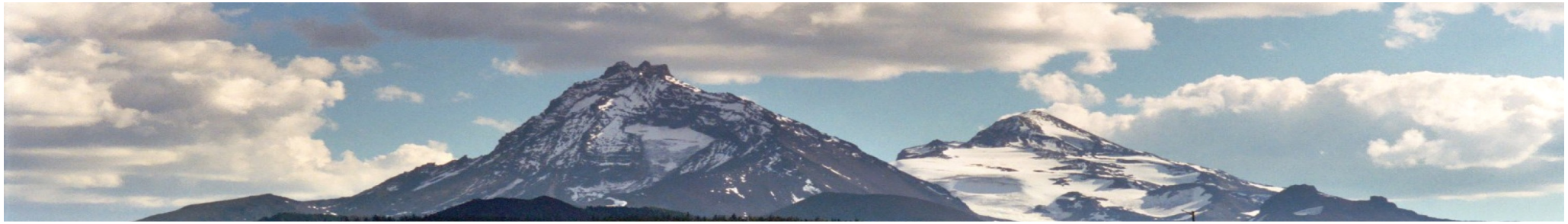


CS 415
Operating Systems
System Calls

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

Logistics

- Project 1 posted
- Labs
  - Intended to support projects
  - Lab 2 posted on Monday
- Start reading OSC Chapter 3
- Make use of Canvas discussion
- Get familiar with the Linux man pages

  https://man7.org/linux/man-pages/index.html

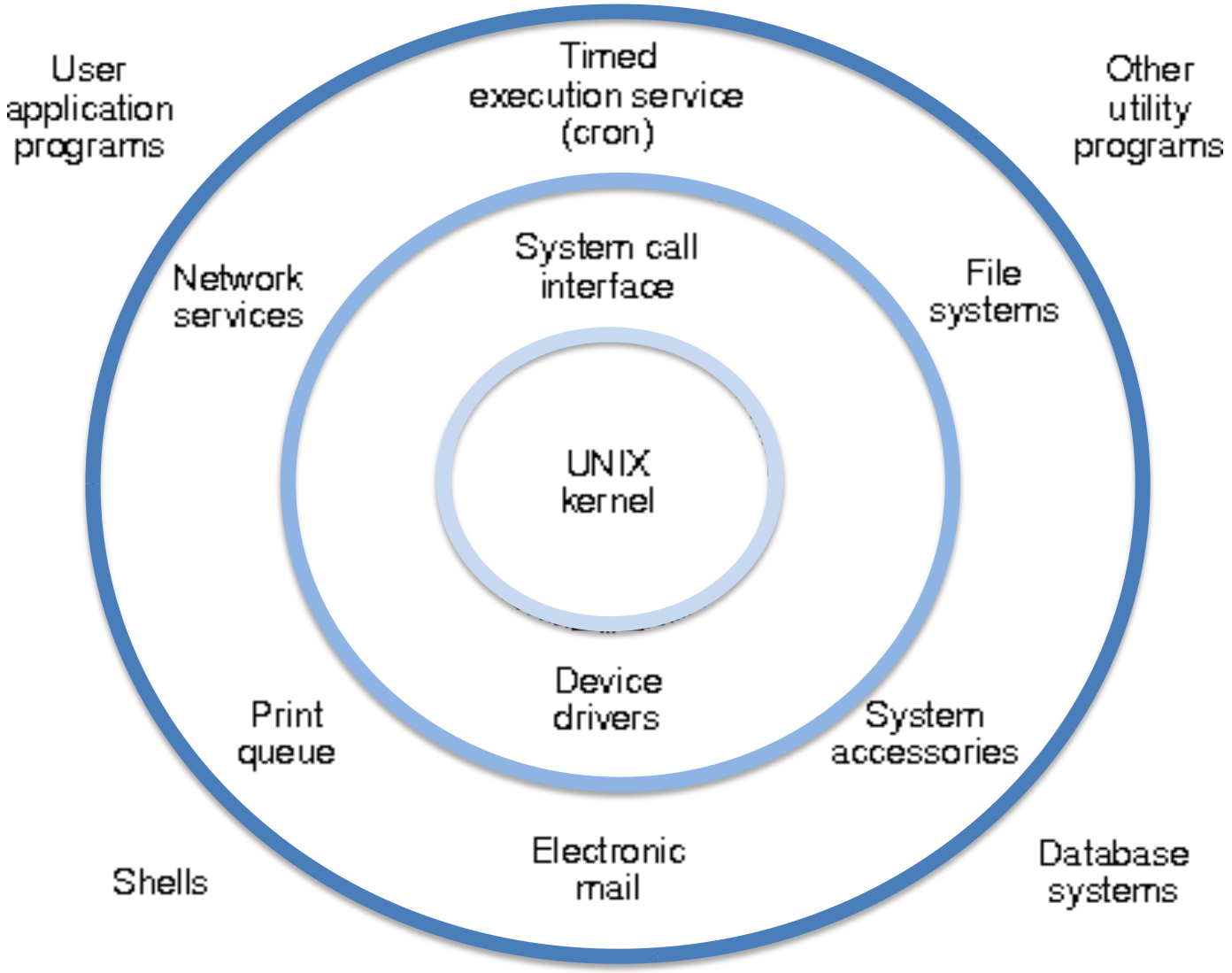
Outline

- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation

Objectives

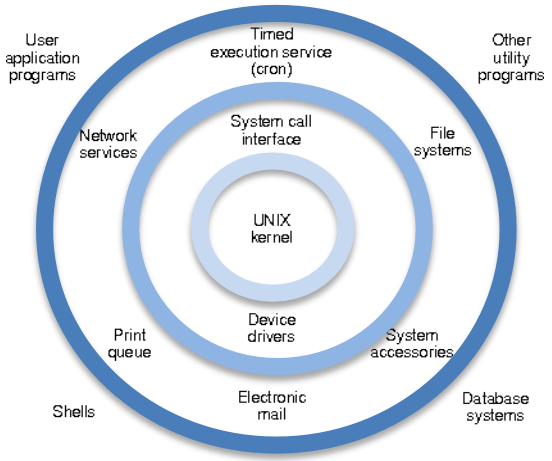
- To describe the services an operating system provides to users, processes, and other systems
- To explain how operating systems handle system calls

Operating System Layers



System Layers

- Application
- Libraries (in application process)
- System Services
- OS API
- Operating system kernel
- Hardware



Applications to Libraries

- Application programming interface (API)
- Libraries
  - Example: *libc*
- Library routines
  - Example: *printf()* of *stdio.h*
- All within the process's address space
  - Statically linked
    - libraries are included as part of the application code
    - calls are resolved at compile time
  - Dynamically linked
    - libraries are loaded by the OS at execution time as needed
    - jump tables and pointers are resolved dynamically by linker

Libraries to System Routines

- System call interface
  - UNIX man pages, section 2
  - Examples
    - *open()*, *read()*, *write()* – defined in *unistd.h*
  - Call these via libraries? *fopen()* vs. *open()*
  - See links in schedule
- Special files
  - Drivers
  - */proc*
  - *sysfs*

System to Hardware

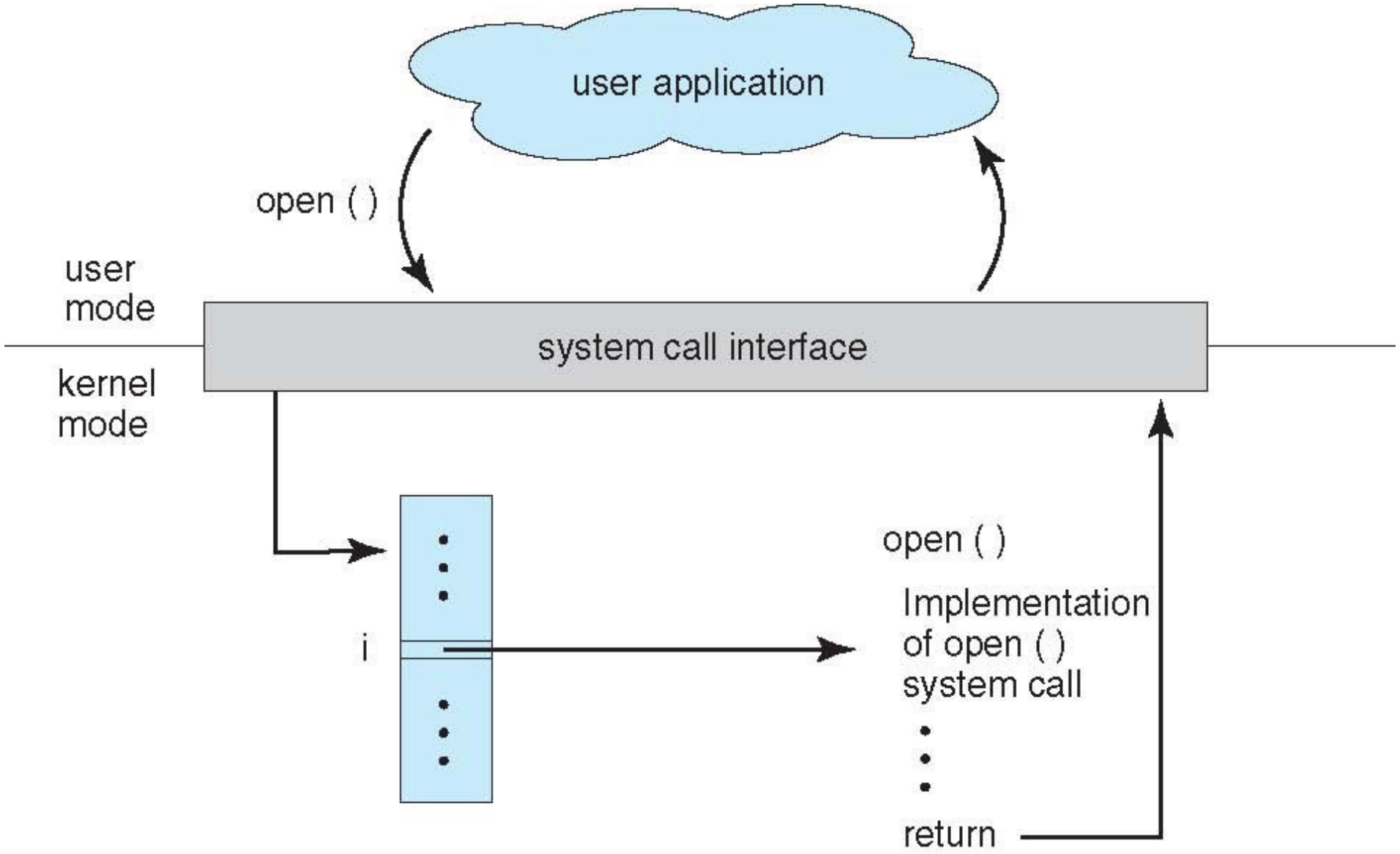
- Software-hardware interface
- OS kernel functions
  - Concepts ⟺ Managers (hardware)
  - Files ⟺ Files systems (drivers and devices)
  - Address space ⟺ Virtual memory (memory)
  - Programs ⟺ Process model (CPU, ISA)
- OS provides abstractions of devices and hardware objects
  - These abstractions are represented in software running in the OS and data structures that it maintains

