

CS 241: Systems Programming

Lecture 18. System Calls I

Fall 2023

Prof. Stephen Checkoway

What is an operating system?

Operating system tasks

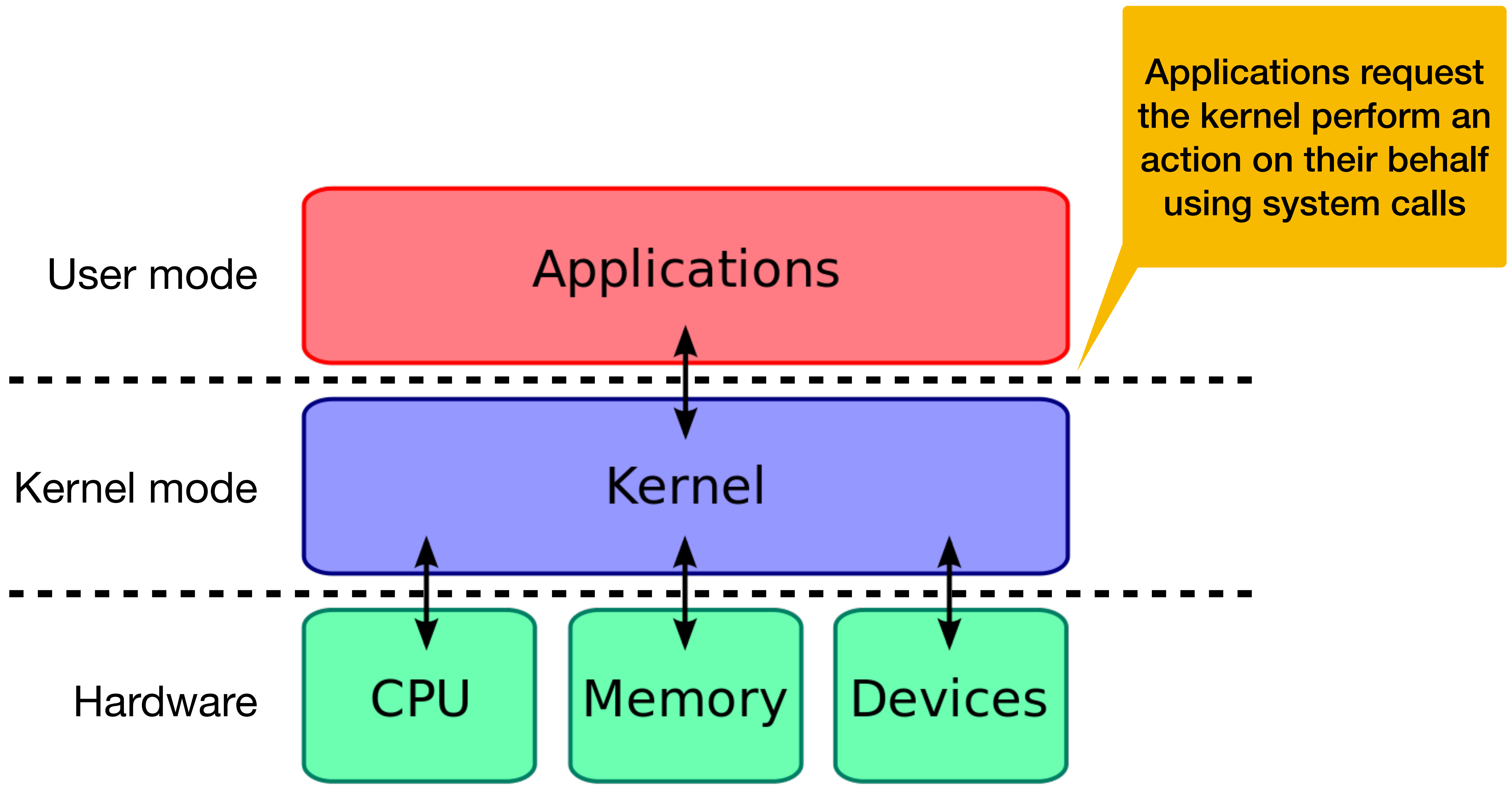
Managing the resources of a computer

- 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



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 calling `exit ()` or returning from `main ()` ...
which calls `exit ()`

