CS 111: Operating System Principles

Lecture 2

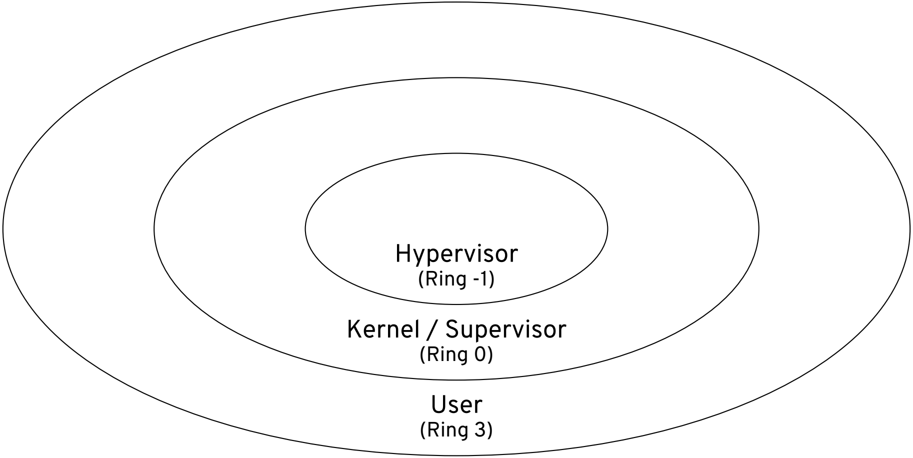
Interfaces

1.0.1

Jon Eyolfson

March 30, 2021

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

User space

Kernel space

System Calls Transition between User and Kernel Mode

User space

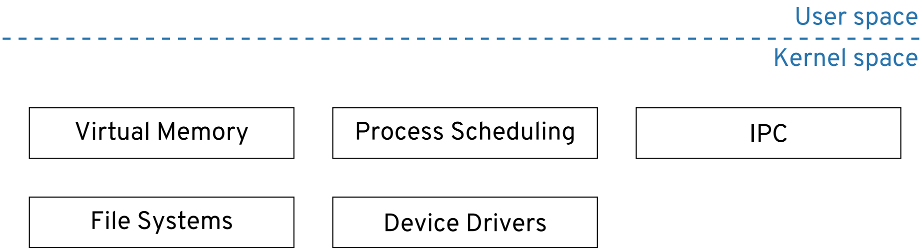
`read write open close stat mmap brk pipe clone fork`

`execve exit wait4 chdir mkdir rmdir creat mount` (352 total)

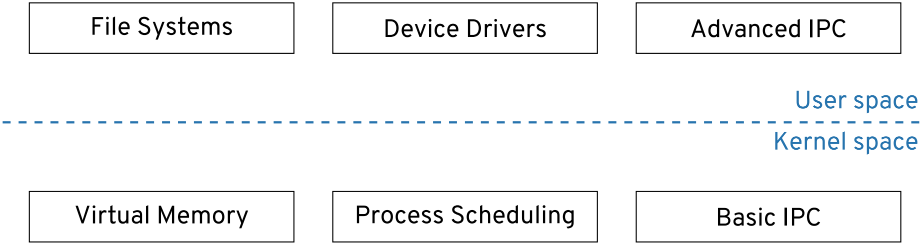
`init_module delete_module clock_nanosleep exit_group`

Kernel space

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

Let’s Execute a 178 Byte “Hello World” on Linux x86-64

```
0x7F 0x45 0x4C 0x46 0x02 0x01 0x01 0x03 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x02 0x00 0x3E 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
0xB2 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xB2 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x48 0xC7 0xC0 0x01 0x00 0x00 0x00 0x48
0xC7 0xC7 0x01 0x00 0x00 0x00 0x48 0xC7 0xC6 0xA6 0x00 0x01 0x00 0x48 0xC7 0xC2
0x0C 0x00 0x00 0x00 0x0F 0x05 0x48 0xC7 0xC0 0xE7 0x00 0x00 0x00 0x48 0xC7 0xC7
0x00 0x00 0x00 0x00 0x0F 0x05 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 <filename>`

Contains the following:

- 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)

The header is 64 bytes, so we still have to account for 114 more.

Result of `readelf -h` on "Hello world"

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x10078
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         1
  Size of section headers:           64 (bytes)
  Number of section headers:         0
  Section header string table index: 0
```

ELF Program Header

Tells the operating system how to load the executable:

- Which type? Examples:
  - Load directly into memory
  - Use dynamic linking (libraries)
  - Interpret the program
- Permissions? Read / Write / Execute
- Which virtual address to put it?
  - Note that you'll rarely ever use physical addresses (for embedded)

For "Hello world" we load everything into memory

One program header is 56 bytes

58 bytes left

Result of `readelf -l` on “Hello world”

```
Elf file type is EXEC (Executable file)
Entry point 0x10078
There is 1 program header, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000010000 0x0000000000010000
                 0x00000000000000b2 0x00000000000000b2  R E    0x100
```

“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 API for “Hello world”

`strace` shows the API of system calls

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

The `exit_group` system call’s API is:

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

System Call ABI for Linux x86-64

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

- `rax` – System call number
- `rdi` – 1st argument
- `rsi` – 2nd argument
- `rdx` – 3rd argument
- `r10` – 4th argument
- `r8` – 5th argument
- `r9` – 6th 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 x86-64 ABI

Plug in the next 46 bytes into a disassembler, such as:
`https://onlinedisassembler.com/`

Our disassembled instructions:

```
mov rax,0x1
mov rdi,0x1
mov rsi,0x100a6
mov rdx,0xc
syscall
mov rax,0xe7
mov rdi,0x0
syscall
```

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 178 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 `examples/lecture-02`

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", 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

Kernel Interfaces Operate Between CPU Mode Boundaries

The lessons from the lecture:

- Code running in kernel mode is part of your kernel
- Different kernel architectures shift how much code runs in kernel mode
- System calls is the interface between user and kernel mode
- Everything involved to define a simple “Hello world” (in 178 bytes)
    - Difference between API and ABI
    - How to explore system calls