Systems Calls

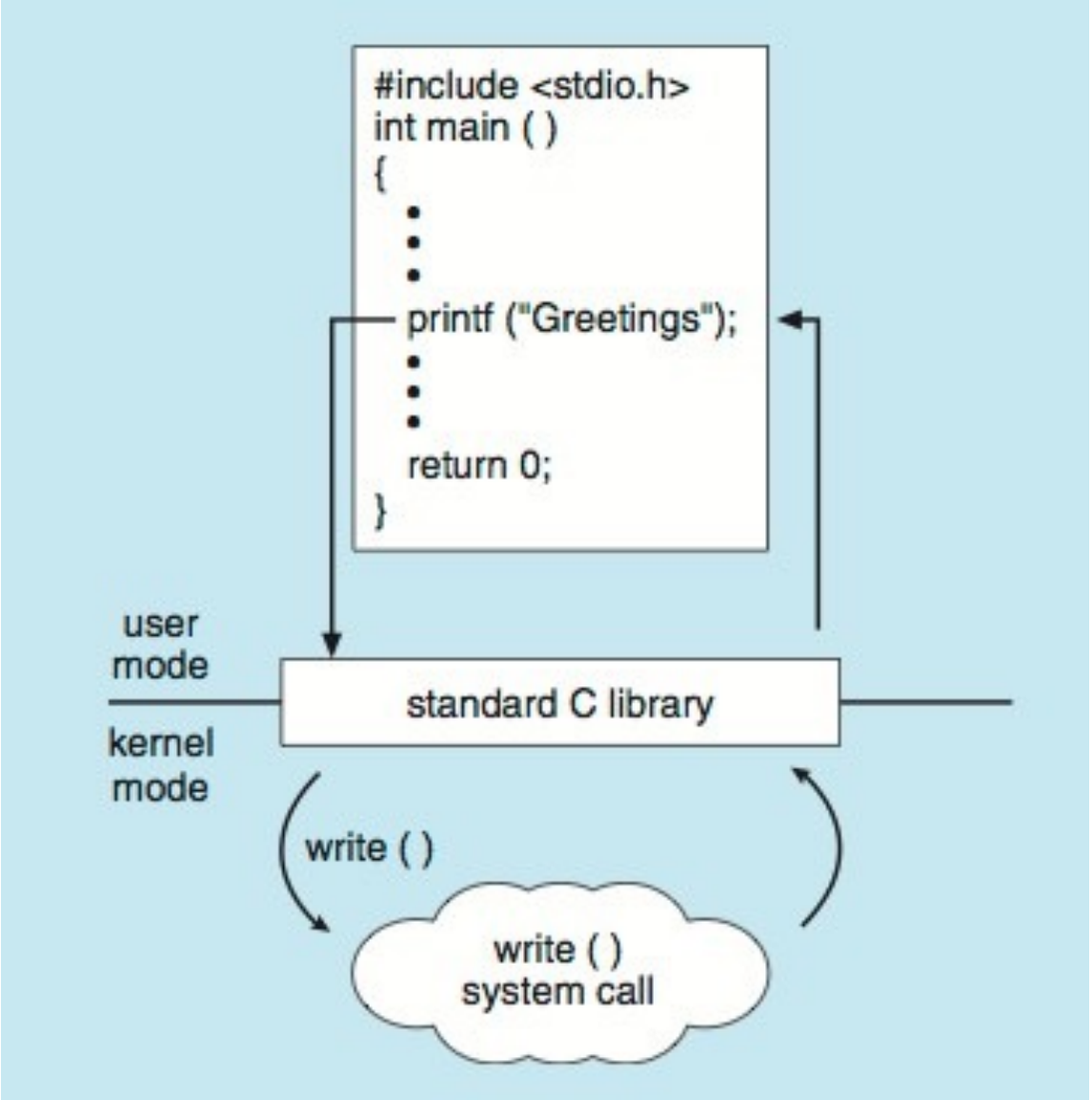
- Programming interface to OS services via system libraries
- Typically written in a high-level language (C, C++)
- Mostly accessed by programs via a high-level *application programming interface* (API) (versus direct system call use)
  - Win32 (Windows), POSIX (Unix, Linux, MacOS), Java (JVM)
- Typically, a number is associated with each system call
  - System-call interface maintains a table indexed by call #
- System call interface invokes the intended system call in OS kernel and returns status of the system call and return values
- Caller just obeys API and understand what OS will do
  - Most details of OS interface hidden by API

