CS 241: Systems Programming
Lecture 18. System Calls I

Fall 2023

Prof. Stephen Checkoway

What is an operating system?

Operating system tasks

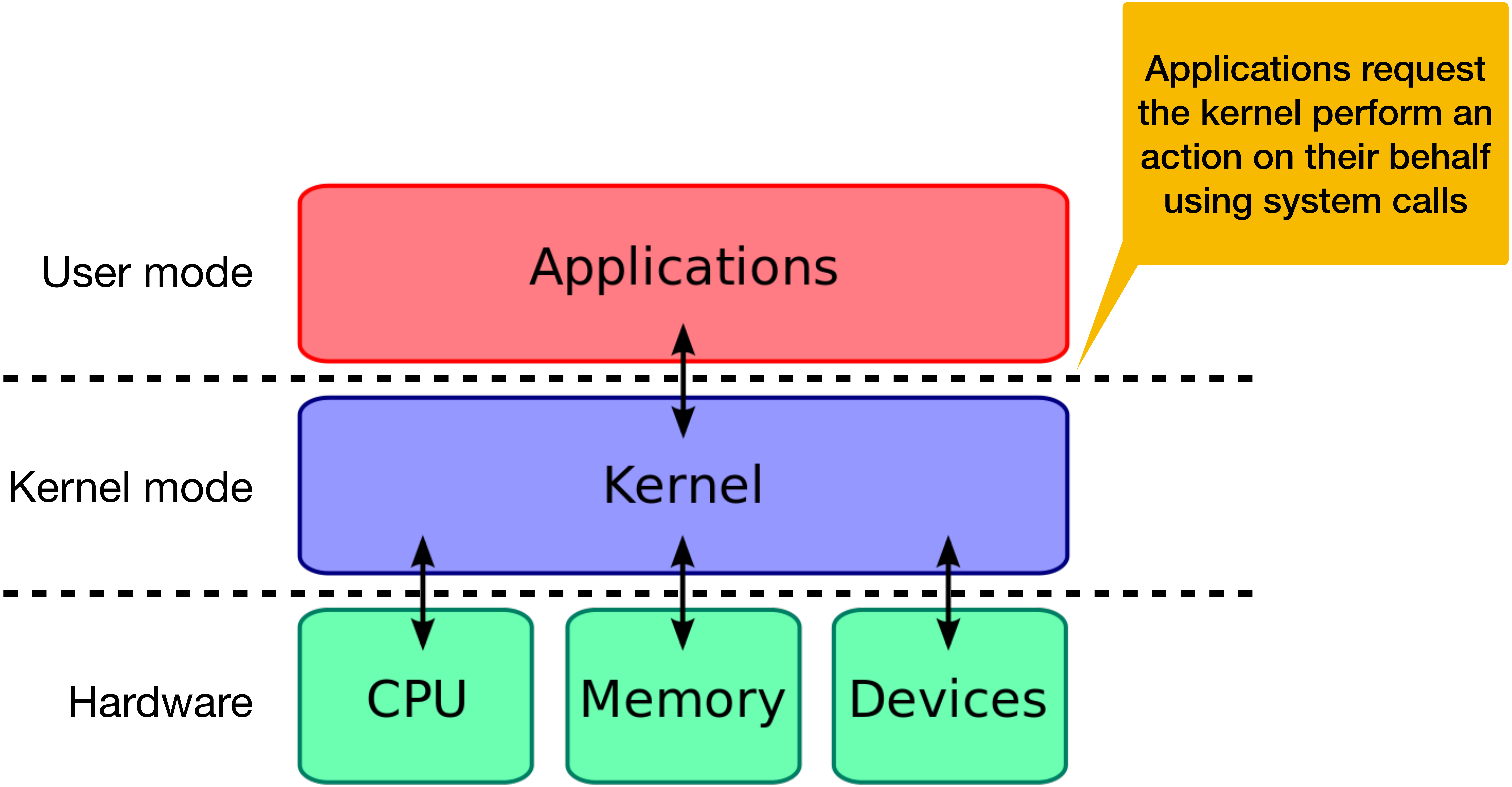
Managing the resources of a computer

- 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



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

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

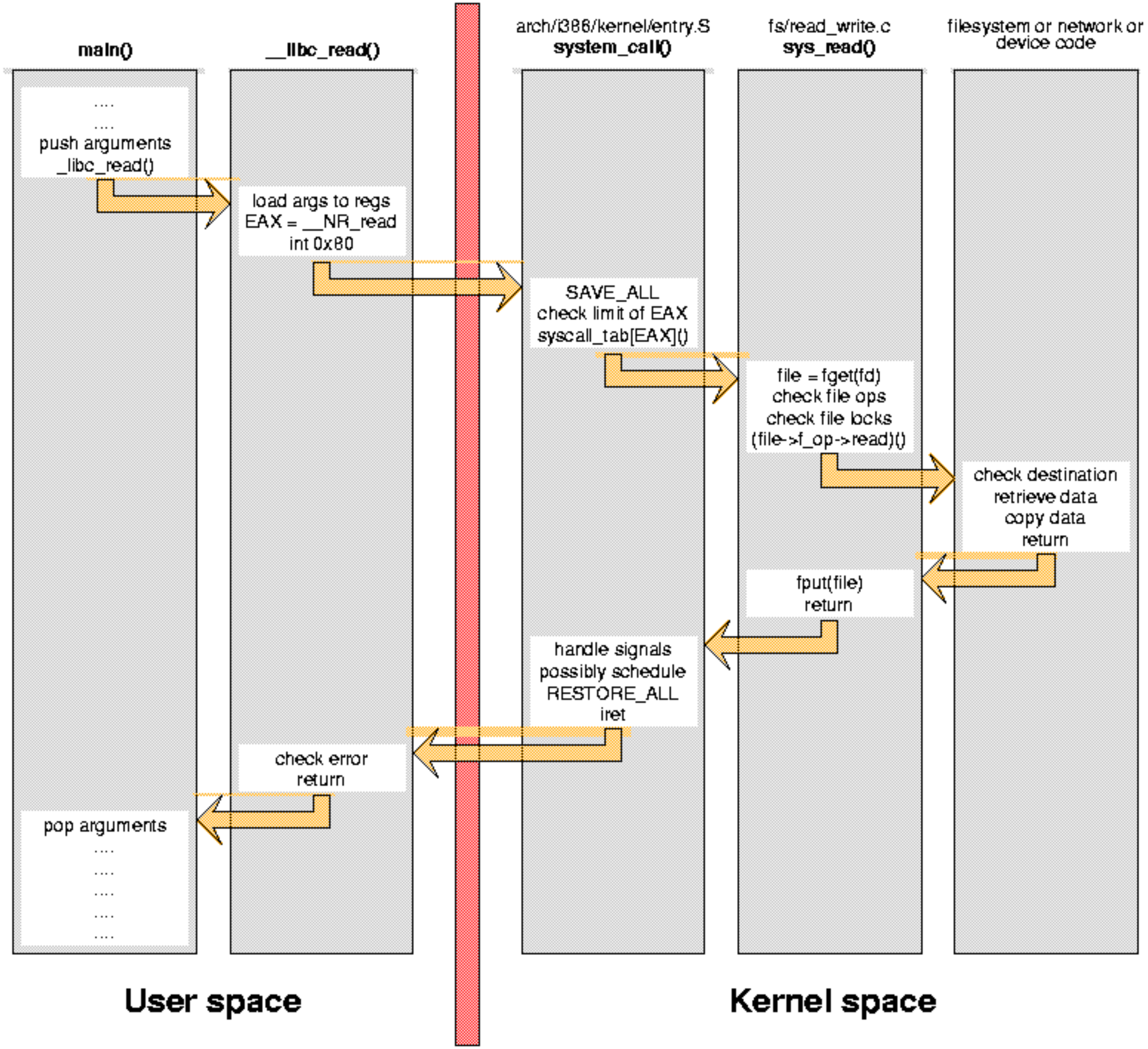
When the kernel receives an `exit` system call from a program, it

- 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



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)`

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

Unsafe Rust

Rust lets you be unsafe

Rust has an unsafe keyword that lets you perform unsafe operations

- **Call functions marked as unsafe** (including everything in the libc)
- **Dereference raw pointers** (we'll talk about these shortly)
- Modify a mutable global variable
- Implement an unsafe trait (we'll talk about traits in a few lectures)
- Access fields of a C-style union

