CS 241: Systems Programming
Lecture 26. System Calls I

Spring 2020

Prof. Stephen Checkoway

What is an operating system?

Operating system tasks

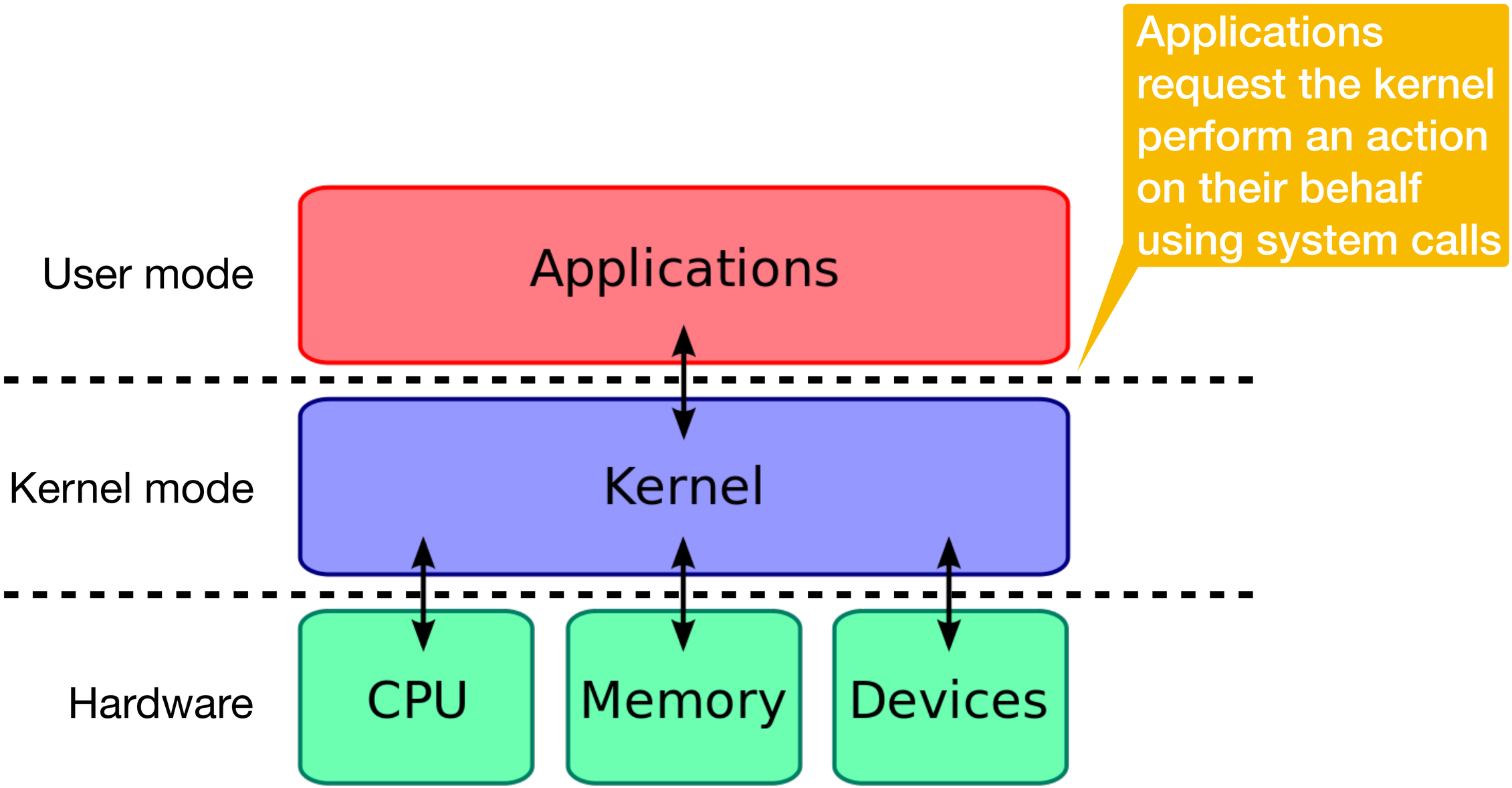
Managing the resources of a computer

- CPU, memory, network, etc.

Coordinate the running of all other programs

OS can be considered as a set of programs

- kernel – name given to the core OS program



https://en.wikipedia.org

Do we need an operating system?