Note: names of systems calls in slides are generic

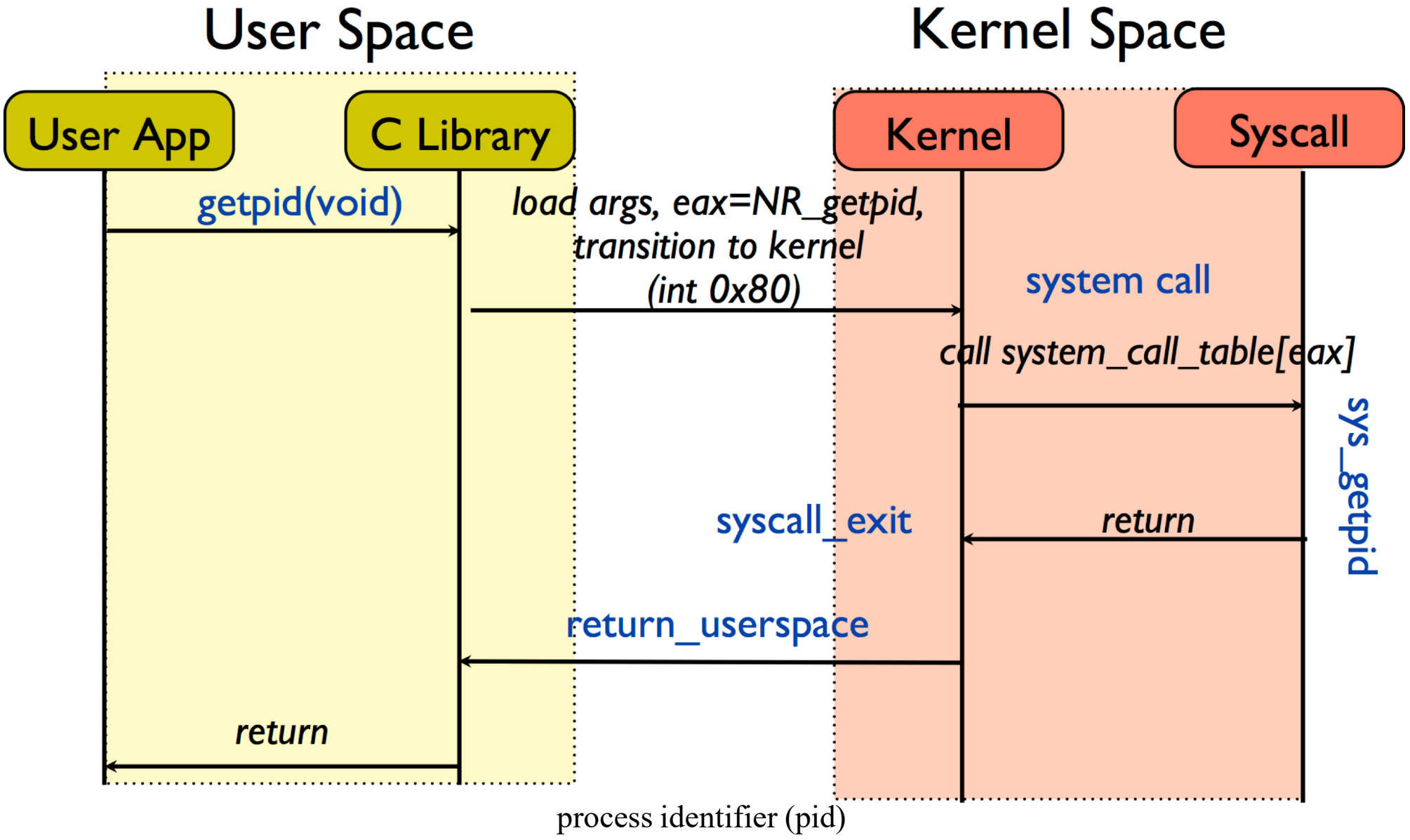
System Call – OS Relationship



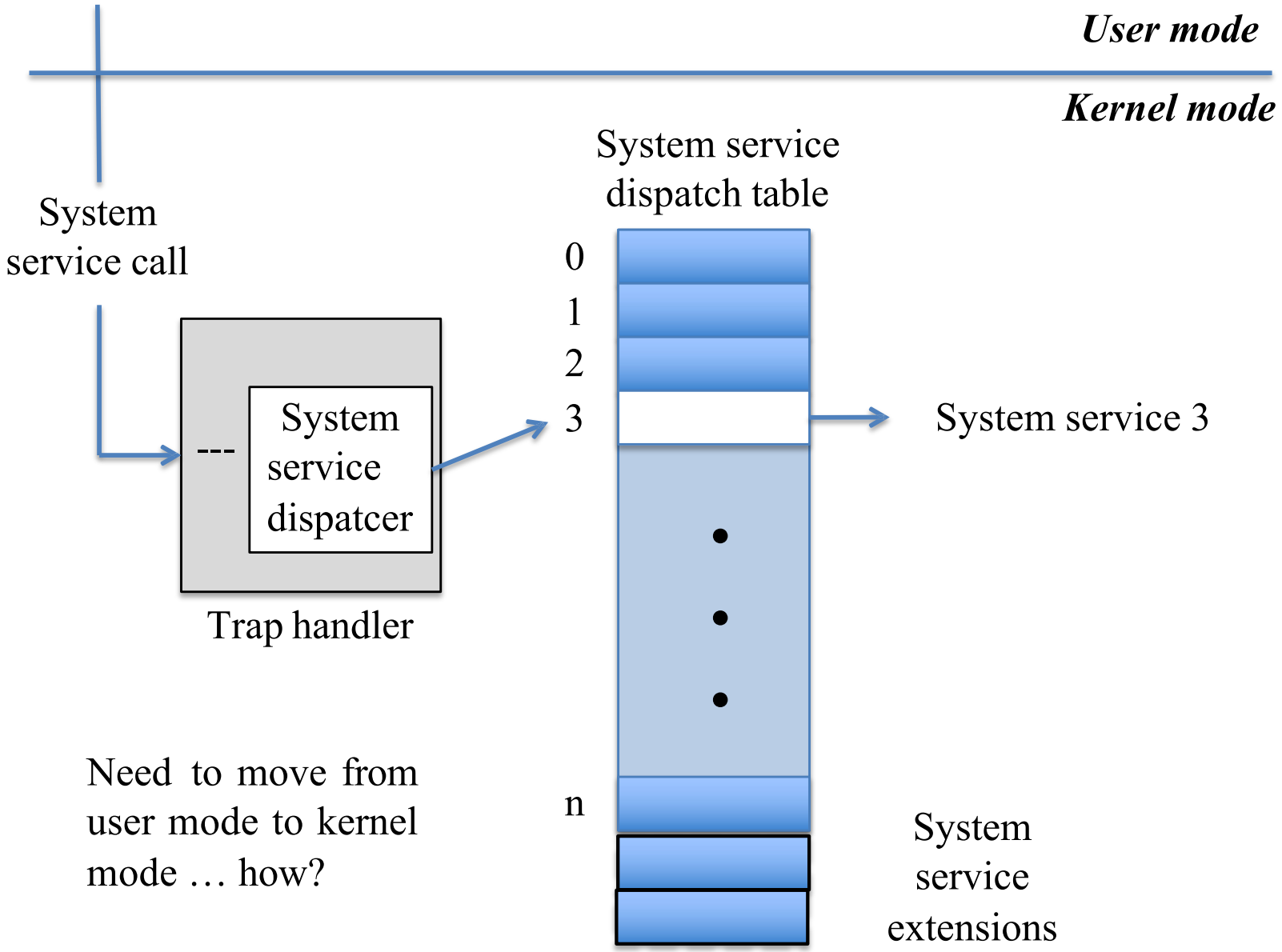
Standard C Library Example (printf())



System Call Example (getpid())



System Call Handling

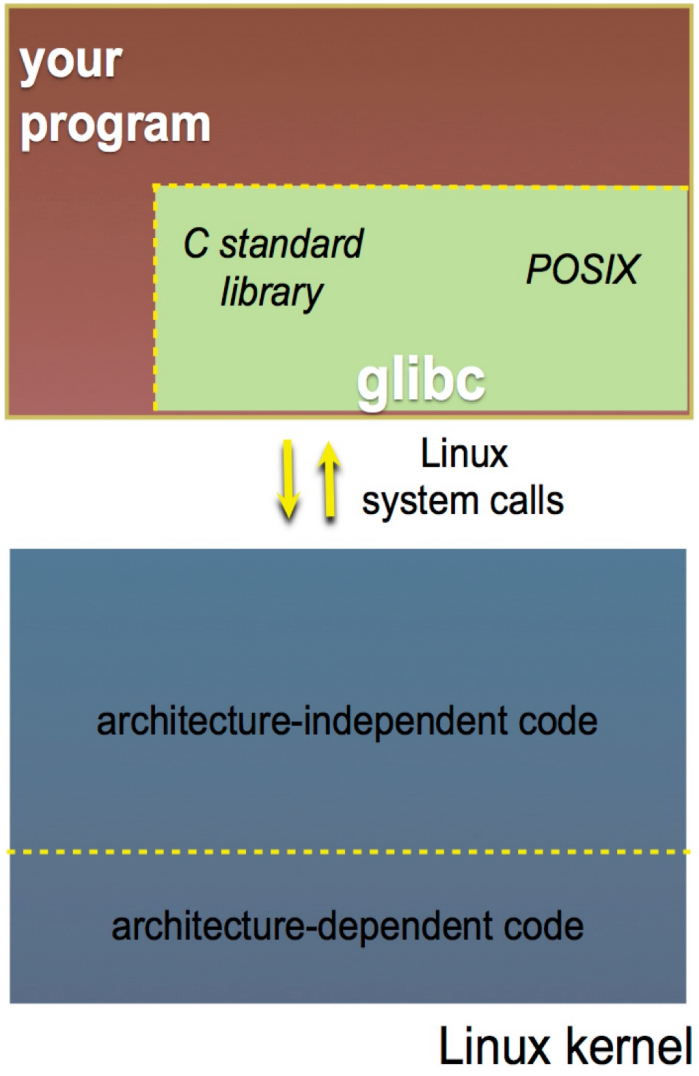


System Call Process

- Procedure call in user process
- Initial work in user mode — *libc*
- Trap instruction to invoke kernel — *int 0x80*
- Invoke system calls — *sys_read, mmap2*
- Do I/O — *read from disk*
- Wait — *disk is slow*
- Completion interrupt handling
- Return-from-interrupt instruction
- Final work in library — *libc*
- Return to user code

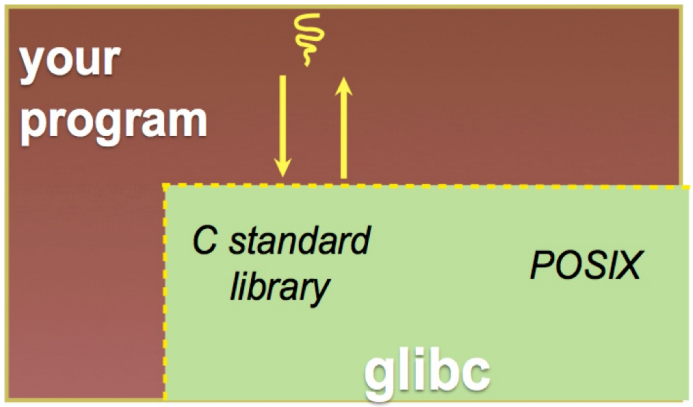
Details on x86 / Linux

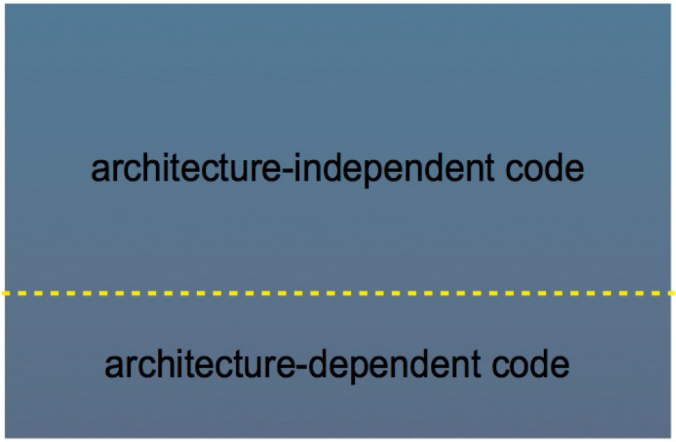
- A more accurate picture:
  - Consider a typical Linux process
  - Its course of execution can be in several places
    - in your program's code
    - in C standard library
    - in the kernel model (system calls)



Details on x86 / Linux

- Some routines your program invokes may be entirely handled by glibc
  - Without involving the kernel
    - for example, *strcmp( )* from stdio.h
  - There is some initial overhead when invoking functions in dynamically linked libraries …
  - … but after symbols are resolved, invoking glibc routines is nearly as fast as a function call within your program itself

