Operating Systems:

System Calls

CS400

Week 2: 21st Jan

Spring 2020

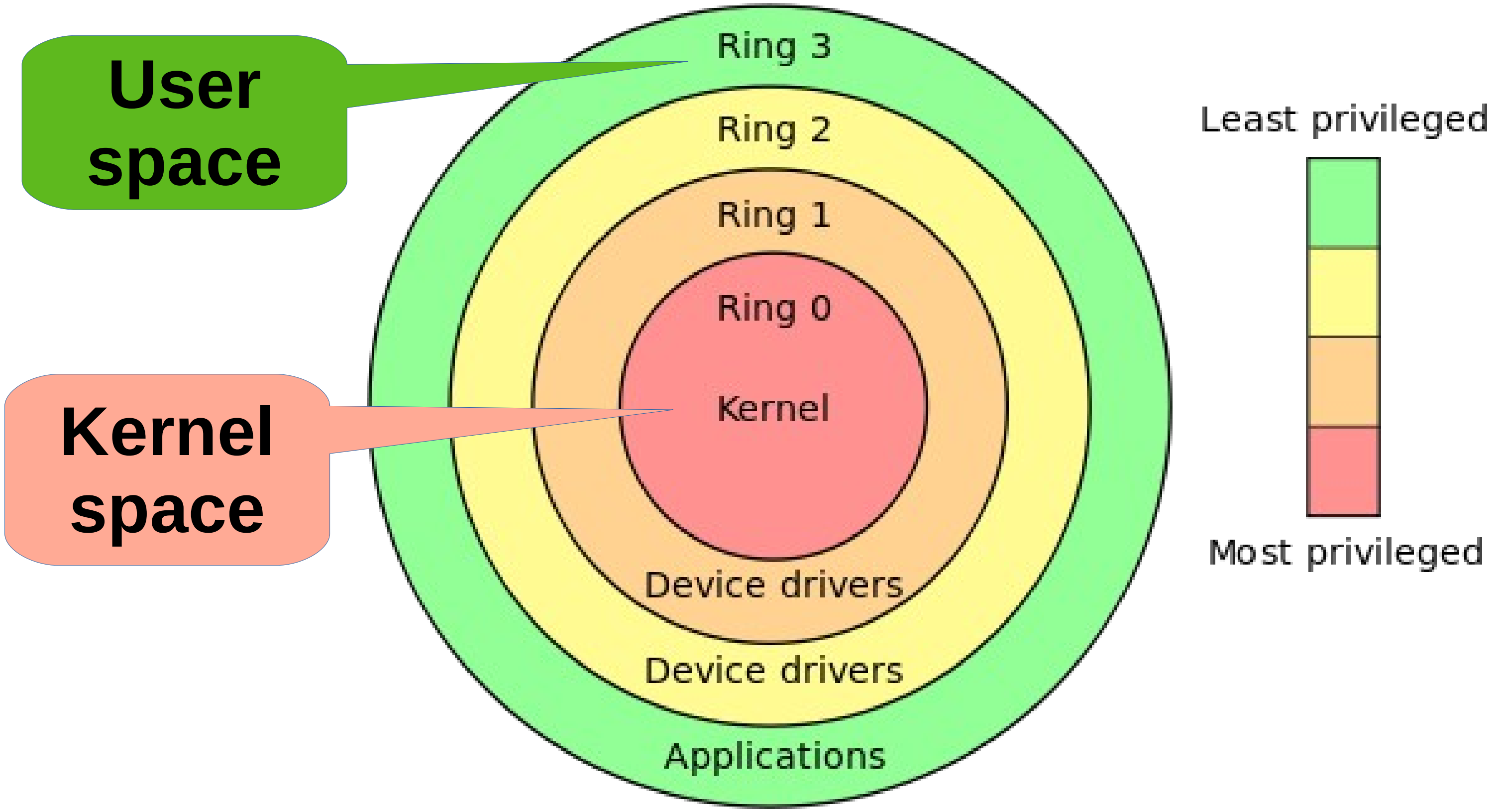
Oliver BONHAM-CARTER

Introduction

- What is the difference between the *User space* and the *Kernel space*?
- What happens when code “talks” to the kernel?
- How is a system task requested to be processed by the kernel?
- How does the kernel signal the user upon completion?

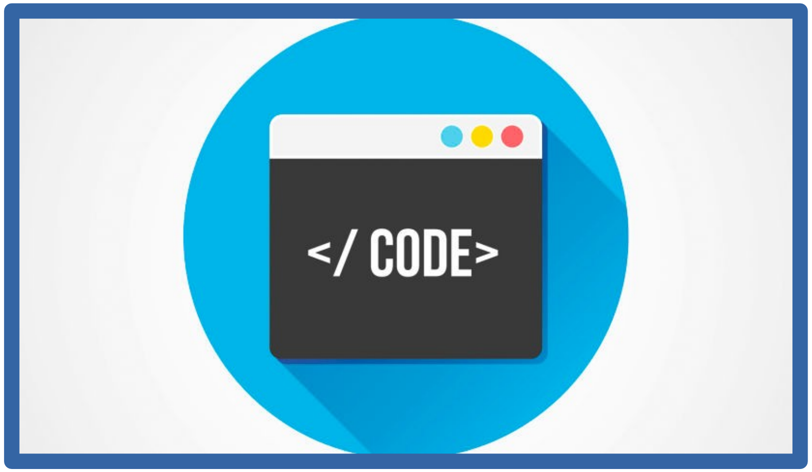
User and Kernel Action Rings

- Rings Model: software has priority for resources.
- Level of trust are enforced by allowing SCs
- To prevent software from accessing resources OS enforces privilege by requiring SCs be used.