To make system calls, we'll need unsafe for the first two

The purpose of unsafe

The compiler (and language) is *conservatively correct*

It ensures that the programs (that don't use unsafe) are memory safe

It rejects programs that are safe due to its inability to prove them safe

unsafe provides a way to bypass those limitations

unsafe limits the scope of where memory errors can occur to precisely those regions of the code marked unsafe

Unsafe functions/methods

Functions and methods can be marked as unsafe by using the unsafe keyword

Unsafe functions can only be called from within an unsafe block (or unsafe function)

```
unsafe fn does_unsafe_things() -> i32 { 0 }

fn main() {
    let x = unsafe {
        does_unsafe_things()
    };
    println!("{x}");
}
```

Functions in other languages

Rust can call functions in other languages (usually C functions)

All such external functions are unsafe and can only be called from unsafe blocks

Why did the Rust designers require that functions written in other languages be called from an unsafe block?

A. Select A when you have an answer

Raw pointers

We've seen pointers in Rust

- Shared references (e.g., `&i32`)
- Mutable references (e.g., `&mut i32`)
- Boxes
- Slices

Rust has two additional pointer types

- Constant pointer (e.g., `*const i32`)
- Mutable pointer (e.g., `*mut i32`)

Pointers

Pointers are like their reference counterparts but without some restrictions

References must always point to valid, aligned objects of the appropriate type

Additionally, mutable references may not be aliased

Pointers may be invalid (including null) or point to a misaligned object

Mutable pointers may alias

Alignment

Alignment of a value refers to its address in memory

An **aligned** value is one whose address is a multiple of its size in bytes (at least for primitive types like i32 or usize, structs are aligned at the alignment of their largest member)

A **misaligned** value is one whose address is not a multiple of its size (or its largest member)

Rust (and most programming languages) require values be aligned

This restriction comes from hardware which often doesn't support misaligned memory reads/writes or performs them more slowly

Creating a pointer from a reference

```
fn pointer_stuff(ptr: *const i32) { }

fn main() {
    let x = 10;

    // Cast the reference to a pointer
    let p = &x as *const i32;
    pointer_stuff(p);

    // Implicit conversion
    pointer_stuff(&x);
}
```

Creating a mutable pointer

```
fn pointer_stuff(ptr: *const i32) { }

fn main() {
    let mut x = 10;

    // Cast the mutable reference to a mutable pointer
    let p = &mut x as *mut i32;
    // Implicit conversion from *mut i32 to *const i32
    pointer_stuff(p);

    // Implicit conversion from &mut i32 to *const i32
    pointer_stuff(&mut x);
}
```

Reading or writing values

The reason one wants to create a pointer is to read or write the memory it points to

```
fn main() {
    let mut x = 10;
    let mut y = 20;
    let x_ptr = &mut x as *mut _;
    let y_ptr = &mut y as *mut _;

    println!("Before: x={x} y={y}");
    unsafe {
        let tmp = *x_ptr; // Read
        *x_ptr = *y_ptr; // Read + write
        *y_ptr = tmp; // Write
    }
    println!("After:  x={x} y={y}");
}
```

The _ causes the compiler to use type inference to determine the type, in this case: *mut i32

```
Output:
Before: x=10 y=20
After:  x=20 y=10
```

Pointer from a slice (or Vec or String)

```
let v: Vec<char> = vec!['🥸'; 1000];
let p: *const char = v.as_ptr();

let s = String::from("Pointers!");
let p: *const u8 = s.as_ptr();
```

Strings hold their characters UTF-8 encoded in a Vec<u8>

Gives a pointer to the first element of the slice

Use .as_mut_ptr() to get a *mut _ rather than *const _

Is the .as_ptr() method necessary or can we just cast the reference?