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

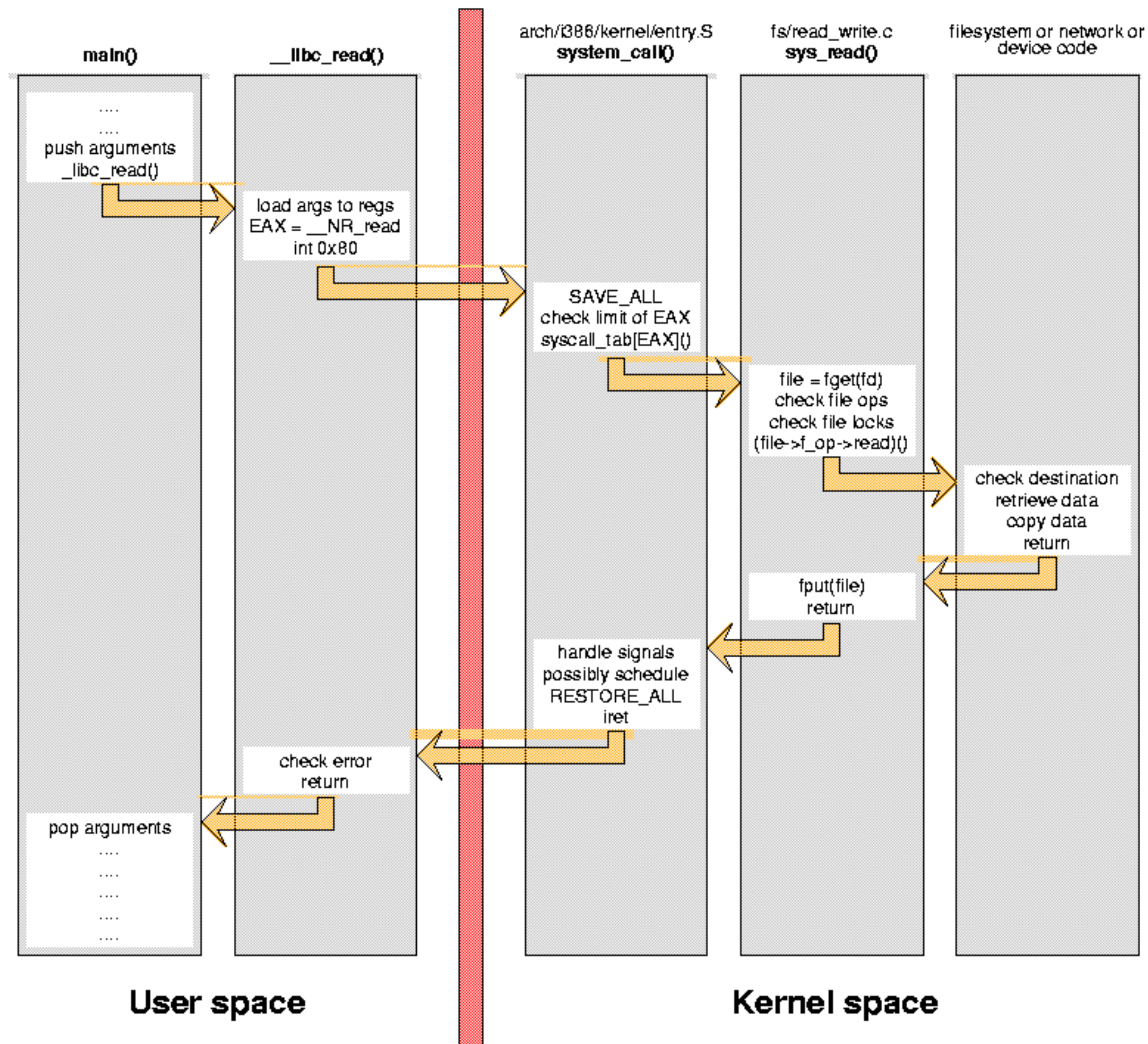
When the kernel receives an `exit` system call from a program, it

- 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



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)`

System calls and Rust

OS vendors make changes to their system calls over time

Different computer architectures use different system call numbers

To deal with this, the system call interface lives in libc:

- To make a system call, applications call functions in libc
- libc places the system call number and arguments in the correct registers and traps into the kernel

In Rust, we have two options

1. Use higher-level functionality provided by the standard library
2. Call functions in libc

libc crate

The libc crate exposes libc functions/types/constants in Rust

System call wrapper in libc	Rust declaration of the function	Notes
<code>void _exit(int status)</code>	<code>fn _exit(status: c_int) -> !</code>	Exit (doesn't return)
<code>pid_t getpid(void)</code>	<code>fn getpid() -> pid_t</code>	Get the process ID
<code>ssize_t read(int fildes, void *buf, size_t nbyte);</code>	<code>fn read(fd: c_int, buf: *mut c_void, count: size_t) -> ssize_t</code>	Read data from a file
<code>int rename(const char *old, const char *new)</code>	<code>fn rename(oldname: *const c_char, newname: *const c_char) -> c_int</code>	Renames files

Types of arguments

Arguments to syscalls fall into a few basic types

C type	Libc crate's equivalent	Normal Rust equivalent (on many platforms)	Notes
int	c_int	i32	Normal integer
size_t/ssize_t	size_t/ssize_t	usize/usize	Represents the size of things ssize_t is used to return -1 as an error
char *	*const c_char *mut c_char	&CStr CString *const i8/*mut i8	0-byte terminated string
void *	*const c_void *mut c_void		A pointer to <i>anything</i>
Pointers to structs	Pointers to structs with the same name		References can be converted to pointers

Errors

When a system call fails

- the C wrapper returns -1 (or NULL, in some cases)
- the per-thread global variable `errno` is set to an integer specifying the reason

Rust's `std::io::last_os_error()` reads `errno` and constructs a `std::io::Error` which we can use with a `Result`

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)