A. Yes

B. No

C. I don't know/I'm not sure

System calls

Programs talk to the OS via system calls

- Set of functions to request access to resources of the machine
- System calls vary by operating system and computer architecture

Types of system calls

- Input/output (may be terminal, network, or file I/O)
- File system manipulation (e.g., creating/deleting files/directories)
- Process control (e.g., process creation/termination)
- Resource allocation (e.g., memory)
- Device management (e.g., talking to USB devices)
- Inter-process communication (e.g., pipes and sockets)
- …

Most basic UNIX system call: exit

Programs (normally) end by returning from `main()` or calling `exit()`

After running the `atexit` handlers, the program asks the kernel to stop running the program using the `exit` system call

The `exit` system call takes an exit status as its only parameter

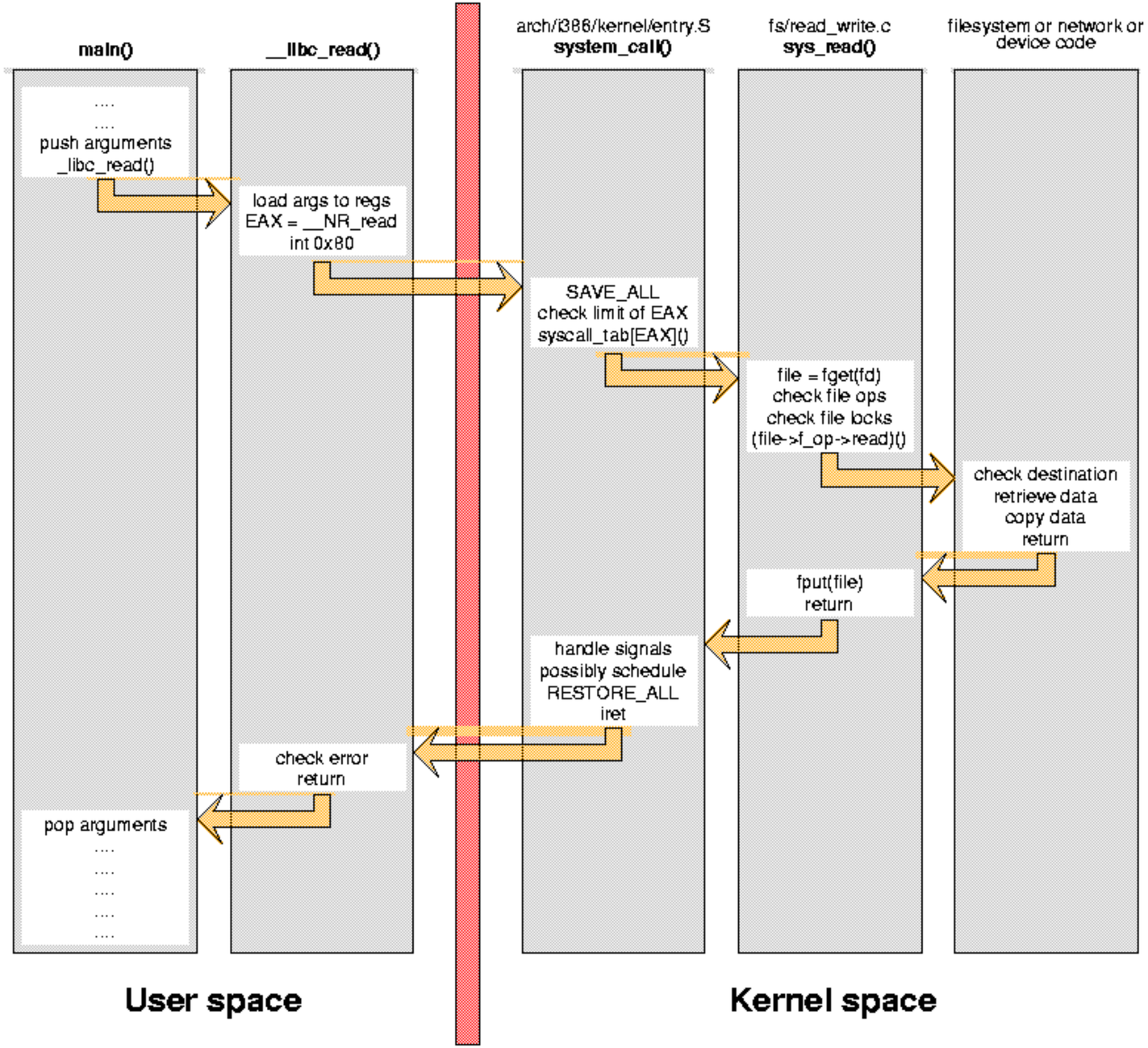
When the kernel receives an `exit` system call from a program, it

- cleans up all of the resources associated with the program
- notifies the program that created the exiting program (the parent) that a child has exited

System calls as API

System calls are an example of an **application programming interface** (API)

- Each system call is assigned a small integer (the system call number)
- System calls are performed by setting up the arguments (often in registers) and using a dedicated "system call" or "interrupt" instruction
- The kernel's system call handler calls an appropriate function based on the system call number
- Data (and success/failure) is returned to the application



http://www.linux.it/~rubini/docs/ksys/

System calls and libc

C standard library

- Some functions make no system calls (e.g., `strcpy(3)`)
- Some functions "wrap" a single system call (e.g., `open(2)`)
- Some functions have complex behavior and might make a variable number of system calls (e.g., `malloc(3)`)

We're going to focus on the libc wrappers for the system calls

- These live in section 2 of the manual: `open(2)`, `_exit(2)`, `fork(2)`

Why do we use system calls instead of making a function call directly to the function in the kernel that will handle our system call request?

