Operating Systems
CS 415

Lecture 3:

System Calls

Suyash Gupta

Assistant Professor

Distopia Labs and ORNG

Dept. of Computer Science

(E) suyash@uoregon.edu

(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Announcements

- **Suyash Gupta**
  - Office Hours: Thursday, 10-11am, Deschutes 334

- **Nihal Balivada (TA)**
  - Office Hours: Wednesday/ Friday, 11-12pm, Deschutes 335

- **Ranjitha Rani (TA)**
  - Office Hours: Monday /Tuesday, 1-2pm, Deschutes 229

Assignment 1 is Out!

- **Deadline → April 21, 2026 at 11:59pm PST**
- Please start working and talk to us if you are stuck.

- **Midterm → April 28, 2026 (in-class)**
  - Closed book, no cheat sheets, no discussions.

Last Class

- OS Structures
- Interrupts

System Call

- A software-triggered entry into the kernel.
- **Explicitly** requested by the program.
- Provides access to the services made available by an operating system.

System Call Flow

User Mode

Kernel Mode

Program Running

System Call Flow

User Mode

Kernel Mode

Program Running

Syscall Arrives

System Call Flow

User Mode

Kernel Mode

Program Running

Syscall Arrives

Kernel executes syscall code

System Call Flow

User Mode

Program Running

Syscall Arrives

Kernel Mode

Kernel executes syscall code

Kernel Returns

System Call Flow

User Mode

Kernel Mode

Program Running

Syscall Arrives

Kernel executes syscall code

Kernel Returns

Program Resumes

System Call Example

- How can you copy data from one file to another?

File Copy: Unix

- How can you copy data from one file to another?
- Using Unix File Copy: **cp file1.text file2.text**
- Writing a C or C++ Program
- Using GUI to copy a file.

File Copy

- How can you copy data from one file to another?

File Copy

- **User Mode: Parse arguments/setup**
  - **cp** is a user-space binary.
  - validates arguments
  - sets up buffers (e.g., 128KB chunks).
  - Example → You want to “copy novel A to B?” and grab your writing desk (buffer)

File Copy

- **Kernel Mode: Stat source**
  - ***cp.c*** calls ***stat() / lstat()*** syscall → switch to kernel mode.
  - Kernel retrieves metadata (size, perms)
  - More like collecting statistics about the file.
  - Data copied from kernel to user space → returns to user mode.
  - Librarian hands you the summary card: “5,000 pages, 2kg, last printed 2025, hardcover.” No reading the story yet.

File Copy

- **Kernel Mode: Open source file**
  - ***cp.c*** calls ***openat(O_RDONLY)*** syscall → switch to kernel mode.
  - Kernel opens src file.
  - Returns a File descriptor (FD) → switch to user mode.
    - A small integer (like ticket #3) your program uses to refer to an open resource.
    - Librarian unlocks novel A so you can read it.

File Copy

- **Kernel Mode: Open/Create Destination file**
  - *cp.c* calls *openat(O_WRONLY|O_CREAT)* syscall → switch to Kernel mode.
  - kernel creates a file if does not exist.
  - If already exists, truncates the size to 0.
  - returns FD for this file → switch to user mode.
  - Librarian gives you blank notebook B to write into.

File Copy

- **User Mode: Enter Copy Loop**
  - Allocates buffers
  - Enters copy loop based on file size → ***while(pages_left)***.
  - You plan “while pages remain, copy handfuls at a time.”

File Copy

- **Kernel Mode: Read Chunk from Source**
  - ***cp.c*** calls ***read(src_fd, buf, 128KB)*** syscall → switch to kernel mode.
  - Copying a block of data → return to user mode.
  - Librarian photocopies 128 pages of novel A into your hands.

File Copy

- **User Mode: Buffer handling**
  - You prepare to write
  - You hold the 128-page photocopy, ready to transcribe.

File Copy

- **Kernel Mode: Write Chunk to Destination**
  - ***cp.c*** calls ***write(dest_fd, buf, 128KB)*** syscall → switch to kernel mode.
  - Actual writing happens → return to user mode.
  - Librarian staples your 128-page handwriting into notebook B.

File Copy

- **Kernel Mode: Finalize and Sync**
  - Once writing completes, user calls a series of syscalls to flush pages to disk and close files.
  - fdatasync(dest_fd=4) → flush dirty pages to disk
  - fstat(dest_fd=4) → verify final size matches source
  - close(src_fd=3) → release source file
  - close(dest_fd=4) → release destination file
  - Librarian checks notebook B complete? + shelves both books.