Privileges in Code

- Running software limited to own address space.
- Low-privilege software is prevented from directly accessing hardware.
- Since reading, writing files is often necessary (for example), the OS (the highest level of privilege) accepts requests for reading and writing tasks.



Interrupts: Requests For Resources

- Interrupts (signals to the processor) are emitted by hardware or software for immediate (quick) action.
- High priority case: OS interrupts other processes to prepare the resources for the software, hardware which uses the interrupt.
  - Saving other states of processing
  - Prepping system to handle event
- After completion, the system returns to former activities

Communication between User and Kernel Space

User Space
Browser,
Music player,
Spreadsheet,
Etc...

OS
Connects The User Space to the Kernel Space

- How does the User space signal the Kernel space and vice versa?

- **System calls:** requests for the operating system to do something on behalf of the user's program and are functions used in the kernel itself

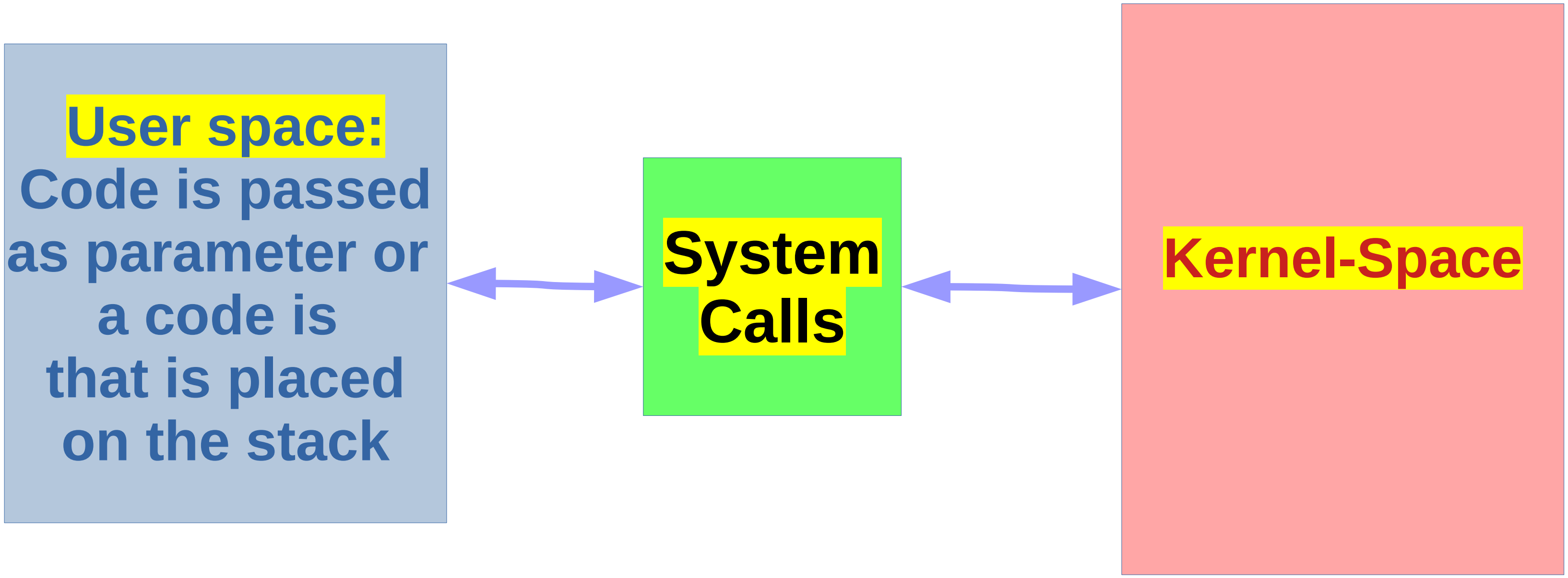
Kernel Space
Processes,
Memory management,
Storage,
Etc...

Actors of Calls

- **User Space**: A system call appears as a normal C function call.
- **Kernel Space**: a system call executes code in the kernel and so there must be a mechanism to change the mode of a process from User mode to Kernel mode.
- C compiler uses a predefined library of functions (the C library) that have the names of the system calls.
- Library functions invoke instructions that change the process execution mode from User mode to kernel mode and causes the kernel to start executing code to handle system calls.