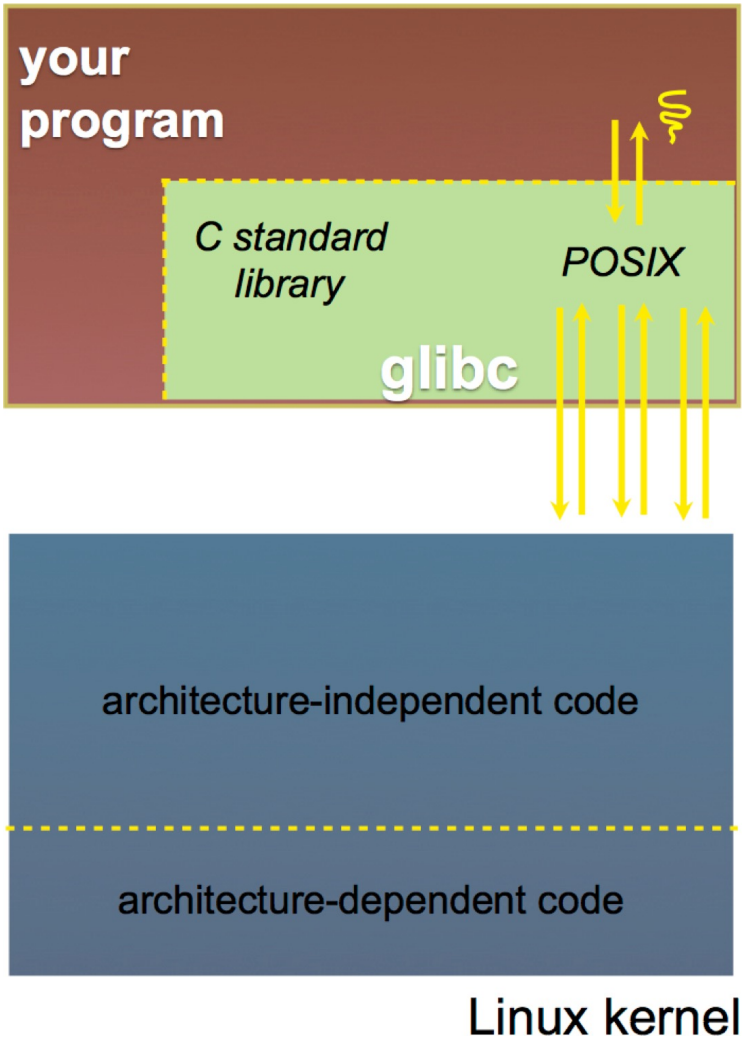




Linux kernel

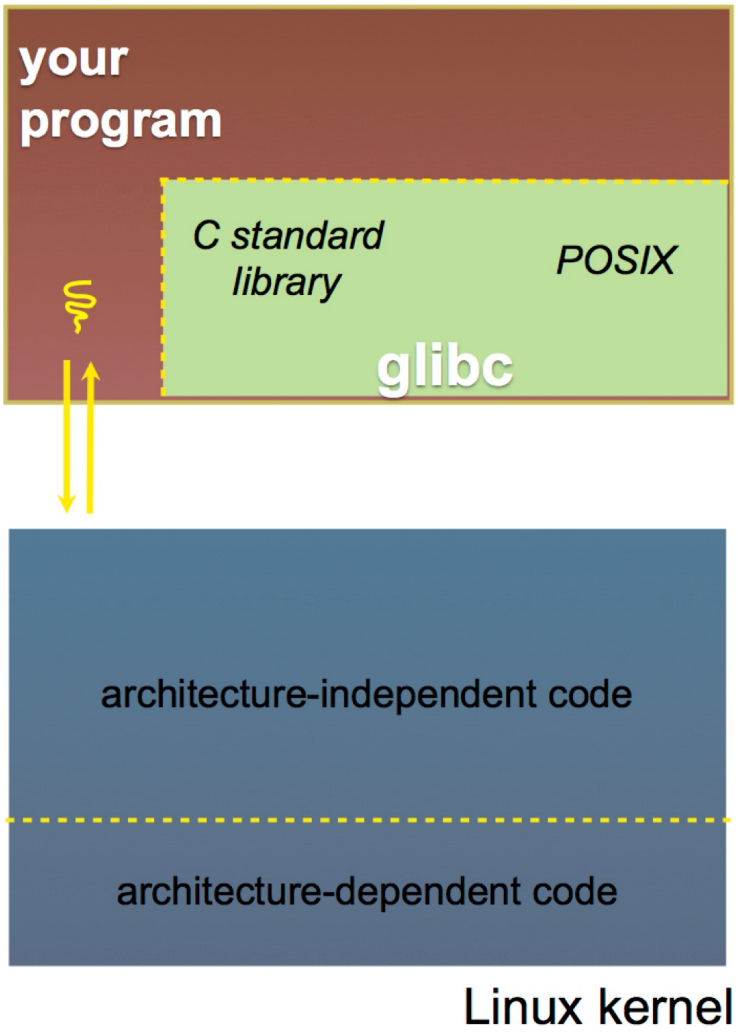
Details on x86 / Linux

- Some routines may be handled by glibc, but they in turn invoke Linux system calls
  - Example: POSIX wrappers around Linux syscalls
    - POSIX *readdir( )* invokes the underlying Linux *readdir( )*
  - Example: C *stdio* functions that read and write from files
    - *fopen( ), fclose( ), fprintf( ), …* invoke underlying Linux *open( ), read( ), write( ), close( ), …*



Details on x86 / Linux

- Your program can choose to directly invoke Linux system calls as well
  - Nothing forces you to link with glibc and use it
  - But relying on directly invoked Linux system calls may make your program less portable across UNIX varieties



Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

System Call Wrappers

- Typically, a program does NOT directly invoke system calls
- Instead, system call wrappers are used most of the time
  - Convenience
  - Portability
  - Error handling
  - Abstraction

Most of the time, when we say 'system calls', we are actually referring to the system call wrappers.

```
asm volatile (
        "syscall"
        : "=a" (ret)
        : "a"(n), "D"(a1), "S"(a2), "d"(a3)
        : "rcx", "r11", "memory"
    );
```

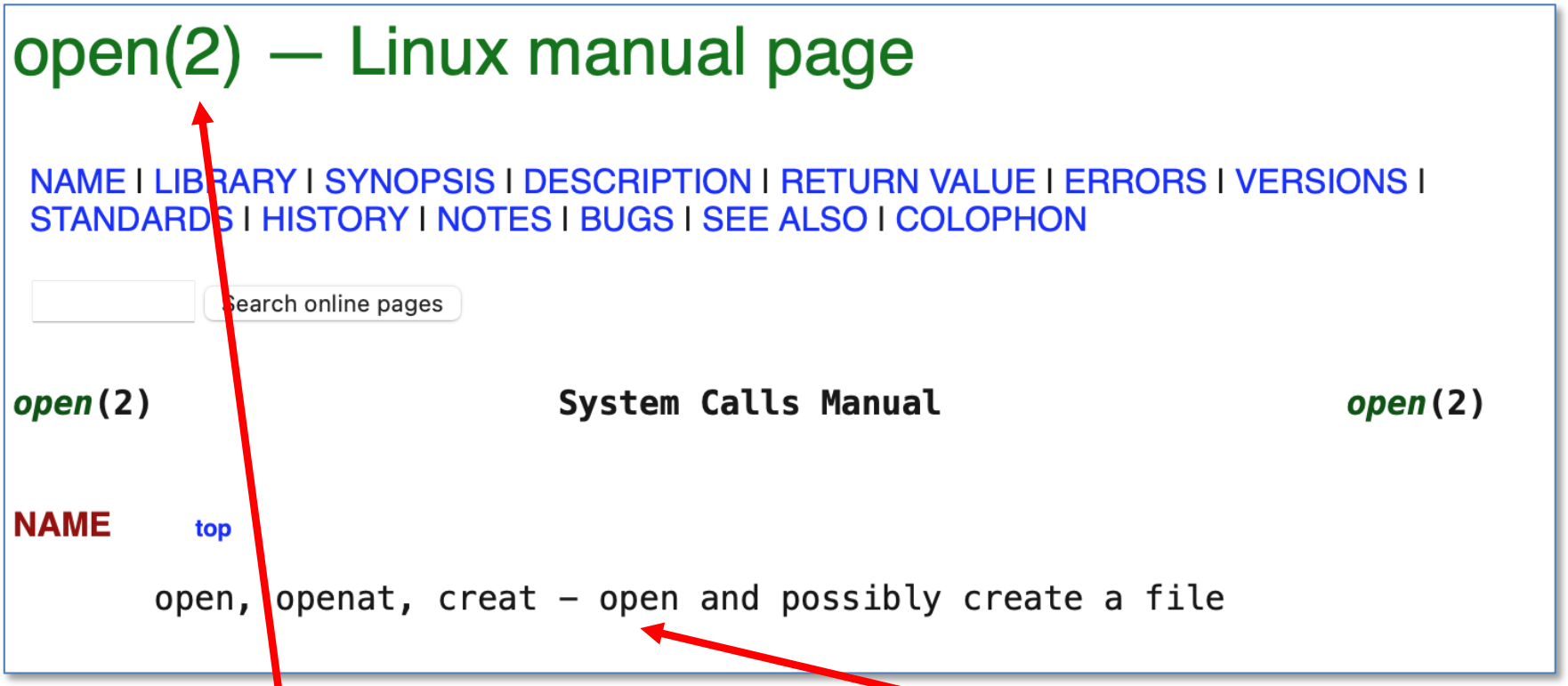
System call instruction

```
int fd = open("file.txt", O_RDONLY);
```

System call wrapper

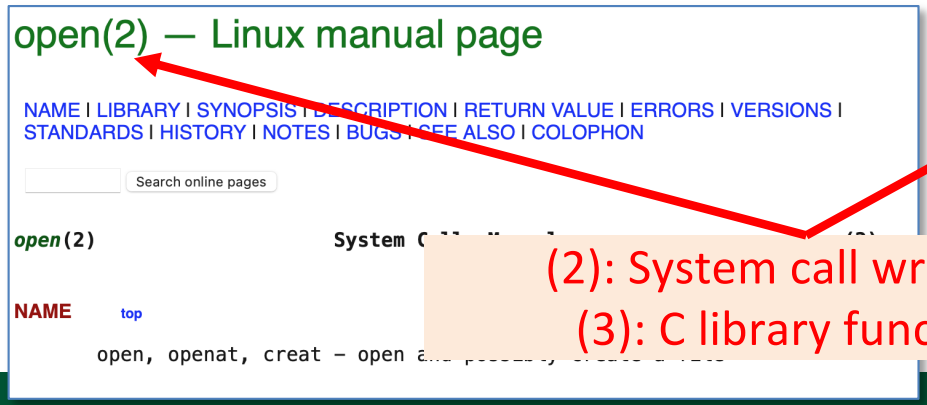
System Call Wrappers

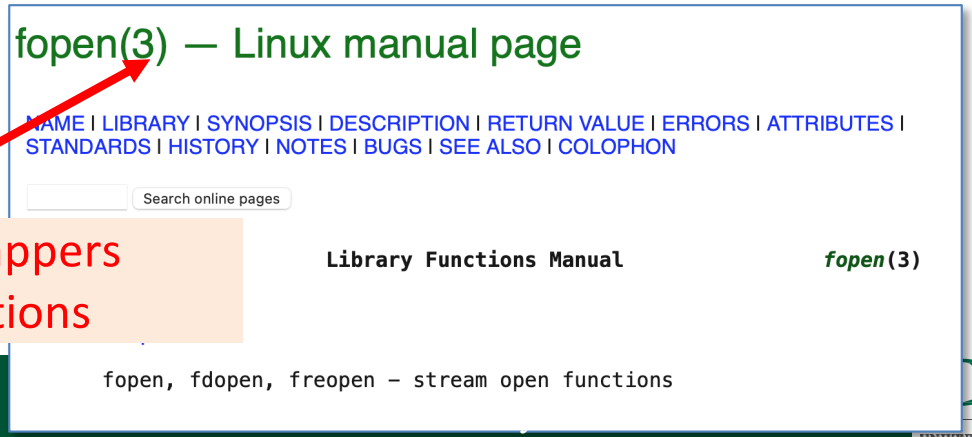
https://man7.org/linux/man-pages/man2/open.2.html