File Copy

- **Kernel Mode: Post-copy and Exit**
  - Flush pages to disk and close files
  - fdatasync(dest_fd=4) → flush dirty pages to disk
  - fstat(dest_fd=4) → verify final size matches source
  - close(src_fd=3) → release source file
  - close(dest_fd=4) → release destination file
  - ou copy novel A's cover details to notebook B, declare "done!"

Application Programming Interface

Application Programming Interface

- Systems execute thousands of system calls per second.
- Most programmers never see this level of detail.
- What you see (open/read/write) - the “interface”.
- Programmers follow an application programming interface (API).
- API specifies: Available set of functions, function parameters and return values.
- Three of the most common: Windows API, POSIX and the Java API.
- API are accessed through libraries
  - C library → libc.

Syscall Wrappers

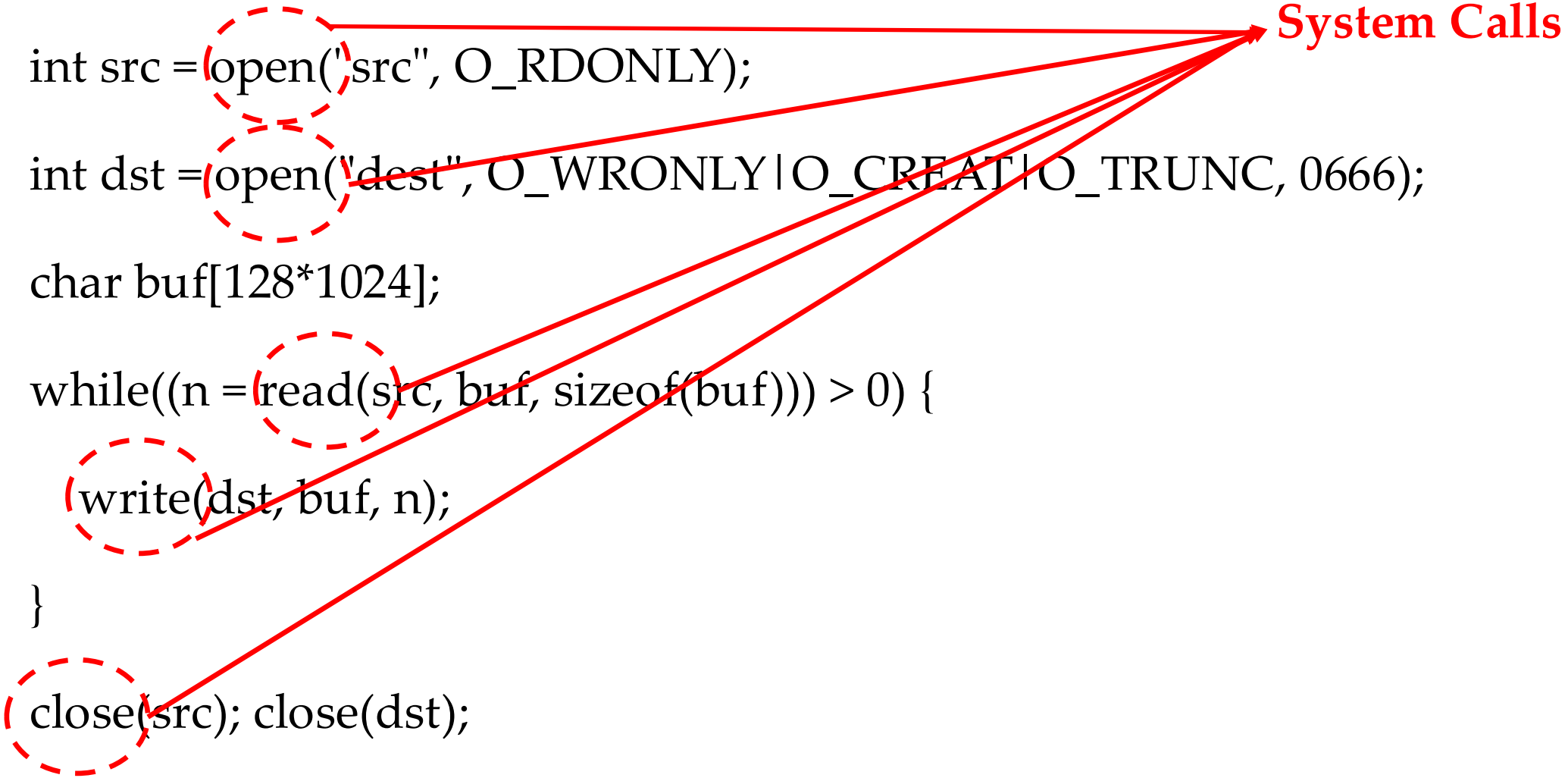
Syscall Wrappers

- Wraps and translates user API calls to OS specific system calls.
- How it works internally - the “translator”.
- It is a machine code generator.