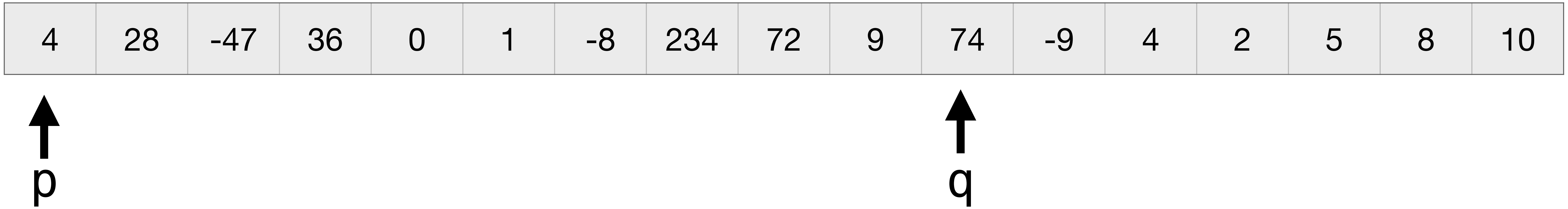
```
let s = String::from("Pointers!");
let p = &s as *const u8;
```

A. .as_ptr() is necessary

B. Casting the reference also works

C. .as_ptr() is necessary for a String but casting would work for a Vec

Creating pointers from other pointers




```rust
fn main() {
    let v = vec![4, 28, -47, 36, 0, 1, -8, 234,
                 72, 9, 74, -9, 4, 2, 5, 8, 10];
    let p = v.as_ptr();

    unsafe {
        let q = p.offset(10);
        println!("*p = {}; *q = {}", *p, *q);
    }
}
```

Null pointers

Use `std::ptr::null()` and `std::ptr::null_mut()` to create `*const _` or `*mut _`

Use `.is_null()` to test if a pointer is null

```
let ptr: *mut i32 = std::ptr::null_mut();
println!("{}", ptr.is_null());
```

libc crate

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

System call wrapper in libc	Rust declaration of the function	Notes
`void _exit(int status)`	`fn _exit(status: c_int) -> !`	Exit (doesn't return)
`pid_t getpid(void)`	`fn getpid() -> pid_t`	Get the process ID
`ssize_t read(int fildes, void *buf,` `size_t nbyte);`	`fn read(fd: c_int, buf: *mut c_void,` `count: size_t) -> ssize_t`	Read data from a file

`int rename(const char *old,`
`const char *new)`
	`fn rename(oldname: *const c_char,`
	`newname: *const c_char)`
	`-> c_int`
		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/isize`	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 *anything*
`Pointers to structs`	`Pointers to structs with the same name`		References can be converted to pointers

C strings

C considers a string to be a sequence of nonzero (often signed) bytes followed by a byte with value 0

Here's "Hello 😵‍💫"

72	101	108	108	111	32	-16	-97	-104	-75	-30	-128	-115	-16	-97	-110	-85	0

Rust considers a string to be a sequence of u8 of UTF-8 encoded characters and an associated length

72	101	108	108	111	32	240	159	152	181	226	128	141	240	159	146	171

len = 17

There's no difference between -16 and 240 other than interpretation. Both have binary value 11110000

The kernel uses C strings

The kernel, being written in C, uses C strings

More importantly, the system call interface uses C strings

Converting between a C string and a Rust string isn’t difficult but can be subtle

- Who owns the data?
- Is the string from the kernel valid UTF-8?

&CStr and CString

To pass a Rust string to the kernel, use `std::ffi::CString`

```
let cstr = CString::new(some_str)?;
let ptr = cstr.as_ptr();
```

To convert a C string into a Rust string, use `std::ffi::CStr`

```
let normal_str = CStr::from_ptr(ptr).to_str()?.to_string();
```

`CString::new()` will return an `Err(err)` if some_str contains a 0 byte

`.to_str()` will return an `Err(err)` if the `CStr` points to non UTF-8 data

```rust
use std::ffi::{CStr, CString};

type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;

fn get_home_dir_for_user(user: &str) -> Result<String> {
    let user = CString::new(user)?;
    unsafe {
        let pwd: *const libc::passwd = libc::getpwnam(user.as_ptr());

        if pwd.is_null() {
            return Err(std::io::Error::last_os_error().into());
        }
        if (*pwd).pw_dir.is_null() {
            return Err("No home directory found".into());
        }
        let home_dir = CStr::from_ptr((*pwd).pw_dir).to_str()?.to_string();
        Ok(home_dir)
    }
}
```

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)

```
#include <fcntl.h>

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

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

- `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 `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

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!

```rust
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()

```c
#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)

```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

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), getdirents(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)