System Call Wrappers vs. C Library Functions

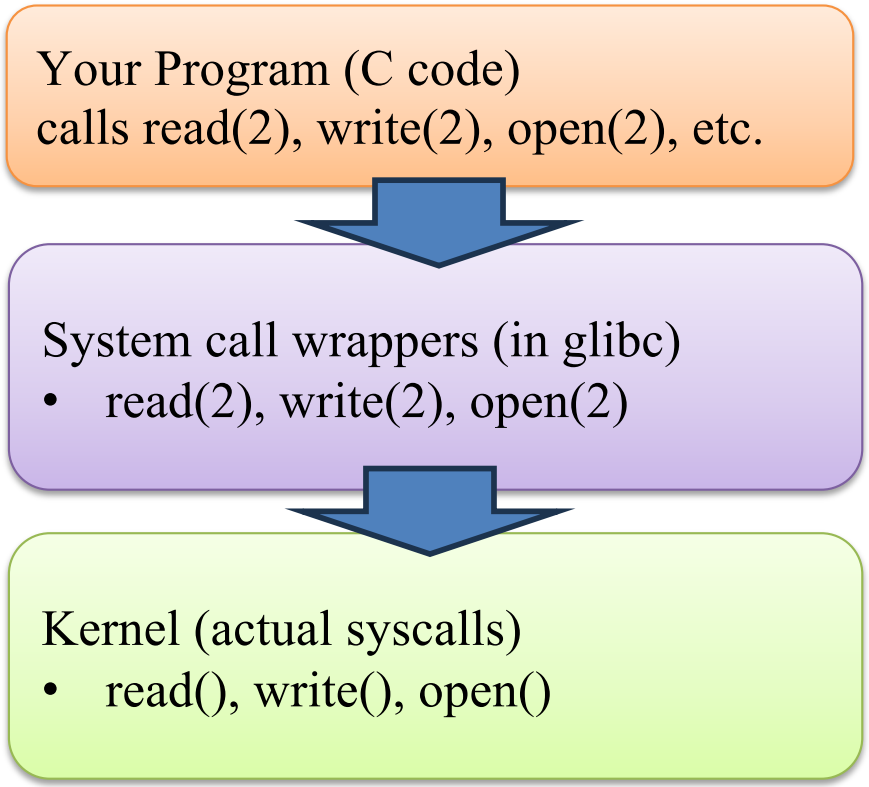
- Both are in C standard library: glibc
- System call wrapper
  - Thin C function that corresponds almost 1:1 with a kernel system call.
  - Bridge user space and kernel space (for a program to request OS services)
- C Library Functions
  - Higher-level functions
  - Could be purely user-space (string manipulations)
  - Could call system call wrappers to request OS services (open files)





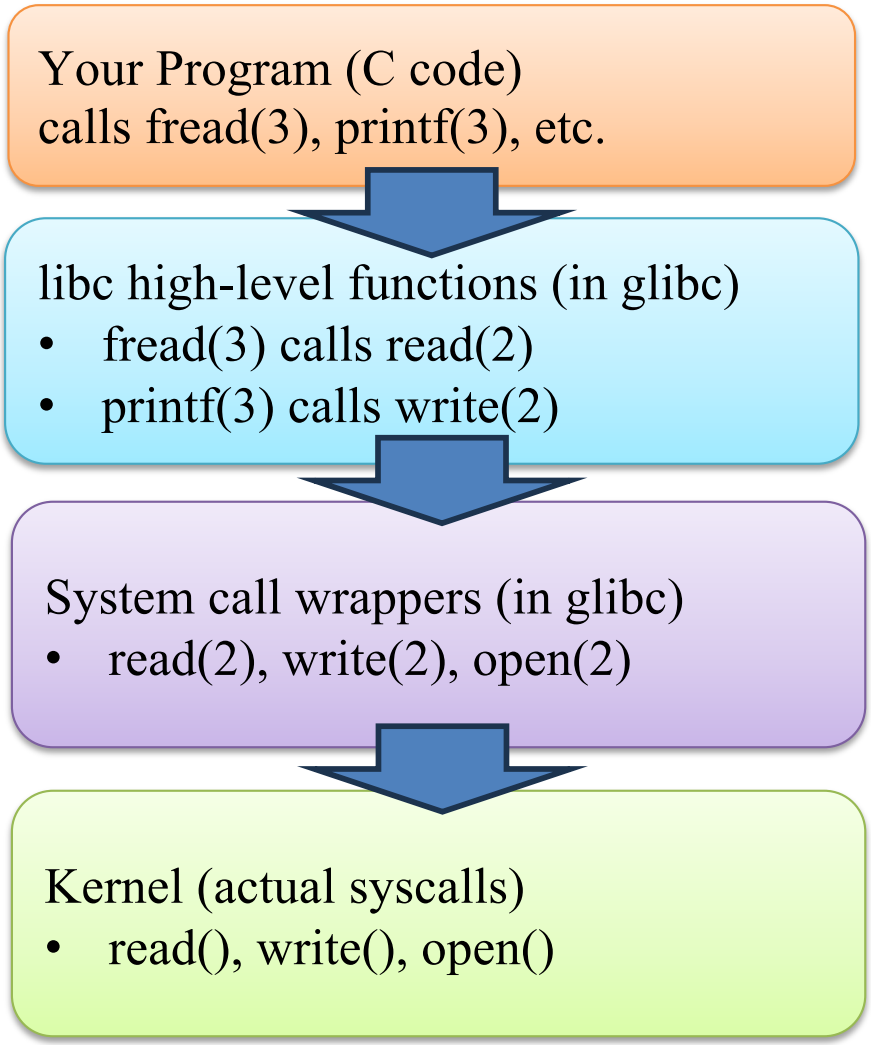
(2): System call wrappers
(3): C library functions

System Call Wrappers vs. C Library Functions



Directly call system call wrapper

Both are valid ways to make system calls



C library functions call system call wrapper

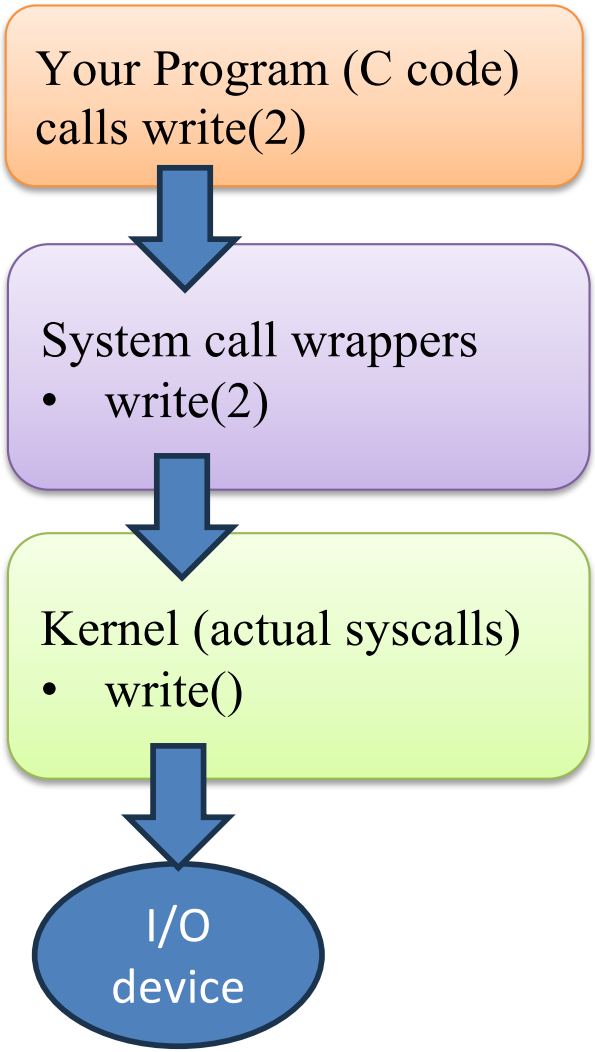
System Call Wrappers vs. C Library Functions

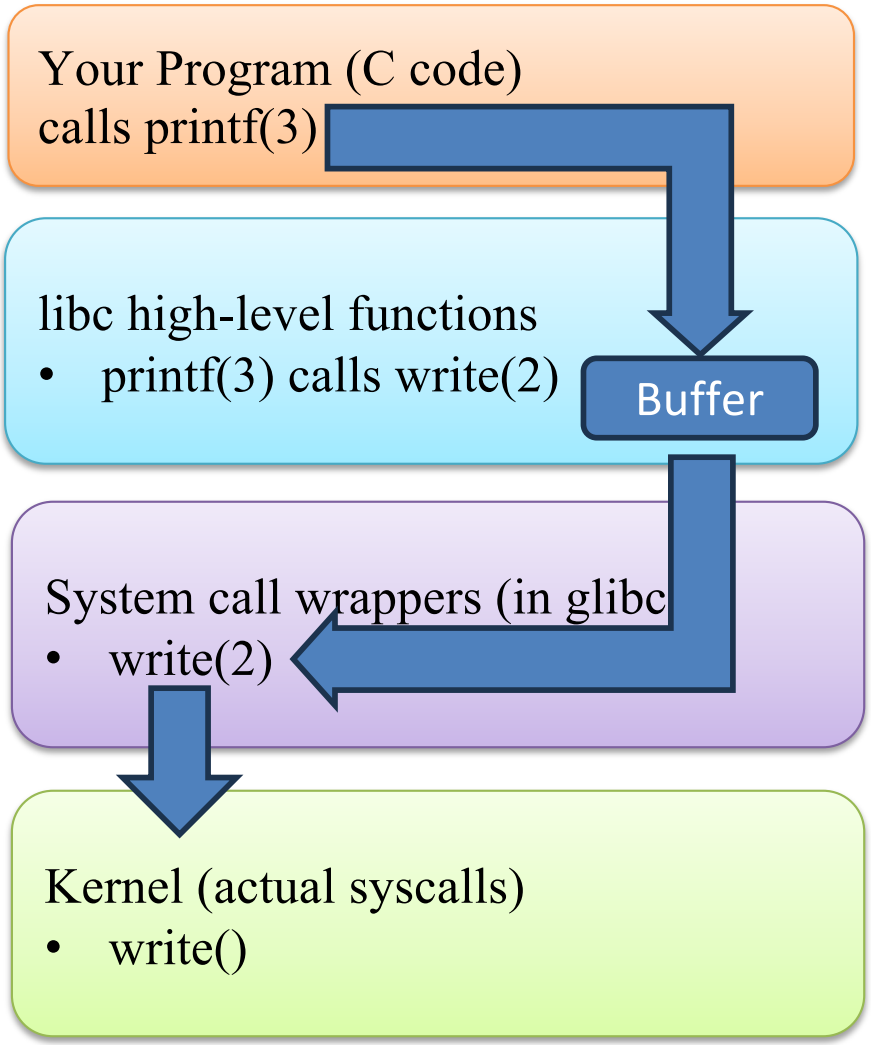
Aspect	System Call Wrapper	C Library Function
Layer	Low-level (full OS service functions)	High-level (less OS service functions)
Location	glibc	glibc
Implementation	Directly maps to a kernel syscall without buffering	May use multiple syscalls internally and may use internal buffers
Performance	Lower latency (immediate response)	More efficient for frequent small I/O operations
Portability	Less portable (depends on OS syscall API)	More portable (follows C/POSIX standards)
Error Handling	Returns -1 and sets errno	Also sets errno, may do extra handling
Example Functions	read(), write(), open(), close(), fork(), execve()	fread(), fwrite(), fopen(), fprintf(), printf(), system(), malloc()