Syscalls, API and Wrappers

```
int src = open("src", O_RDONLY);

int dst = open("dest", O_WRONLY|O_CREAT|O_TRUNC, 0666);

char buf[128*1024];

while((n = read(src, buf, sizeof(buf))) > 0) {

    write(dst, buf, n);

}

close(src); close(dst);
```

Syscalls, API and Wrappers



Syscalls, API and Wrappers

This is provided to you **libc POSIX API**

int src = open("src", O_RDONLY);

Syscalls, API and Wrappers

This is provided to you **libc POSIX API**

int fd = open("src", O_RDONLY);

libc does all the hard task of translating for you, as a wrapper.

```
int fd;
asm volatile (
    "mov $257, %%rax\n"    // Syscall #257 = openat
    "mov $-100, %%rdi\n"   // fd=AT_FDCWD
    "mov %1, %%rsi\n"      // path="src"
    "mov $0, %%rdx\n"      // O_RDONLY=0
    "syscall\n"
    : "=a"(fd)             // return fd
    : "r"(path) );
```

How does libc handle Syscall?

Syscall Dispatch Table

- Includes an entry for each syscall.
- A syscall handler function that specified how to execute the syscall.

0	**sys_restart_syscall**
63	**sys_read**
64	**sys_write**
257	**sys_openat**

What about Syscall parameters?

Registers for Loading Parameters

- Syscalls carry parameters (path, flags, etc.).
- User space (libc) loads them into **CPU registers** before the syscall instruction.
- Example registers: rax, rdi, rsi.

Syscall Flow

- You write: open(“src”, O_RDONLY)

Syscall Flow

- You write: open(“src”, O_RDONLY)
- Compiler: calls libc_open() function

Syscall Flow

- You write: open(“src”, O_RDONLY)
- Compiler: calls libc_open() function
- libc_open() → openat(AT_FDCWD, “src”)

Syscall Flow

- You write: open(“src”, O_RDONLY)
- Compiler: calls libc_open() function
- libc_open() → openat(AT_FDCWD, “src”)
- libc_openat() → INLINE_SYSCALL(openat)

Syscall Flow

- You write: open(“src”, O_RDONLY)
- Compiler: calls libc_open() function
- libc_open() → openat(AT_FDCWD, “src”)
- libc_openat() → INLINE_SYSCALL(openat)
- Find the syscall number in the table (say 257)

Syscall Flow

- You write: open(“src”, O_RDONLY)
- Compiler: calls libc_open() function
- libc_open() → openat(AT_FDCWD, “src”)
- libc_openat() → INLINE_SYSCALL(openat)
- Find the syscall number in the table (say 257)
- Load the syscall number and parameters in registers

Syscall Flow

- You write: open(“src”, O_RDONLY)
- Compiler: calls libc_open() function
- libc_open() → openat(AT_FDCWD, “src”)
- libc_openat() → INLINE_SYSCALL(openat)
- Find the syscall number in the table (say 257)
- Load the syscall number and parameters in registers
- Check table: dispatch_table[257] → syscall handler function

Syscall Flow

- You write: open(“src”, O_RDONLY)
- Compiler: calls libc_open() function
- libc_open() → openat(AT_FDCWD, “src”)
- libc_openat() → INLINE_SYSCALL(openat)
- Find the syscall number in the table (say 257)
- Load the syscall number and parameters in registers
- Check table: dispatch_table[257] → syscall handler function
- OS jumps to that function to execute that function