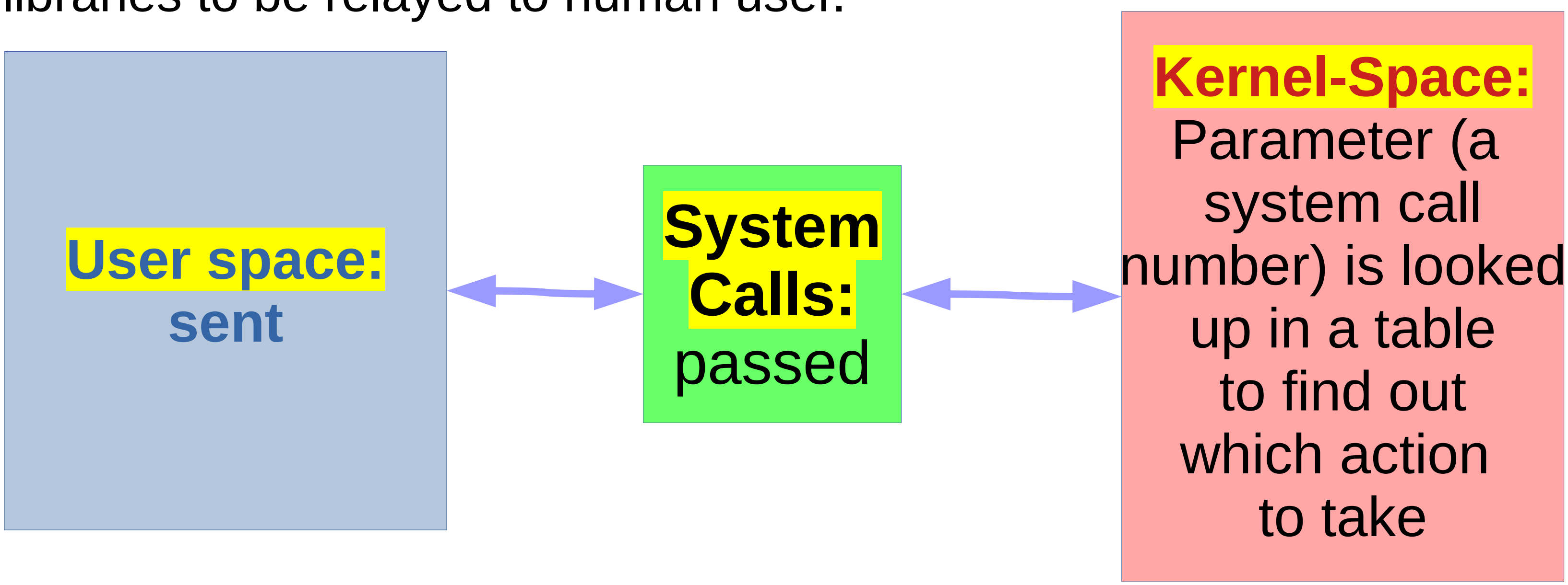
Invoking a Kernel Action

- User space processes do not directly access Kernel resources. Instead SCs are made to direct Kernel actions
- To invoke a system call, library functions pass the kernel a unique machine dependent number as a parameter via code, or is placed on the stack for the kernel to process.



Invoking a Kernel Action

- The kernel looks up the call identifier number
- The kernel calculates the user address of the first parameter of the SC
- The user parameters are copied to the *u-area* while the relevant SC routine is called.
- Errors are detected, clean-up in memory and exit codes are returned to libraries to be relayed to human user.



File Structure Related Calls

SPECIFIC CLASS	SYSTEM CALL
Creating a Channel	creat()
	open()
	close()
Input/Output	read()
	write()
Random Access	lseek()
Channel Duplication	dup()
Aliasing and Removing Files	link()
	unlink()
File Status	stat()
	fstat()
Access Control	access()
	chmod()
	chown()
	umask()
Device Control	ioctl()

UNIX

Process Related Calls

Process Creation and Termination	exec() fork() wait() exit()
Process Owner and Group	getuid() geteuid() getgid() getegid()
Process Identity	getpid() getppid()
Process Control	signal() kill() alarm()
Change Working Directory	chdir()

UNIX

InterProcess Communication

Pipelines	pipe()
Messages	msgget()
	msgsnd()
	msgrcv()
	msgctl()
Semaphores	semget()
	semop()
Shared Memory	shmget()
	shmat()
	shmdt()

UNIX

Common POSIX SCs

- A set of formal descriptions that provide a *standard* for the design of operating systems, especially ones that are compatible with Unix (and Linux).