System Call Wrappers vs. C Library Functions

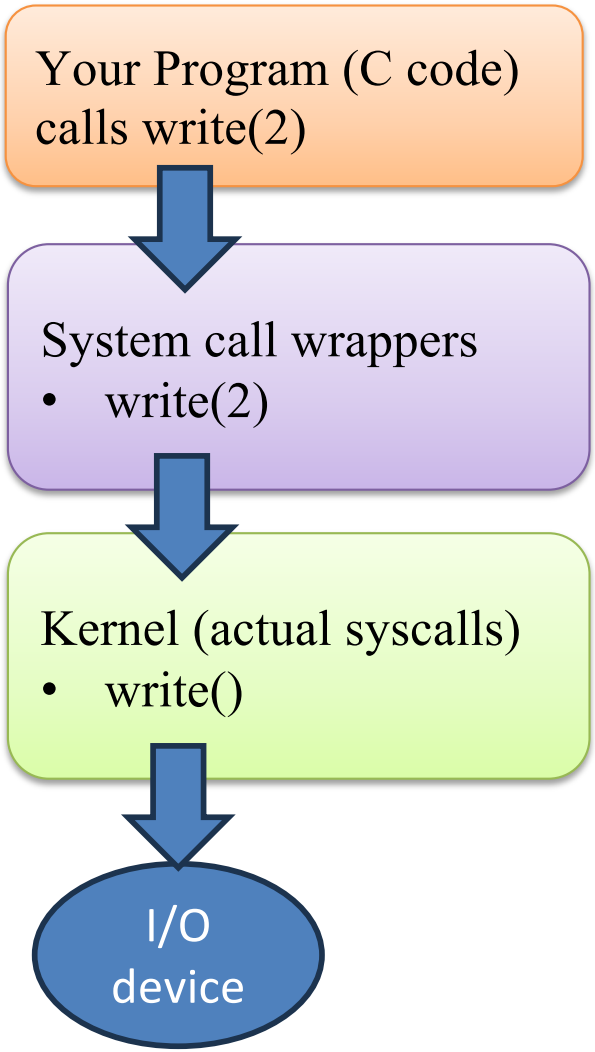
Aspect	System Call Wrapper	C Library Function
Layer	Low-level (full OS service functions)	High-level (less OS service functions)
Location	glibc	glibc
Implementation	Directly maps to a kernel syscall without buffering	May use multiple syscalls internally and may use internal buffers
Performance	Lower latency (immediate response)	More efficient for frequent small I/O operations
Portability	Less portable (depends on OS syscall API)	More portable (follows C/POSIX standards)
Error Handling	Returns -1 and sets errno	Also sets errno, may do extra handling
Example Functions	read(), write(), open(), close(), fork(), execve()	fread(), fwrite(), fopen(), fprintf(), printf(), system(), malloc()

System Call Wrappers vs. C Library Functions





System Call Wrappers vs. C Library Functions



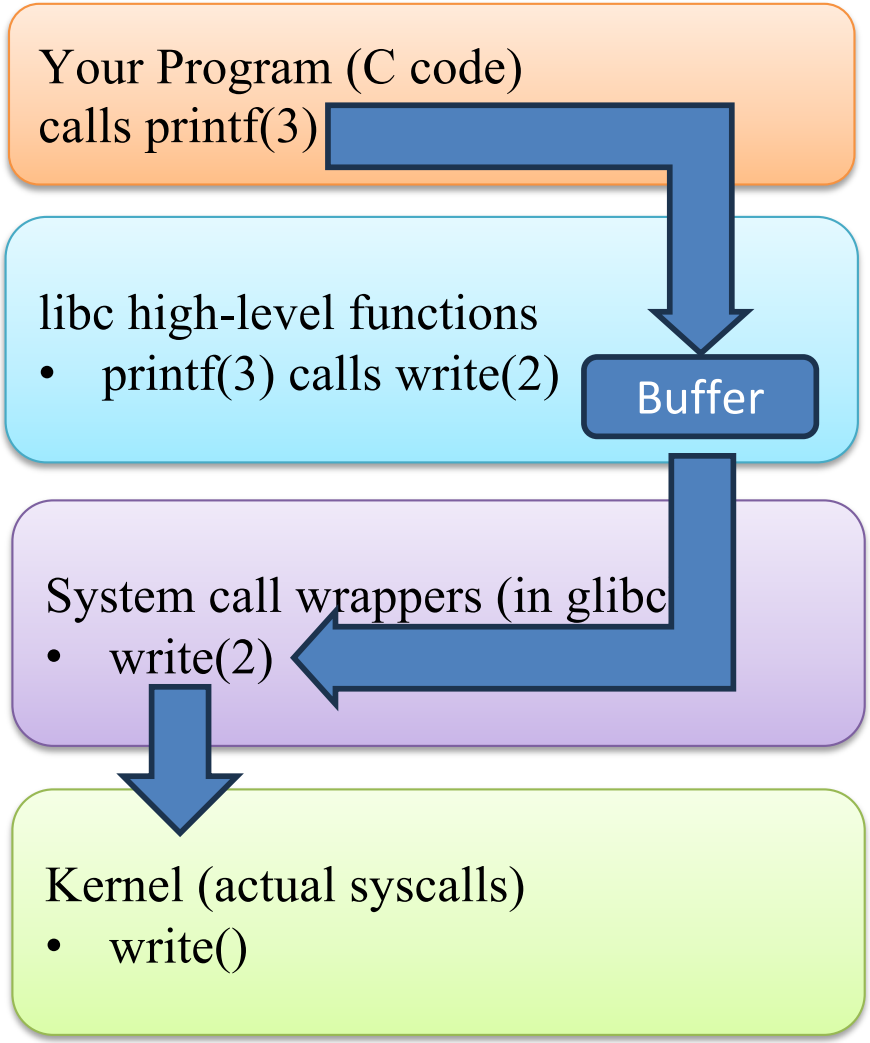
System call wrapper

- Making immediate system calls
- Read/write are immediately done
- ✓Minimize latency (e.g., log, progress bar, interactive app)
- ✗ If you do many small I/O, it leads to many system calls (high function call overhead)

```c
#include <unistd.h>
#include <stdio.h>
int main() {
      printf("Hello ");
      write(STDOUT_FILENO, "World\n", 6);
      return 0;
}
```

What would be the output?

System Call Wrappers vs. C Library Functions



C library function

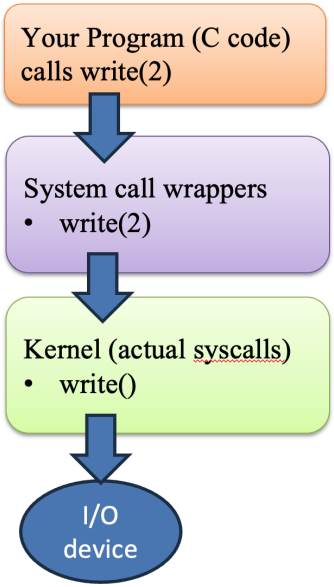
- Maintain an internal buffer
- Read/write is first done in buffer
- Write out to I/O when buffer is full or we explicitly let it flush
- Read is faster if data is in buffer
- ✓ Minimize the number of system calls
- ✗ May have higher latency (read/write are NOT immediately done

New line or *fflush(stdout)* *can flush stdout buffer*

```c
#include <unistd.h>
#include <stdio.h>
int main() {
      printf("Hello \n");
      write(STDOUT_FILENO, "World\n", 6);
      return 0;
}
```

What would be the output?

System Call Wrappers vs. C Library Functions

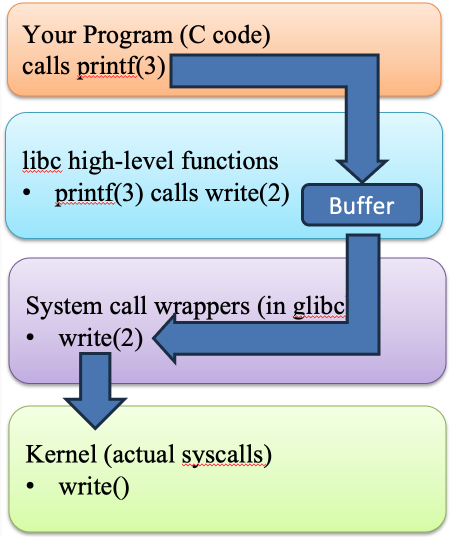



```c
// unbuffered_file_write_timed.c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

int main() {
    int fd = open("unbuffered.txt",
O_WRONLY | O_CREAT | O_TRUNC,
0644);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    const char c = 'A';
    for (int i = 0; i < 100000; i++) {
        if (write(fd, &c, 1) != 1) {
            perror("write");
            return 1;
        }
    }

    close(fd);
return 0;
}
```

Directly call system call wrapper

```c
// buffered_file_write_timed.c
#include <stdio.h>
#include <time.h>

int main() {
    FILE *fp = fopen("buffered.txt", "w");
    if (!fp) {
        perror("fopen");
        return 1;
    }

    const char c = 'A';
    for (int i = 0; i < 100000; i++) {
        if (fwrite(&c, 1, 1, fp) != 1) {
            perror("fwrite");
            fclose(fp);
            return 1;
        }
    }

    fclose(fp);

return 0;
}
```

C library functions call system call wrapper



How fast can they perform 100,000 small write (1 byte) operations?