Discuss with your group ~~and select A on your clickers when you have a reason (or multiple reasons)~~

Input/output system calls

Open a file: open(2)

```
#include <fcntl.h>

int open(char const *path, int oflag, ...);
```

- `O_RDONLY` open for reading only
- `O_WRONLY` open for writing only
- `O_RDWR` open for reading and writing
- `O_APPEND` append on each write
- `O_TRUNC` truncate size to 0
- `O_CREAT` create file if it does not exist
- `O_EXCL` error if O_CREAT and the file exists
- `O_NONBLOCK` do not block on open or for data to become available

Last arg is the "int mode" -- see chmod(2) and umask(2)

Returns file descriptor on success, -1 on error

File descriptors

Integer index into OS file table for this process

3 are automatically created for you

- **`STDIN_FILENO`** 0 standard input
- **`STDOUT_FILENO`** 1 standard output
- **`STDERR_FILENO`** 2 standard error

These are what are used in shell redirection

- `$ ./a.out 2> errors.txt`

Read data: read(2)

```
#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);
```

- Attempts to read `nbytes` from `filedes` storing data in `buf`
- Returns the number of bytes read
- Upon **`EOF`**, returns `0`
- Upon error, returns `-1` and sets **`errno`**

Write data: write(2)

```
#include <unistd.h>

ssize_t write(int fildes, void const *buf, size_t nbyte);
```

- Attempts to write nbyte of data to the object referred to by filedes from the buffer buf
- Upon success, returns number of bytes are written
- On error, returns -1 and sets errno

read(2)/write(2) vs. fread(3)/fwrite(3)

Each call to read/write makes the corresponding system call

fread/fwrite maintain an internal array for buffering (recall line/block buffering)

- Results in fewer system calls in the usual case
- Often improves performance with many small reads and writes

Seek in file: lseek(2)

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

- Like `fseeko(3)` but for file descriptors, not streams
- `whence` is one of **`SEEK_SET`**, **`SEEK_CUR`**, **`SEEK_END`**
- On success, returns the resultant offset in terms of bytes from the beginning of the file
- On error, returns `(`**`off_t`**`)-1` and sets **`errno`**

Close files: close(2)

```
#include <unistd.h>

int close(int fildes);
```

- Closes `fildes`, returns 0 on success
- Returns -1 and sets errno on error

File descriptor <-> stream

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

```
FILE *fdopen(int fildes, const char *mode);
```

- Opens a file descriptor as a stream
- When you `fclose()`, descriptor is closed

```
int fileno(FILE *stream);
```

- Returns file descriptor associated with a `stream`

It's best not to mix stdio functions with low-level system calls: use one or the other

Which statement is true if we run the following code

```
FILE *fp = fopen(path, "r");  // Open a file
fgets(buf, size, fp);         // Read a line
int fd = fileno(fp);          // Get the underlying file descriptor
lseek(fd, 0, SEEK_SET);       // Rewind to the beginning of the file
fgets(buf2, size, fp);        // Read a line
```

A. `buf` and `buf2` have the same contents

B. `buf` and `buf2` have different contents (unless the first two lines are identical)

C. There's no way to know if they will be the same or different

D. It's an error to mix `lseek(2)` and fgets(3)

File system manipulation system calls

Delete files: unlink(2)

```c
#include <unistd.h>

int unlink(char const *path);
```

- Removes `path`, returns `0` on success
- Returns `-1` and sets **`errno`** on error

Rename files: rename(2)

```
#include <stdio.h>

int rename(char const *oldpath, char const *newpath);
```

- Renames `oldpath` to `newpath`, returns `0` on success
- Returns `-1` and sets **`errno`** on error
- This can change directories, but not file systems!

Get current directory: getcwd(3)

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

- Copies absolute path of current working directory to `buf`
  - length of array is "`size`"
  - if path is too long (including null byte), `NULL/ERANGE`
- Linux allows `NULL` for `buf` for dynamic allocation, see man page

Change directories: chdir(2)

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fildes);
```

Change working directory of calling process

- How "`cd`" is implemented
- `fchdir()` is only in certain standards, but widely available
- `fchdir()` lets you return to a directory referenced by a file descriptor from `open(2)`ing a directory

`0` on success, `-1`/**`errno`** on error

Create/delete a directory

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(char const *path, mode_t mode);
```

- Create a directory called path
- Don't forget execute bits in mode!

```
#include <unistd.h>

int rmdir(char const *path);
```

- Delete the directory specified by path

`0` for success, `-1`/**`errno`** on error

Reading directories

`opendir(3)`, `readdir(3)`, `closedir(3)`

- Enables the application to read the contents of directories

In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-26.html

Grab a laptop and a partner and try to get as much of that done as you can!