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Common POSIX SCs

Directory- and file-system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

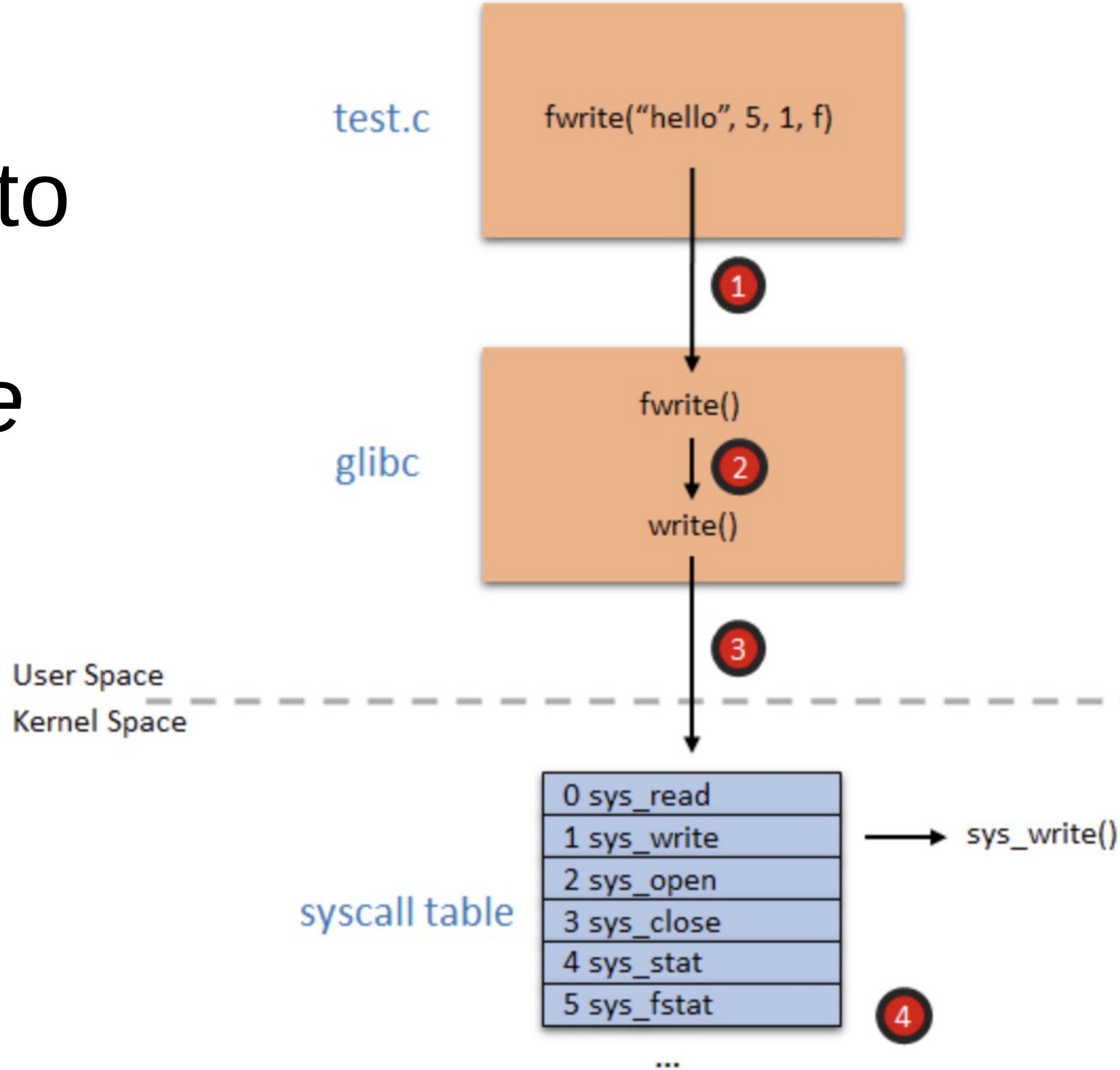
Return code s is -1 if error has occurred.

- Pid = process id
- Fd = file descriptor
- N = byte count
- Position = offset in file
- Seconds = time unit

The *write* Library Call (*fwrite*)

- Declaration for fwrite() function
  - **`size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`**
- Three parameters:
  - ptr − This is the pointer to the array of elements to be written.
  - size − This is the size in bytes of each element to be written.
  - nmemb − This is the number of elements, each one with a size of size bytes.
  - stream − This is the pointer to a FILE object that specifies an output stream.