System Call Wrappers vs. C Library Functions

```c
// unbuffered_file_write_timed.c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

int main() {
    int fd = open("unbuffered.txt",
O_WRONLY | O_CREAT | O_TRUNC,
0644);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    const char c = 'A';
    for (int i = 0; i < 100000; i++) {
        if (write(fd, &c, 1) != 1) {
            perror("write");
            return 1;
        }
    }

    close(fd);
return 0;
}
```

Directly call system call wrapper

- System call wrappers are more complex more functionalities
- C library functions are simpler to use, less functionalities

- fd and fp are different types
- Both are for referring to a file
- System call wrapper use file descriptor,
  - int type, managed by the kernel
- C library function uses file stream
  - FILE type, managed by C library
- Can covert between each other
  - fdopen ()
  - fileno

```c
// buffered_file_write_timed.c
#include <stdio.h>
#include <time.h>

int main() {
    FILE *fp = fopen("buffered.txt", "w");
    if (fp == 0) {
        perror("fopen");
        return 1;
    }

    const char c = 'A';
    for (int i = 0; i < 100000; i++) {
        if (fwrite(&c, 1, 1, fp) != 1) {
            perror("fwrite");
            fclose(fp);
            return 1;
        }
    }

    fclose(fp);

return 0;
}
```

C library functions call system call wrapper

System Call Wrappers vs. C Library Functions

Aspect	System Call Wrapper	C Library Function
Layer	Low-level (full OS service functions)	High-level (less OS service functions)
Location	glibc	glibc
Implementation	Directly maps to a kernel syscall without buffering	May use multiple syscalls internally and may use internal buffers
Performance	Lower latency (immediate response)	More efficient for frequent small I/O operations
Portability	Less portable (depends on OS syscall API)	More portable (follows C/POSIX standards)
Error Handling	Returns -1 and sets errno	Also sets errno, may do extra handling
Example Functions	read(), write(), open(), close(), fork(), execve()	fread(), fwrite(), fopen(), fprintf(), printf(), system(), malloc()

System Call Wrappers vs. C Library Functions

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>


int main() {
    int fd = open("example.txt", O_WRONLY |
O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        return 1;
    }


    const char *text = "Hello, world!\n";
    if (write(fd, text, 14) != 14) {
        perror("write");
    }


    close(fd);
    return 0;
}
```

✅ Works fine on **Linux**, **macOS**, other **POSIX systems**.

❌ **Will fail on Windows**: Windows doesn't use open/write directly

Windows has CreateFile and other Win32 APIs instead.

```c
#include <stdio.h>


int main() {
    FILE *fp = fopen("example.txt", "w");
    if (!fp) {
        perror("fopen");
        return 1;
    }


    fprintf(fp, "Hello, world!\n");
    fclose(fp);
    return 0;
}
```

✅ Works on **Linux, macOS, Windows**. The C standard library implements fopen/fwrite for each OS internally. You don't have to worry about the underlying system calls or permissions differences.

File Interface

- Goal:
  - Provide a uniform abstraction for accessing the OS and its resources
- Abstraction:
  - File
- Use file system calls to access OS services
  - Devices, sockets, pipes, and so on
  - Also us in OS in general

I/O with System Calls

- Much I/O is based on a streaming model
  - Sequence of bytes
- *write()* sends a stream of bytes somewhere
- *read()* blocks until a stream of input is ready
- Annoying details:
  - Might fail, can block for a while
  - Working with file descriptors
  - Arguments are pointers to character buffers
  - See the *read()* and *write()* man pages

File Descriptors

- A process might have several different I/O streams in use at any given time
- These are specified by a kernel data structure called a *file descriptor*
  - Each process has its own table of file descriptors
- *open()* associates a file descriptor with a file
- *close()* destroys a file descriptor
- Standard input and standard output are usually associated with a terminal
  - More on that later

File Descriptors

```
// header files omitted
int main() {
    const char *msg_stdout = "This goes to stdout (terminal)\n";
    const char *msg_stderr = "This is an error message (stderr)\n";
    write(STDOUT_FILENO, msg_stdout, 33);
    write(STDERR_FILENO, msg_stderr, 34);

    int log_fd = open("log.txt", O_WRONLY | O_CREAT |
O_TRUNC, 0644);
    if (log_fd == -1) {
        perror("open log.txt");
        return 1;
    }
    const char *log_msg = "This is a log message\n";
    write(log_fd, log_msg, 22);

    int input_fd = open("input.txt", O_RDONLY);
    if (input_fd == -1) {
        perror("open input.txt");
        close(log_fd);
        return 1;
    }

    char buffer[100];
    ssize_t n = read(input_fd, buffer, sizeof(buffer) - 1);
    write(1, buffer, n);  // write to stdout

    close(log_fd);
    close(input_fd);
}
```

STDOUT_FILENO=1: standard output
STDERR_FILENO=2: standard error

Open a file for writing logs

Open a file for reading input

Regular File

- File has a *pathname*: */tmp/foo*
- Can open the file
  - *int fd = open("/tmp/foo", O_RDWR)*
  - For reading and writing
- Can read from and write to the file
  - *bytes = read(fd, buf, max); /* buf get output */*
  - *bytes = write(fd, buf, len); /* buf has input */*

flags for read/write access

pointer to buffer

Socket File

- File has a pathname: */tmp/bar*
  - Files provide a persistence for a communication channel
  - Usually used for local communication (UNIX domain sockets)
- Open, read, and write via socket operations
  - *sockfd = socket(AF_INET, SOCK_STREAM, 0);*
  - *local.path* is set to */tmp/bar*
  - *bind (sockfd, &local, len)*
  - Use sock operations to read and write

Socket File

```
// header files omitted
int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
// create socket
    if (sockfd < 0) {
        perror("socket");
        return 1;
    }

    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080); // server port
    if (inet_pton(AF_INET, "127.0.0.1",
&server_addr.sin_addr) <= 0) {
        perror("inet_pton");
        close(sockfd);
        return 1;
    }

    // connect to server
    if (connect(sockfd, (struct sockaddr*)&server_addr,
sizeof(server_addr)) < 0) {
        perror("connect");
        close(sockfd);
        return 1;
    }
```

File descriptor for network socket

```
 // send data using file descriptor
    const char *msg = "Hello server\n";
    write(sockfd, msg, strlen(msg));

    // receive response
    char buffer[1024];
    ssize_t n = read(sockfd, buffer, sizeof(buffer)-1);
    if (n > 0) {
        buffer[n] = '\0';
        printf("Received: %s", buffer);
    }

    // close socket
    close(sockfd);
    return 0;
}
```

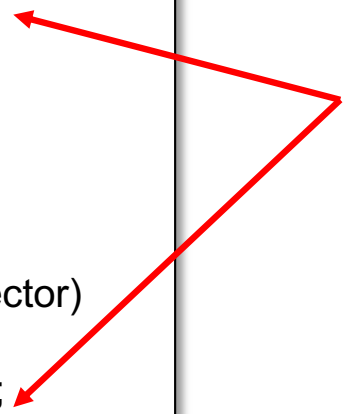
Use write/read to send/receive data over networks

Device File

- Files for interacting with physical devices
  - */dev/null* (no device)
  - */dev/cdrom* (CD-drive)
- Use file system operations, but are handled in device-specific ways
  - *open()*, *read()*, *write()* correspond to device-specific functions (act as function pointers!)
  - Also, use *ioctl* (I/O control) to interact (later)

Device File

```
// header files omitted
int main() {
    // Open CD-ROM device
    int fd = open("/dev/cdrom", O_RDONLY);
    if (fd < 0) {
        perror("open /dev/cdrom");
        return 1;
    }

    // Read first 2048 bytes (one CD-ROM sector)
    unsigned char buffer[2048];
    ssize_t n = read(fd, buffer, sizeof(buffer));
    if (n < 0) {
        perror("read");
        close(fd);
        return 1;
    }

    printf("Read %zd bytes from CD-ROM\n", n);

