Chapter 2

Operating-System Structures

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

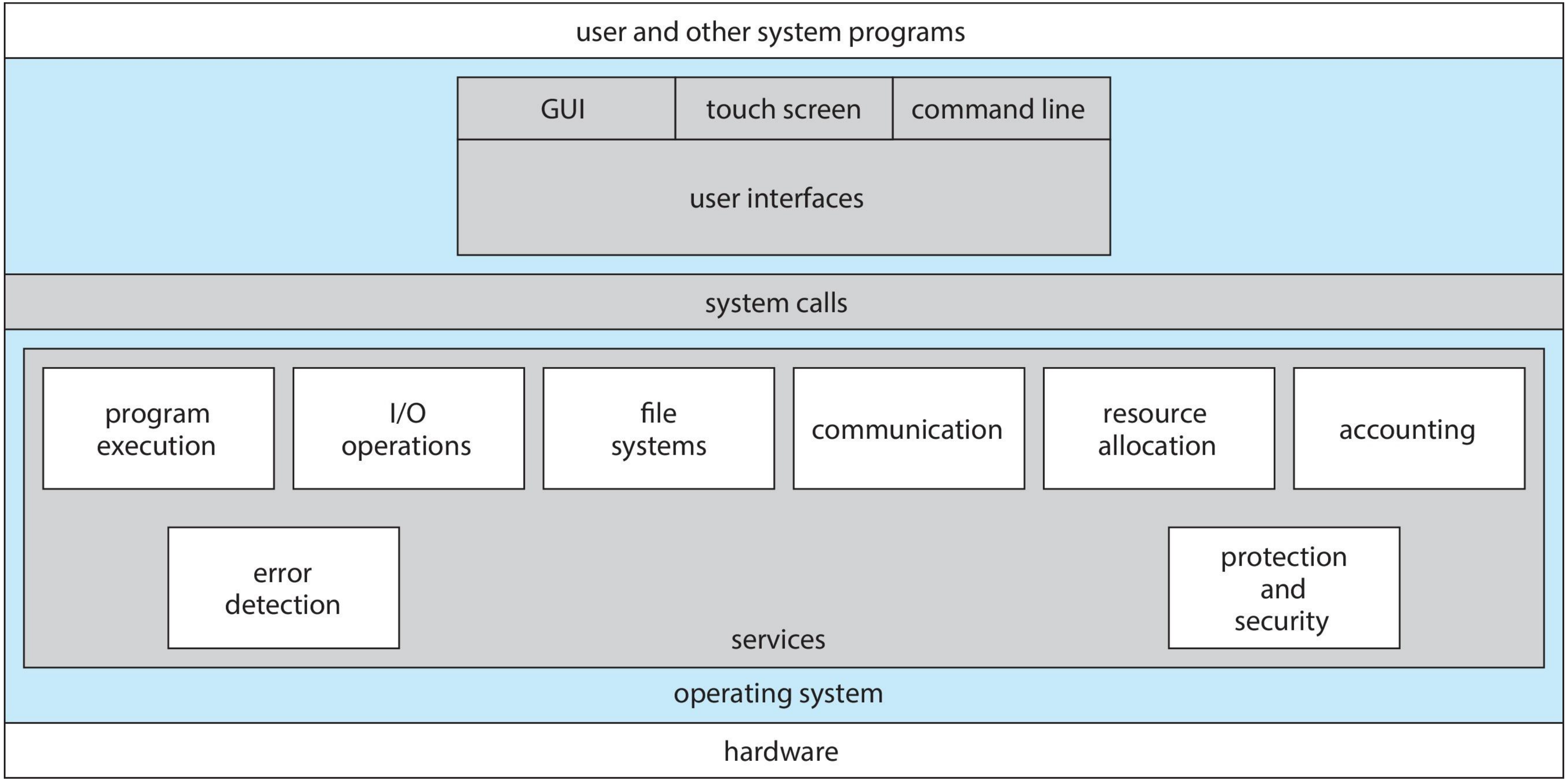
Agenda

- Operating System Services
- System Calls
- Operating System Operations
- Operating System Structure