Steps to Call *fwrite*



Steps to Call *fwrite*

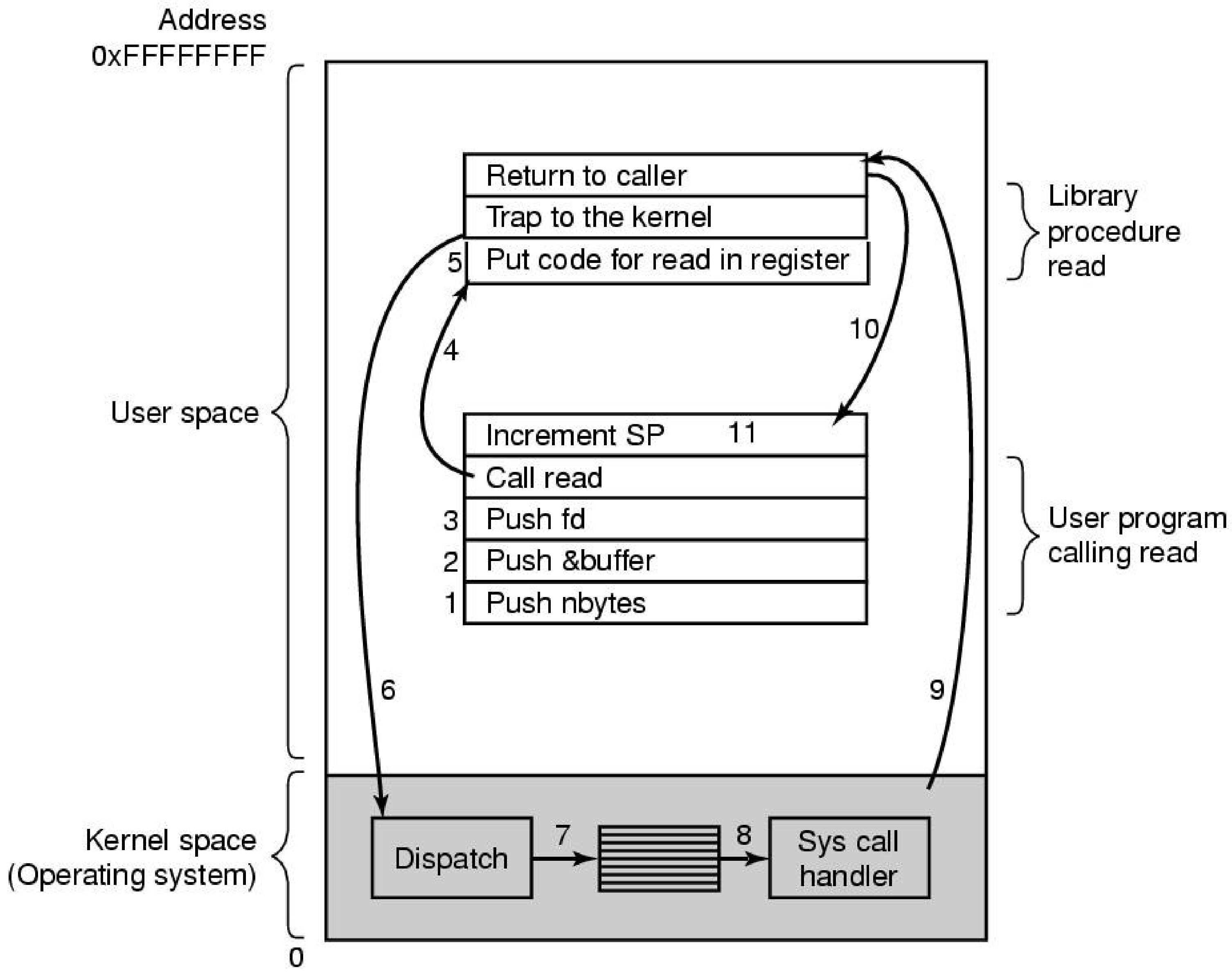
- fwrite, together with the rest of the C standard library, is implemented in glibc*, which is one of the core components of the Linux operating system.
- fwrite is essentially a wrapper for the write library call.
- write will load the system call ID (which is 1 for write) and arguments into the processor registers, and then cause the processor to switch to kernel level. The way this is done depends on the processor architecture, and sometimes on the processor model. For example, x86 processors usually call interrupt 80, while x64 processors use the syscall processor instruction.