    // Close the device
    close(fd);
    return 0;
}
```



Reading from the CD-ROM

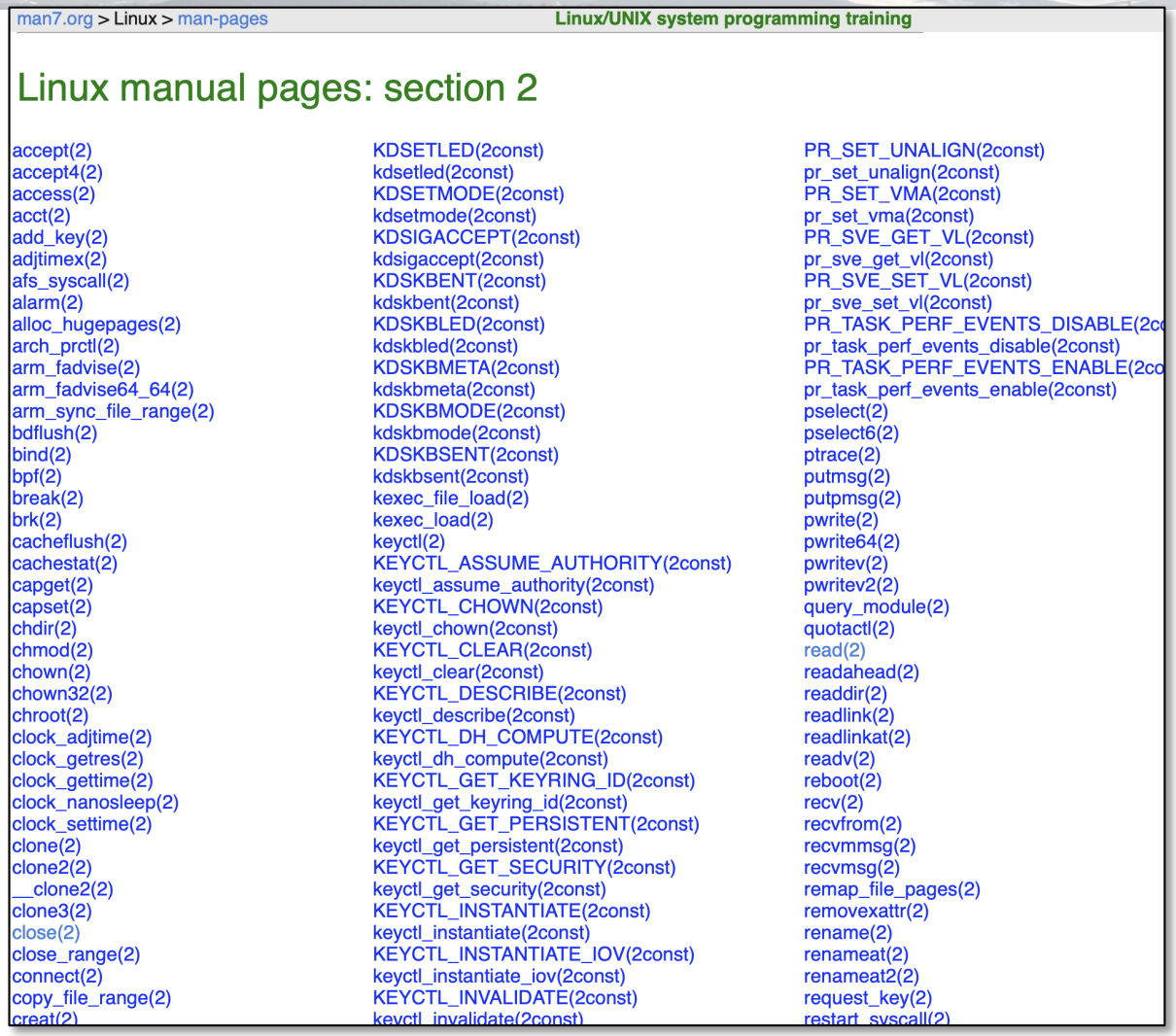
Other System Calls

- It's possible to hook the output of one program into the input of another
  - *pipe()*
- It's possible to block until one of several file descriptor streams is ready
  - *select()*
- Special calls for dealing with network
  - AF_INET sockets, and so on
- Send a message to other (or all) processes
  - *signal()*
- Most of these in section 2 of manual

Syscall Functionality

- System calls are the main interface between processes and the OS
  - Like an extended "instruction set" for user programs that hide many details
  - First Unix system had a couple dozen system calls
  - Current systems have many more
    - >300 in Linux and >500 in FreeBSD
  - Understanding the system call interface of a given OS lets you write useful programs under it
- Natural questions to ask:
  - Is this the right interface? how to evaluate?
  - How can these system calls be implemented?

Syscall Functionality



man7.org > Linux > man-pages — Linux/UNIX system programming training

Linux manual pages: section 2

accept(2)
accept4(2)
access(2)
acct(2)
add_key(2)
adjtimex(2)
afs_syscall(2)
alarm(2)
alloc_hugepages(2)
arch_prctl(2)
arm_fadvise(2)
arm_fadvise64_64(2)
arm_sync_file_range(2)
bdflush(2)
bind(2)
bpf(2)
break(2)
brk(2)
cacheflush(2)
cachestat(2)
capget(2)
capset(2)
chdir(2)
chmod(2)
chown(2)
chown32(2)
chroot(2)
clock_adjtime(2)
clock_getres(2)
clock_gettime(2)
clock_nanosleep(2)
clock_settime(2)
clone(2)
clone2(2)
__clone2(2)
clone3(2)
close(2)
close_range(2)
connect(2)
copy_file_range(2)
creat(2)

KDSETLED(2const)
kdsetled(2const)
KDSETMODE(2const)
kdsetmode(2const)
KDSIGACCEPT(2const)
kdsigaccept(2const)
KDSKBENT(2const)
kdskbent(2const)
KDSKBLED(2const)
kdskbled(2const)
KDSKBMETA(2const)
kdskbmeta(2const)
KDSKBMODE(2const)
kdskbmode(2const)
KDSKBSENT(2const)
kdskbsent(2const)
kexec_file_load(2)
kexec_load(2)
keyctl(2)
KEYCTL_ASSUME_AUTHORITY(2const)
keyctl_assume_authority(2const)
KEYCTL_CHOWN(2const)
keyctl_chown(2const)
KEYCTL_CLEAR(2const)
keyctl_clear(2const)
KEYCTL_DESCRIBE(2const)
keyctl_describe(2const)
KEYCTL_DH_COMPUTE(2const)
keyctl_dh_compute(2const)
KEYCTL_GET_KEYRING_ID(2const)
keyctl_get_keyring_id(2const)
KEYCTL_GET_PERSISTENT(2const)
keyctl_get_persistent(2const)
KEYCTL_GET_SECURITY(2const)
keyctl_get_security(2const)
KEYCTL_INSTANTIATE(2const)
keyctl_instantiate(2const)
KEYCTL_INSTANTIATE_IOV(2const)
keyctl_instantiate_iov(2const)
KEYCTL_INVALIDATE(2const)
keyctl_invalidate(2const)

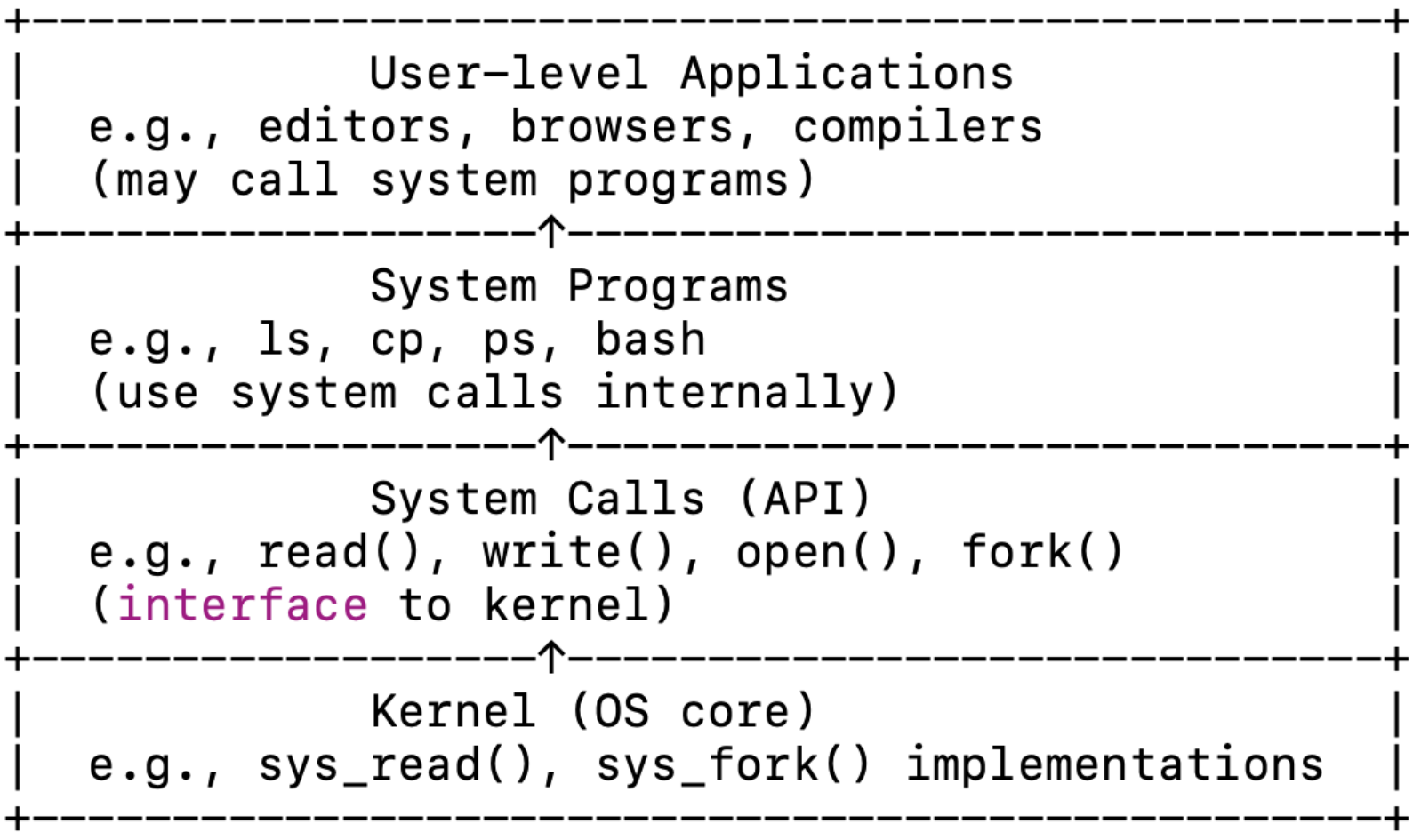
PR_SET_UNALIGN(2const)
pr_set_unalign(2const)
PR_SET_VMA(2const)
pr_set_vma(2const)
PR_SVE_GET_VL(2const)
pr_sve_get_vl(2const)
PR_SVE_SET_VL(2const)
pr_sve_set_vl(2const)
PR_TASK_PERF_EVENTS_DISABLE(2c
pr_task_perf_events_disable(2const)
PR_TASK_PERF_EVENTS_ENABLE(2c
pr_task_perf_events_enable(2const)
pselect(2)
pselect6(2)
ptrace(2)
putmsg(2)
putpmsg(2)
pwrite(2)
pwrite64(2)
pwritev(2)
pwritev2(2)
query_module(2)
quotactl(2)
read(2)
readahead(2)
readdir(2)
readlink(2)
readlinkat(2)
readv(2)
reboot(2)
recv(2)
recvfrom(2)
recvmmsg(2)
recvmsg(2)
remap_file_pages(2)
removexattr(2)
rename(2)
renameat(2)
renameat2(2)
request_key(2)
restart_syscall(2)

… and a lot more

System Programs

- System programs provide a convenient environment for program development and execution
- They can be divided into categories:
  - File manipulation
  - Status information
  - File modification
  - Programing language support (program development)
  - Program loading and execution
  - Application programs
- Most user's view of the operating system is defined by system programs, not the actual system calls

System Programs vs System Calls



* Generated by ChatGPT

Linux man pages online

The links from this page display HTML renderings of the man pages from the Linux *man-pages* project as well as a curated collection of pages from various other free software projects.

- List of all man pages: by section | alphabetically | by project
- Individual sections (page names only): 1, 2, 3, 4, 5, 6, 7, 8
- Intro pages: intro(1), intro(2), intro(3), intro(4), intro(5), intro(6), intro(7), intro(8)

Section	Contents	Example
1	System programs (User commands)	ls(1), grep(1)
2	System calls (functions provided by the kernel)	read(2), open(2)
3	Library functions (C standard library, other libraries)	printf(3), malloc(3)
4	Special files / devices	/dev/null(4), /dev/sda(4)
5	File formats and conventions	/etc/passwd(5), crontab(5)
6	Games and screensavers	nethack(6)
7	Miscellaneous / conventions / standards	signal(7), regex(7)
8	System administration commands (root/admin)	mount(8), iptables(8)

OS Design and Implementation

- Design and implementation of OS is not "solvable" in full, but some approaches have proven successful
- Internal structure of different operating systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- There are *user* goals and *system* goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

OS Policy versus Mechanism

- Important principle to separate
  Policy: What will (should) be done?
  Mechanism: How to do it?
  - Mechanisms determine how to do something
  - Policies decide what will be done
- Separation of policy from mechanism is a **VERY** important principle
  - Allows maximum flexibility if policy decisions are to be changed later
  - Universal principle
- Specifying and designing an OS is a highly creative task of software engineering

Summary

- Operating systems must balance many needs
  - Impression that each process has individual use of system
  - Comprehensive management of system resources
- Operating system structures try to make use of system resources straightforward
  - Libraries
  - System services
  - System calls and other interfaces

Next Class

- Processes