UCLA CS35L

Week 6

Monday

Reminders

- Assignment 5 due this Friday (5/8)
- Assignment 6 due next Friday (5/15)
  - This will use the work you do for Assignment 5
- This Friday (5/8) will be deadline to sign up for Week 10 Partners/Time Slots
  - Afterwards I will randomly assign teammates/time slots
  - I will also accept recordings for those who have time zone issues (just email me in advance)
- Still working on regrade functionality for Assignment 3, will send announcement when it is working
- Anonymous feedback for Daniel
  - https://forms.gle/tZwuMbALe825DBVn8

Kernel

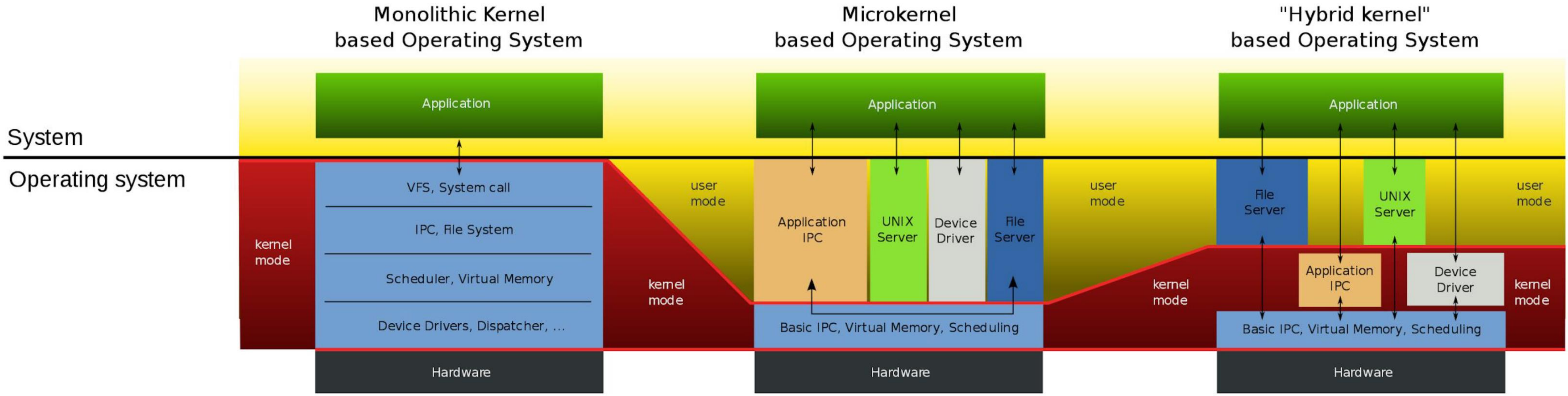
What is a Kernel

- The Kernel is a program that is heart of the Operating System
- Responsible for:
  - **Memory Management** – Keeps track of memory locations, and how much memory is being used to store what
  - **Process Management and Scheduling** – Determine what processes can use the CPU, when, and for how long
  - **Device Drivers** – Acts as the interpreter between hardware and software
  - **Systems Calls and Security** – Protects sensitive information and access while still allowing applications to perform as necessary
- Kernel is the first program loaded at boot-up
- If the Kernel crashes, the whole OS crashes

History of the Kernel

- The idea of a Kernel responsible for scheduling became popular in the 60s and 70s so that Operating Systems could time-share multiple processes or users
- Unix is a combination of the Unix utilities and file system + a Kernel. Linux is specifically the combination of GNU Operating System + Linux Kernel (developed by Linus Torvalds)
- Linux is a monolithic kernel, in contrast with microkernel (Mac OS X)
  - Interesting online “discussion” between Linus Torvalds and Andrew Tanenbaum (OS researcher and developer of his own microkernel) on Kernel design here.