- the processor, now executing in kernel space, feeds the system call ID to the syscall table, extracts the function pointer at offset 1 and calls it. This function, sys_write, is the kernel implementation of writing a file.

Ref: https://sysdig.com/blog/fascinating-world-linux-system-calls/

The *Read* Call

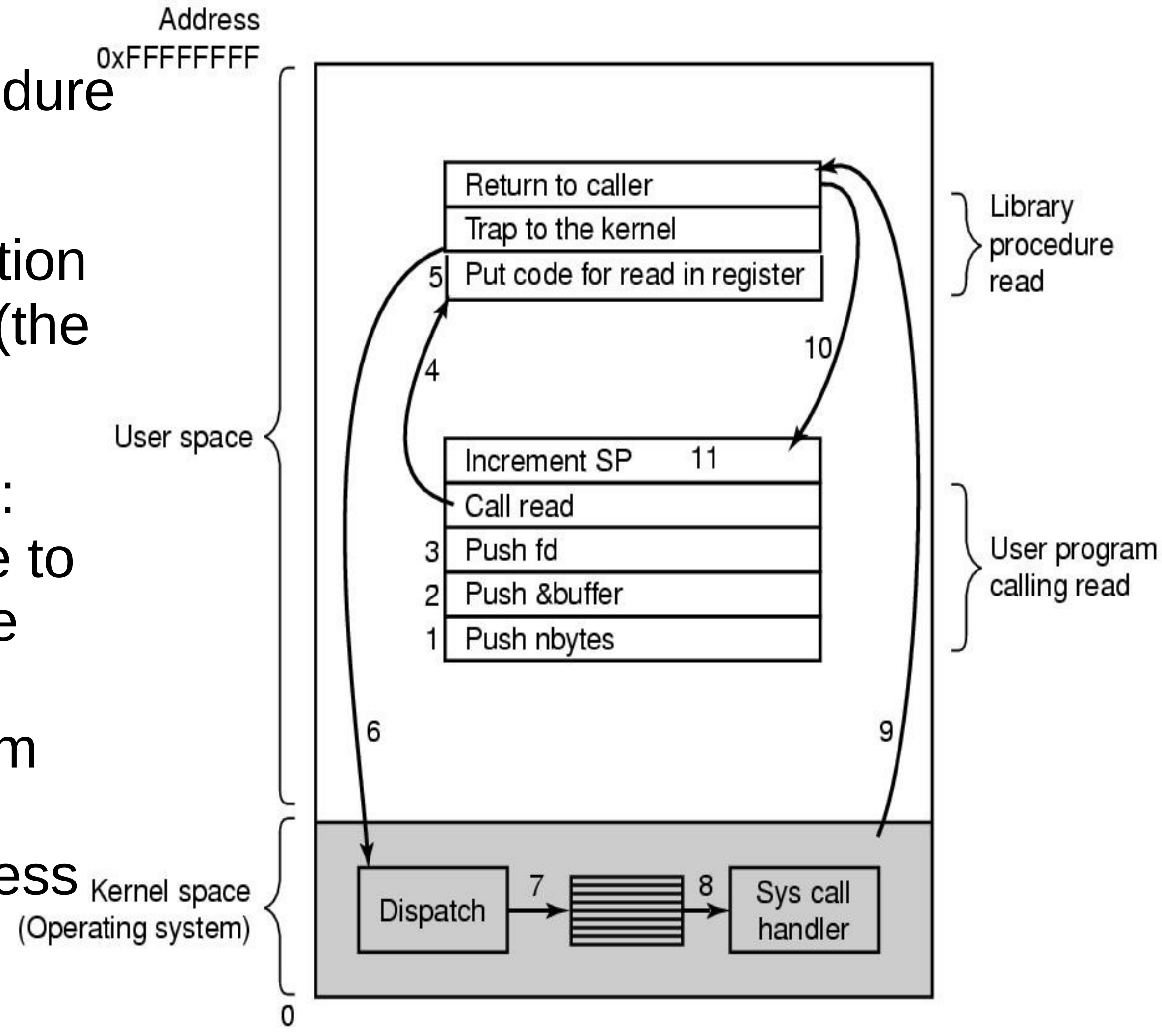
- C code to read data from a file and save result to variable, `count`
  - `Count = read(fd, buffer, nbytes);`
- **Three parameters**:
  - *Fd* = filename
  - *Buffer* = pointer (address) of buffer, not contents of the buffer
  - *Nbytes* = number of bytes to read

