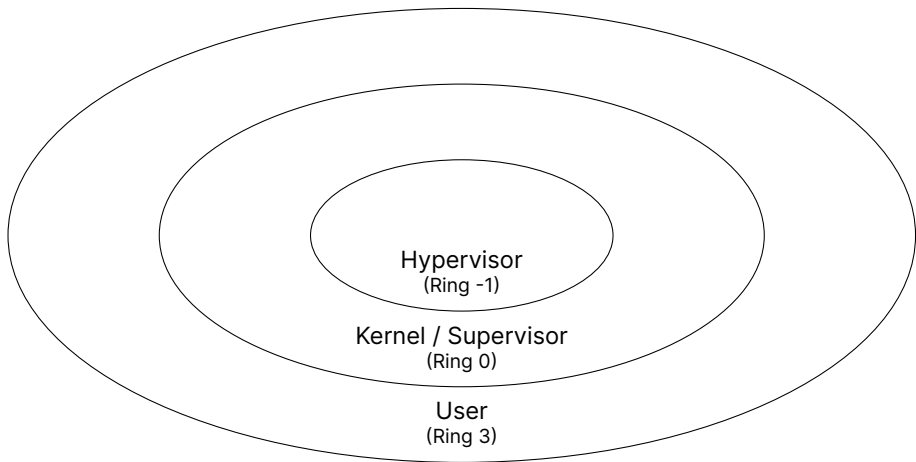
This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/)

I'll Post Examples on GitHub

# GitHub

<https://github.com/eyolfson/ece344-22fall-examples/>

## CPUs Have “Rings” to Control Instruction Access

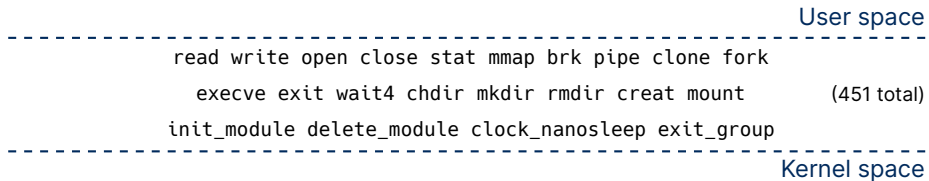


Each ring can access instructions in any of its outer rings

## The Kernel of the Operating System Runs in Kernel Mode



## System Calls Transition between User and Kernel Mode



## System Calls Behave Like Regular Functions, But Run in Kernel Mode

```
ssize_t write(int fd, const void *buf, size_t count);
```

The write system call's API is:

- A file descriptor to write bytes to
- An address to contiguous sequence of bytes
- How many bytes to write from the sequence

```
void exit_group(int status);
```

The exit\_group system call's API is:

- An exit code for the program (0-255)

## Let's Execute a 168 Byte "Hello World" on Linux AArch64

```
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
```

# ELF is the Binary Format for Unix Operating Systems

Executable and Linkable Format (ELF) is a file format

Always starts with the 4 bytes: `0x7F 0x45 0x4C 0x46`

or with ASCII encoding: `0x7F 'E' 'L' 'F'`

Followed by a byte signifying 32 or 64 bit architectures  
then a byte signifying little or big endian

Most file formats have different starting signatures (or magic numbers)



## Use `readelf` to Read ELF File Headers

Command: `readelf -a <filename>` (for all information)

Contains the following:

- A header containing:
  - Information about the machine (e.g. the ISA)
  - The entry point of the program
- Any **program headers** (required for executables)
- Any **section headers** (required for libraries)

## Our Minimal Executable Contains a Single Program Header

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

The 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 starts at `0x1009C` (`0x9C` is 156)

## “Hello world” Needs 2 System Calls

Command: `strace <filename>`

This shows all the system calls our program makes:

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

## Quick Aside: API Tells You What and ABI Tells You How

Application Programming Interface (API) abstracts the details how how to communicate

e.g. A function takes 2 integer arguments

Application Binary Interface (ABI) specifies how to layout data and how to concretely communicate

e.g. The same function using the C calling convention

## System Call ABI for Linux AArch64

Enter the kernel with a svc instruction, using registers for arguments:

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

What are the limitations of this?

Note: other registers are not used, whether they're saved isn't important for us

## Instructions for “Hello world”, Using the Linux AArch64 ABI

Plug in the next 36 bytes into a disassembler, such as:

<https://onlinedisassembler.com/>

Our disassembled instructions:

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

## Finishing Up “Hello world” Example

The remaining 12 bytes 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?

## Source Code for “Hello world” in C

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello world\n");
    return 0;
}
```

Compile with `make` in `lecture-02` in the examples repository

What are other notable differences between this and our “Hello world”?



## 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", 0_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", 0_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)

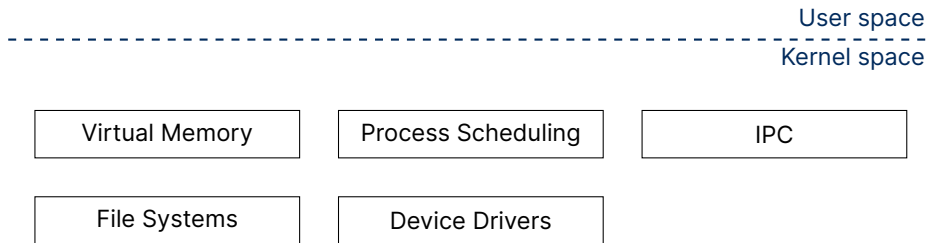
The kernel lets you load code (called modules)

Your code executes on-demand

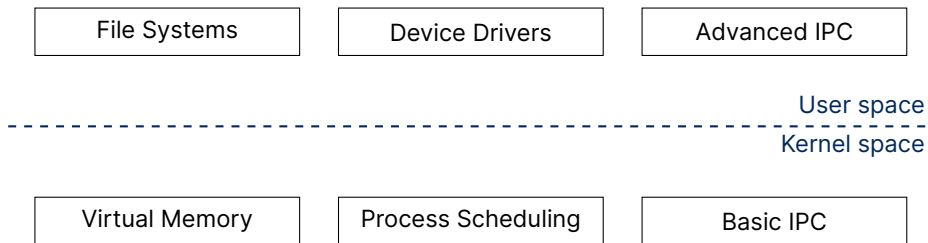
e.g. when it's loaded manually, new hardware, or accessing a certain file

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 many different lines you can draw with different trade-offs

## Kernel Interfaces Operate Between CPU Mode Boundaries

The lessons from the lecture:

- Code running in kernel mode is part of your kernel
- 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