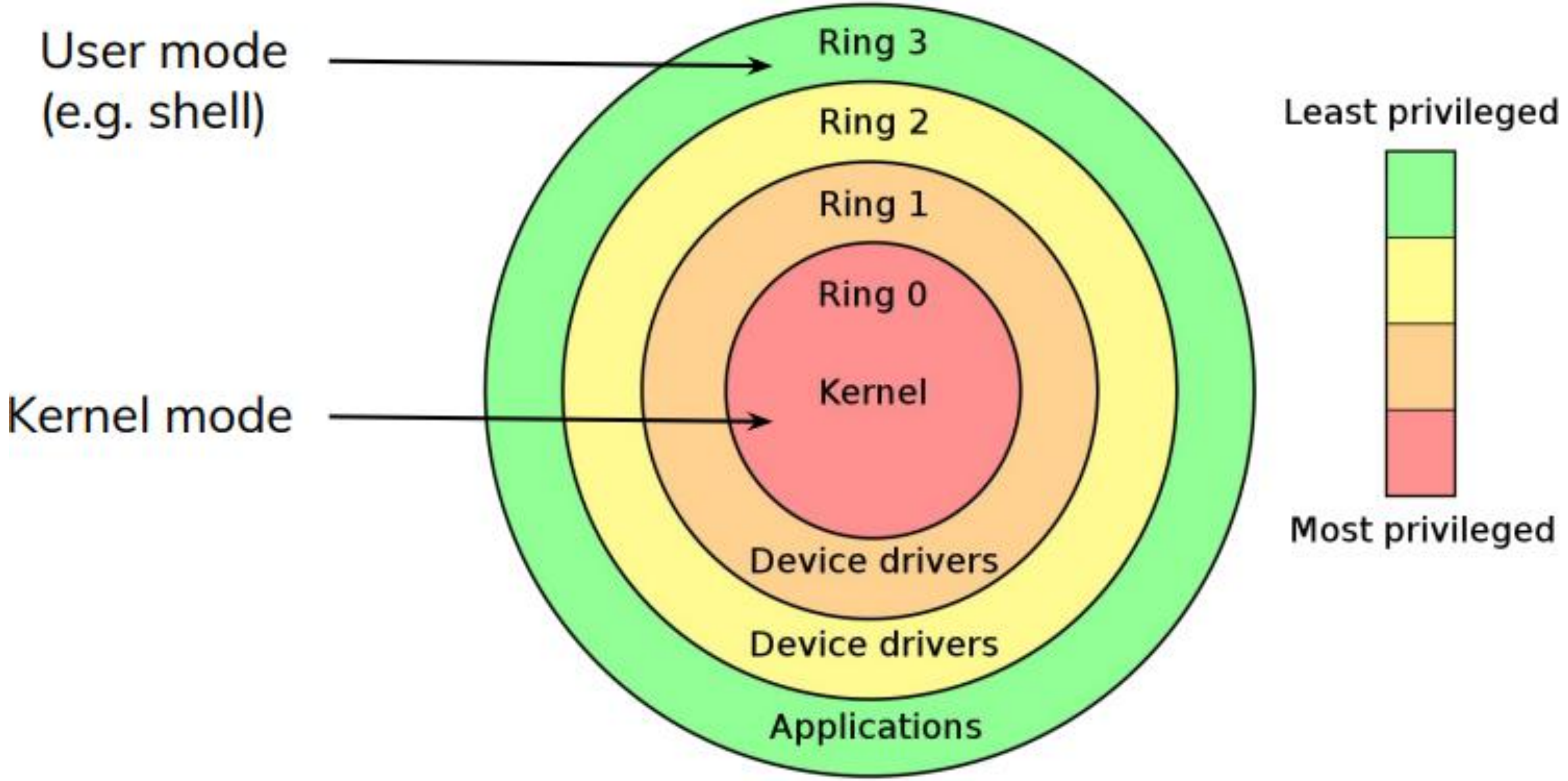
High-Level Kernel Design



The Kernel and Security

- The kernel has control over pretty much anything that your computer has access too, so how do we keep that secure and reliable?
- We can’t give user’s full control over the kernel, but the kernel still needs access. So we have two modes, using a mode bit in the CPU
  - Kernel mode – Full access to hardware and privileged CPU instructions. Can access all memory space
  - User mode – This is where most code executes, must use System APIs to request work to be done. Is limited to specified memory space for application.
- There is also separate user space and kernel space memory

Kernel Model



System Calls

What is a System Call

- When user programs need the kernel to perform a privileged operation, they request that via a systems call.
- Systems Call Process:
  - User process makes system call
  - If allowed, a trap is generated to interrupt the process and switch to kernel mode
  - Kernel executes the requested function
  - Switch back to user process and user mode to continue execution
- Common actions that require a system call:
  - I/O Operations
  - Memory Operations
- This switch is expensive!