Steps to Make a *Read* Call



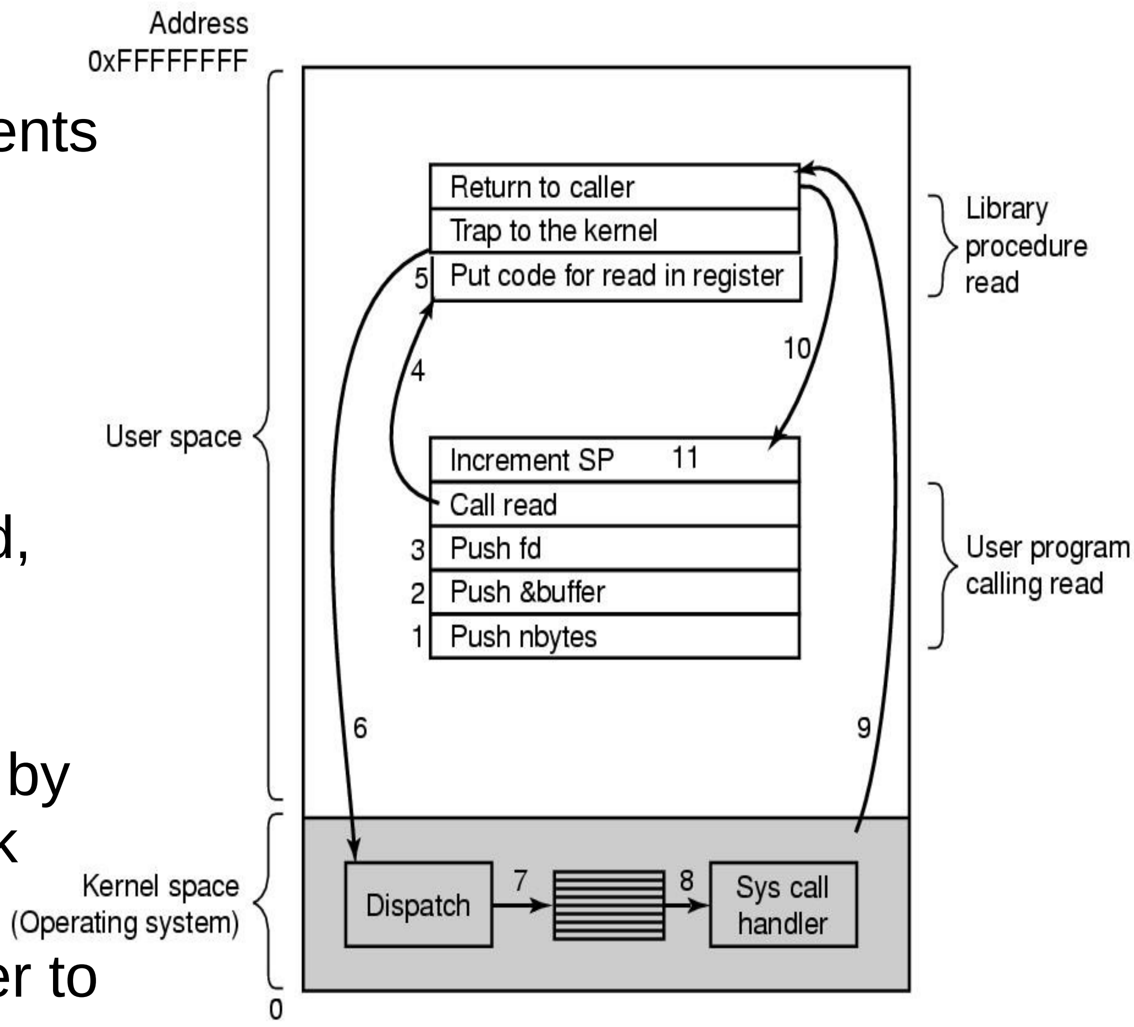
Steps to Make a *Read* Call

- {1,2,3} – Push parameters into stack
- {4} – Call library procedure for reading files
- {5} – Place the instruction to read into a register (the OS's *to-do* list)
- {6} – TRAP instruction: switch from user mode to kernel mode where the instruction can be executed. Execute from fixed address within kernel. Save this address for later on stack.

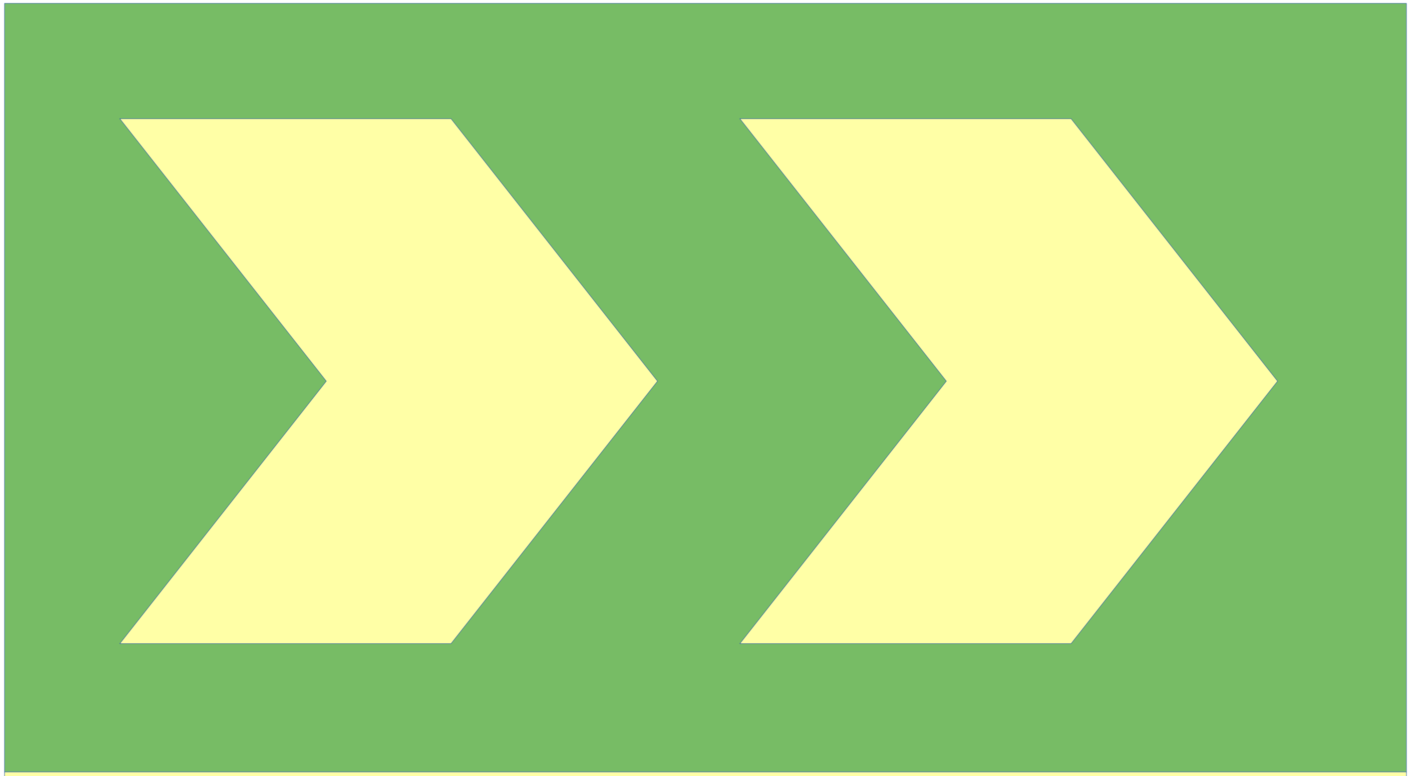


Steps to Make a *Read* Call

- {7} – Local the local system-call handler (a table of pointers to agents who perform the read task)
- {8} – Run system call
- {9, 10} – Kernel space work is now completed, return result to user space
- {11} – Clean the stack by removing the read task and parameters. Increment stack pointer to remove process.



Let’s Code and Run Some Calls!



Up Next!