Types of Syscalls

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Linkers and Loaders

Linkers

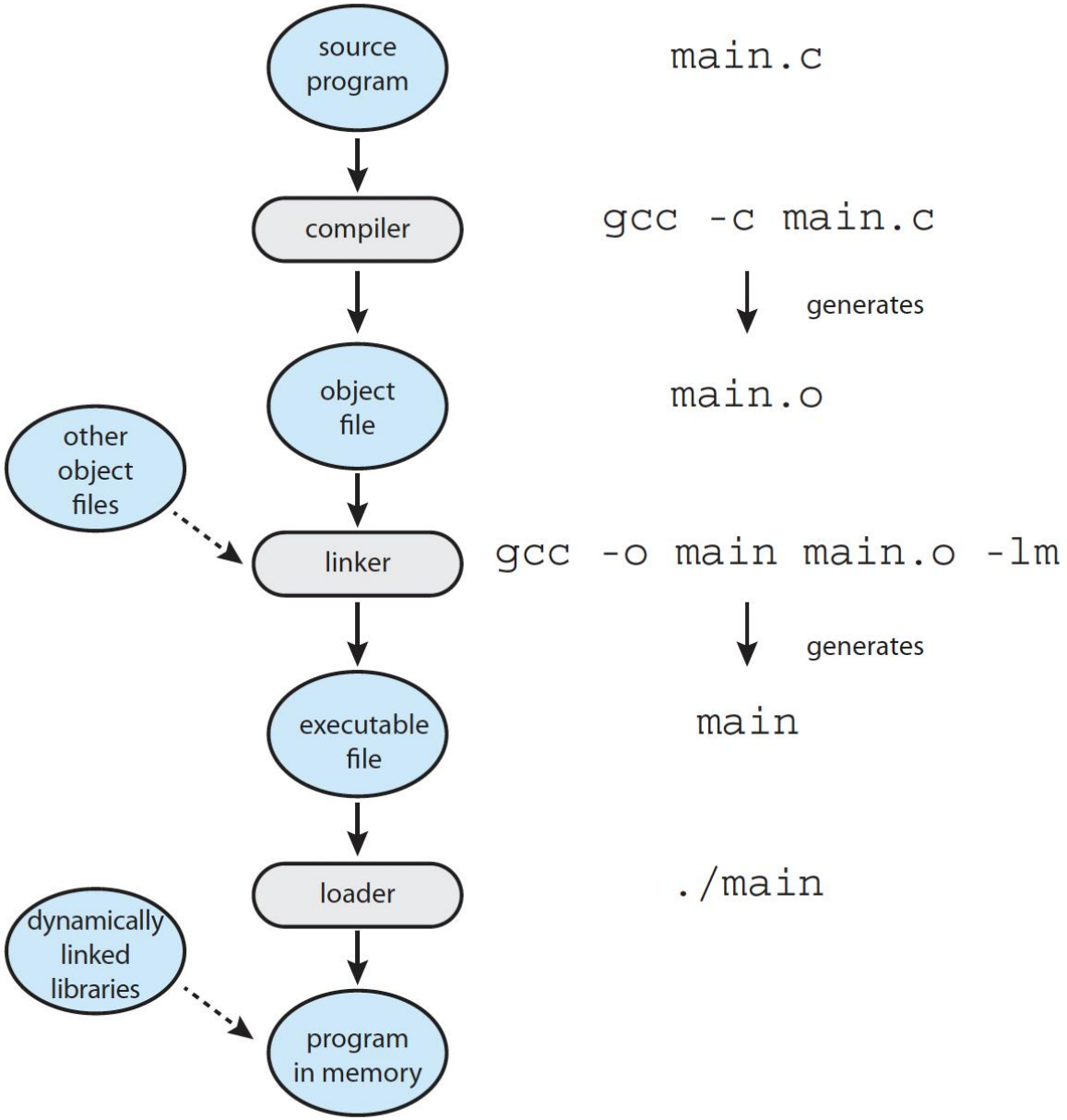
- Source files are compiled into object files → loaded into physical memory.
  - A relocatable object file.
- Linker combines these relocatable object files into a single binary executable file.
- Linking phase also includes other object files or libraries may be
  - Example: standard C or math library.

Loaders

- Loader loads the binary executable file into memory.

- Now, it can run on a CPU core.

Linker and Loader Flow



Are applications OS specific?

Are applications OS specific?

- Yes!
- **Apps call OS specific functions**: (WinAPI on Windows, POSIX on Linux).
  - Different names/signatures → CreateFile() vs open()
- **Binary Format**: PE (.exe) for Windows, ELF for Linux, Mach-O for macOS.
  - OS loader expects its format.
- **System calls and libraries**: Link to kernel32.dll (Windows) or libc.so (Linux) incompatible.