General System Calls Overview

- 5 Types of System Calls
  - Process Control – need to alter process execution of a running program
  - **File management** – create/delete/read/write to files
  - Device Management – manipulate peripherals
  - Information Management – retrieve information from OS like time/date
  - Communication – Create interprocess communication channels

System Call vs Library Call

- We typically use library calls, like putchar or printf. How do they perform privileged operations like printing to the screen?
  - They make System Calls!
- Note these library calls are typically more efficient than if we wrote our own versions, since they minimize the number of system calls made
  - Remember, privilege switch is expensive
- Library calls also abstract the process of making system calls, making them portable across systems

Example System Calls

```
#include <unistd.h>
//returns the process id of the calling process
pid_t getpid(void)

//duplicates a file descriptor fd
int dup(int fd)
//NOTE the default file descriptors
0 – stdin
1 – stdout
2 - stderr
```

System Calls – read/write

```
#include <unistd.h>
//read n number of bytes from files described by fd to buffer
ssize_t read(int fd, void* buffer, size_t n)

//write n number of bytes from files described by fd to buffer
ssize_t write(int fd, void* buffer, size_t n)
```

System Calls – Open/Close

```
//given a pathname to a file, returns a file descriptor (int)
int open(const char *pathname, int flags)

//closes given file descriptor
int close(int fd)

//NOTE stdin, stdout, and stderr have predefined file descriptor
//STDIN_FILENO = 0
//STDOUT_FILENO = 1
//STDERR_FILENO = 2
```

System Calls - fstat

```
//given a fle descriptor, and address to a 'struct stat'. Will
//populate the struct with file details
int fstat(int fd, struct stat *buf)
```

```
//struct stat details:
struct stat {
               dev_t     st_dev;         /* ID of device containing file */
               ino_t     st_ino;         /* Inode number */
               mode_t    st_mode;        /* File type and mode */
               nlink_t   st_nlink;       /* Number of hard links */
               uid_t     st_uid;         /* User ID of owner */
               gid_t     st_gid;         /* Group ID of owner */
               dev_t     st_rdev;        /* Device ID (if special file) */
               off_t     st_size;        /* Total size, in bytes */
               blksize_t st_blksize;     /* Block size for filesystem I/O */
               blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
```

Example syscall – read/write

```c
int main()
{
    int fd = open("input.txt", O_RDONLY, 0);
    if (fd < 0)
    {
        perror("file open failed");
        exit(1);
    }

    printf("\nfd is %d\n", fd);
    char *readBuf1 = malloc(15 * sizeof(char));
    memset(readBuf1, '\0', 15 * sizeof(char));

    int charsRead1 = read(fd, readBuf1, 15 * sizeof(char));

    //STDOUT_FILENO = 1;
    int charsWrite = write(STDOUT_FILENO, readBuf1, strlen(readBuf1));
}
```

Example syscall - fstat

```c
int main()
{
    struct stat fileData;
    //STDIN_FILENO = 0
    if (fstat(STDIN_FILENO, &fileData) < 0)
    {
        fprintf(stderr, "fstat error");
        exit(1);
    }
    printf("Size of File: %ld \n", fileData.st_size);
    return 0;
}
```

General System Visibility - strace

- Shows all system calls made by a program

```
strace [flags] ./someExecutable
```

General System Visibility - time

- Gives actual elapsed time, user CPU time, and system CPU time

```
time [flags] ./someExecutable
```

Other ways to make system calls

- Use syscall() in <sys/syscall.h> with the specific system call number
- Write the assembly instructions directly
- But seriously… Use unistd.h to make your life better
  - This is a library wrapper to the system API

Buffered vs Unbuffered I/O

Why have buffered and unbuffered I/O?

- Making a system call is expensive, so if we had to make a system call for every single character I/O – that's not ideal.
- We can collect as many bytes as possible in a buffer (either for read or write) and then make one system call.
- But sometimes we need immediate results! So have both options available to you

Default buffered/unbuffered behavior

- **Stdout** – Is usually *line buffered* by default. We assume there is a large amount of data going through, and it can wait momentarily until a buffer is collected.
  - Line buffered means we output the buffer at a \n character
  - `fflush(File *stream)` – forces a write of all buffered data

- **Stderr** – Is *unbuffered* by default. We assume errors are infrequent, but we want to know about them immediately.