Please Install
Your Software

- We will be using Git and GitHub. Please setup your account at https://github.com/ and also download a Git client software from https://git-scm.com/downloads
- We will also be using the Atom editor to write code. Please download and install your editor from https://atom.io/
- For most labs, we will be using Docker. Please download and install your Docker Desktop installation (note: not the Docker ToolBox) from https://www.docker.com/. Help: https://hub.docker.com/
- If necessary, please help each other to install this software. Or see the department's Technical Leaders with questions.
- An online C editor and compiler. Note: some code may not run due to security reasons.
  - https://www.tutorialspoint.com/compile_c_online.php

Dockerfile

(Docker Desktop)

Creates your container
to compile and run your
c++ code.

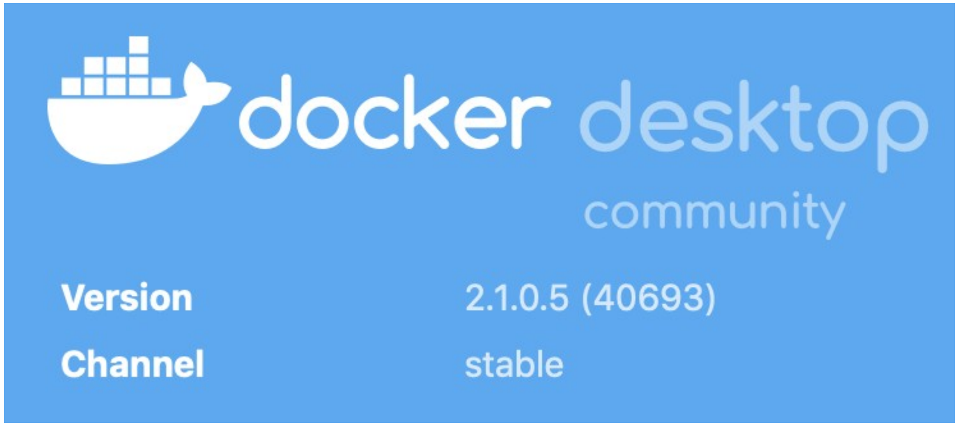
File: sandbox/Dockerfile

```
# date: 21 Jan 2020
# gcc development

FROM ubuntu

RUN apt-get update

RUN \
	apt-get update &&\
	apt-get install -y git &&\
	apt-get install -y htop &&\
	apt-get install -y vim &&\
	apt-get install -y strace &&\
	apt-get install -y gcc &&\
	apt-get install -y g++-8-i686-linux-gnu


RUN useradd gccdev
RUN mkdir /home/gccdev
RUN export HOME=/home/gccdev

WORKDIR /home/gccdev

# Define default command.
CMD ["bash"]
```

Commands to Run
From (Linux) Bash

- Build the container :
  - docker build -t gccdev .

- Run the container :
  - docker run -it gccdev



- Mount local drive and run container :
  - docker run -it --mount type=bind,source=$PWD,target=/home/gccdev gccdev

Note: the directory where you run this becomes your local directory in the container.

Hello World in Assembly

- Print “HelloWorld” as a system call.
- Locate, ***helloWorld.s***
- Run command (on ***Linux***):
- **gcc -c helloWorld.s**
- **ld -o helloWorld helloWorld.o**
- **./helloWorld**

File: sandbox/helloWorld.s

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start

_start:
    movq  $1, %rax
    movq  $1, %rdi
    movq  $msg, %rsi
    movq  $len, %rdx
    syscall

    movq  $60, %rax
    xorq  %rdi, %rdi
    syscall
```

Hello World in Assembly

- ***.data*** stores initialized data of our program (a string and its length)
- ***.text*** contains the code of our program
- **syscall** is a SC

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start

_start:
    movq  $1, %rax
    movq  $1, %rdi
    movq  $msg, %rsi
    movq  $len, %rdx
    syscall

    movq  $60, %rax
    xorq  %rdi, %rdi
    syscall
```

Hello World in Assembly

- Kernel space code
- **_start**:

  rax = the register of the handler.

  Rdi, rsi, rdx = parameters for **write** command found in **syscall.**

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start

_start:
    movq  $1, %rax
    movq  $1, %rdi
    movq  $msg, %rsi
    movq  $len, %rdx
    syscall

    movq  $60, %rax
    xorq  %rdi, %rdi
    syscall
```

What is, *syscall*?

- **Syscall** invokes an handler at privilege 0 (Kernel space) and checks general purpose registers (i.e. mechanisms that perform system functions.)
- See Torvald’s github for many, many examples of *Syscall* in use
- https://github.com/torvalds/linux/blob/master/fs/read_write.c

Example of Torvald’s use of *sysCall*

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
        struct fd f = fdget_pos(fd);
        ssize_t ret = -EBADF;

        if (f.file) {
                loff_t pos = file_pos_read(f.file);
                ret = vfs_write(f.file, buf, count, &pos);
                if (ret >= 0)
                        file_pos_write(f.file, pos);
                fdput_pos(f);
        }

        return ret;
}
```

Tracing the Call

- We use the **strace** diagnostic to follow the userspace interactions with the kernel space.
- Use: **strace ./helloWorld**

```
execve("./helloWorld", ["./helloWorld"],
[/* 17 vars */]) = 0
write(1, "Hello, world!\n", 14Hello, world!
)          = 14
Exit(0)                             = ?
+++ exited with 0 +++
```

- **execve** and **write** = system calls to execute program, write output