... indicates 0 or more additional arguments. In this case, open() takes exactly 2 or 3 arguments

```
#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

Bitwise OR the flags together,
e.g.,
O_WRONLY | O_CREAT

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 fildes 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 fildes from the buffer buf
- Upon success, returns number of bytes are written
- On error, returns -1 and sets errno

Seek in file: lseek(2)

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

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

- 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

Reading a file with system calls


1. Open the file with `libc::open()` and handle errors
2. Reserve space in a `Vec<u8>`
3. Read some data with `libc::read()` and handle errors
4. If all of the data was not read, go back to step 2
5. Close the file with `libc::close()`

Opening the file

```
use std::ffi::CString;
use std::io;

fn read_file(path: &str) -> io::Result<Vec<u8>> {
    let path = CString::new(path)?;
    let mut data: Vec<u8> = Vec::new();

    unsafe {
        let fd = libc::open(path.as_ptr(), libc::O_RDONLY);
        if fd == -1 {
            return Err(io::Error::last_os_error());
        }
        // Read the data here
        libc::close(fd);
    }
    Ok(data)
}
```



Construct a 0-terminated
C string

Reserve space

```
fn read_file(path: &str) -> io::Result<Vec<u8>> {  
    // ...  
    loop {  
        if data.capacity() - data.len() < 4096 {  
            data.reserve(4096);  
        }  
        // ...  
    }  
    // ...  
    Ok(data)  
}
```

Read some data

```
fn read_file(path: &str) -> io::Result<Vec<u8>> {  
    // ...  
    loop {  
        // ...  
        let ptr: *mut libc::c_void = data.as_mut_ptr()  
            .offset(data.len() as isize)  
            .cast();  
        let amount = libc::read(fd, ptr, data.capacity() - data.len());  
        if amount < 0 {  
            let err = io::Error::last_os_error();  
            libc::close(fd);  
            return Err(err);  
        }  
        if amount == 0 {  
            break;  
        }  
        data.set_len(data.len() + amount as usize);  
    }  
    // ...  
    Ok(data)  
}
```

Easy to forget to close
the file!

```

fn read_file(path: &str) -> io::Result<Vec<u8>> {
    let path = CString::new(path)?;
    let mut data: Vec<u8> = Vec::new();

    unsafe {
        let fd = libc::open(path.as_ptr(), libc::O_RDONLY);
        if fd == -1 {
            return Err(io::Error::last_os_error());
        }
        loop {
            if data.capacity() - data.len() < 4096 {
                data.reserve(4096);
            }
            let ptr: *mut libc::c_void = data.as_mut_ptr().offset(data.len() as isize).cast();
            let amount = libc::read(fd, ptr, data.capacity() - data.len());
            if amount < 0 {
                let err = io::Error::last_os_error();
                libc::close(fd);
                return Err(err);
            }
            if amount == 0 {
                break;
            }
            data.set_len(data.len() + amount as usize);
        }
        libc::close(fd);
    }
    Ok(data)
}

```

Contrast with normal Rust

```
fn read_file(path: &str) -> io::Result<Vec<u8>> {  
    use std::io::Read;  
    let mut file = File::open(path)?;  
    let mut data = Vec::new();  
    file.read_to_end(&mut data)?;  
    Ok(data)  
}
```

open system call

1 or more read system
calls

close system call when
file is dropped

Or even easier

```
fn main() {  
    let data1 = read_file("example.txt").unwrap();  
    let data2 = std::fs::read("example.txt").unwrap();  
    assert_eq!(data1, data2);  
}
```



One function to call.
It'll call `open()`, `read()`, and `close()`

```

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void *read_file(char const *path,
                size_t *len_ptr)
{
    int fd = open(path, O_RDONLY);
    if (fd == -1) {
        return NULL;
    }
    *len_ptr = 0;
    char *data = NULL;
    size_t len = 0;
    size_t cap = 0;
    while (1) {
        if (cap - len < 4096) {
            cap += 4096;
            char *new_data = realloc(data,
                                     cap);

            if (new_data == NULL) {
                int old_errno = errno;
                free(data);
                close(fd);
                errno = old_errno;
                return NULL;
            }
            data = new_data;
        }
        ssize_t amount = read(fd,
                              &data[len],
                              cap - len);

        if (amount < 0) {
            int old_errno = errno;
            free(data);
            close(fd);
            errno = old_errno;
            return NULL;
        }
        if (amount == 0) {
            break;
        }
        len += amount;
    }
    close(fd);
    *len_ptr = len;
    return data;
}

```

File system manipulation system calls

Delete files: unlink(2)

```
#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

Basically just a wrapper around the getcwd system call plus some memory allocations

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

These are actually just higher-level wrappers around `open(2)`, `getdents(2)`, and `close(2)` which are themselves wrappers around the corresponding system calls

Libc crate and normal Rust

The libc crate declares all of these functions

The std::fs module has Rust-versions

- remove_file() for unlink(2)
- rename()
- create_dir() for mkdir(2)
- remove_dir() for rmdir(2)
- read_dir() for opening, reading, and closing directories

The std::env module has some other related functions

- current_dir() for getcwd(2)
- set_current_dir() for chdir(2)