Operating System Services

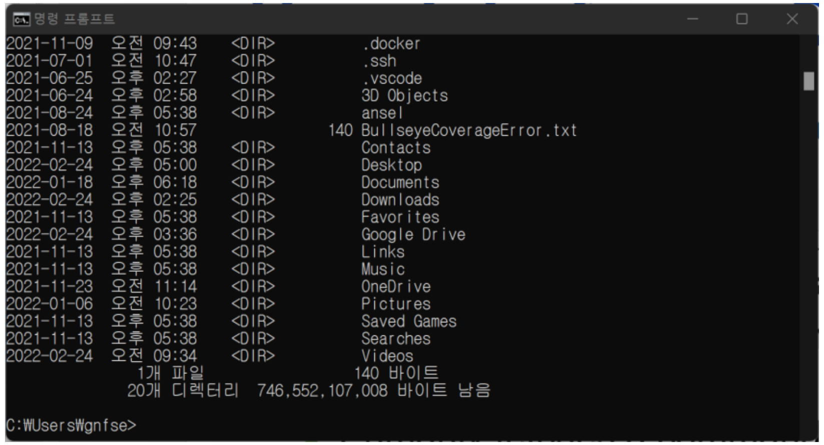
- Operating systems provide an environment for execution of programs and services to programs and users
- Services for user
  - User interface
  - Program execution
  - I/O operations
  - File-system manipulation
  - Communications
  - Error detection
- Functions for efficient operation of system itself
  - Resource allocation
  - Logging
  - Protection and security

Operating System Services



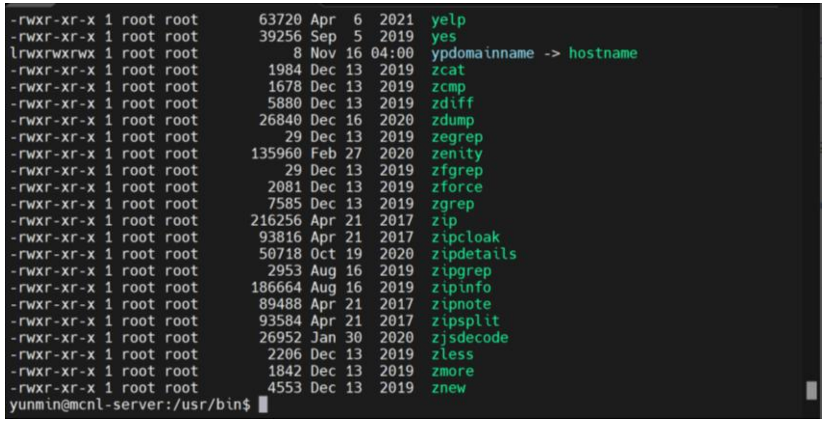
User Interfaces: CLI

- Command line interface or command interpreter allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – shells
  - Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes just names of programs



```
2021-11-09  오전 09:43    <DIR>          .docker
2021-07-01  오전 10:47    <DIR>          .ssh
2021-06-25  오후 02:27    <DIR>          .vscode
2021-06-24  오후 02:58    <DIR>          3D Objects
2021-08-24  오후 05:38    <DIR>          ansel
2021-08-18  오전 10:57               140 BullseyeCoverageError.txt
2021-11-13  오후 05:38    <DIR>          Contacts
2022-02-24  오후 05:00    <DIR>          Desktop
2022-01-18  오후 06:18    <DIR>          Documents
2022-02-24  오후 02:25    <DIR>          Downloads
2021-11-13  오후 05:38    <DIR>          Favorites
2022-02-24  오후 03:36    <DIR>          Google Drive
2021-11-13  오후 05:38    <DIR>          Links
2021-11-13  오후 05:38    <DIR>          Music
2021-11-23  오전 11:14    <DIR>          OneDrive
2022-01-06  오전 10:23    <DIR>          Pictures
2021-11-13  오후 05:38    <DIR>          Saved Games
2021-11-13  오후 05:38    <DIR>          Searches
2022-02-24  오전 09:34    <DIR>          Videos
               1개 파일                 140 바이트
              20개 디렉터리  746,552,107,008 바이트 남음

C:￦Users￦gnfse>
```

<Windows Command Prompt>

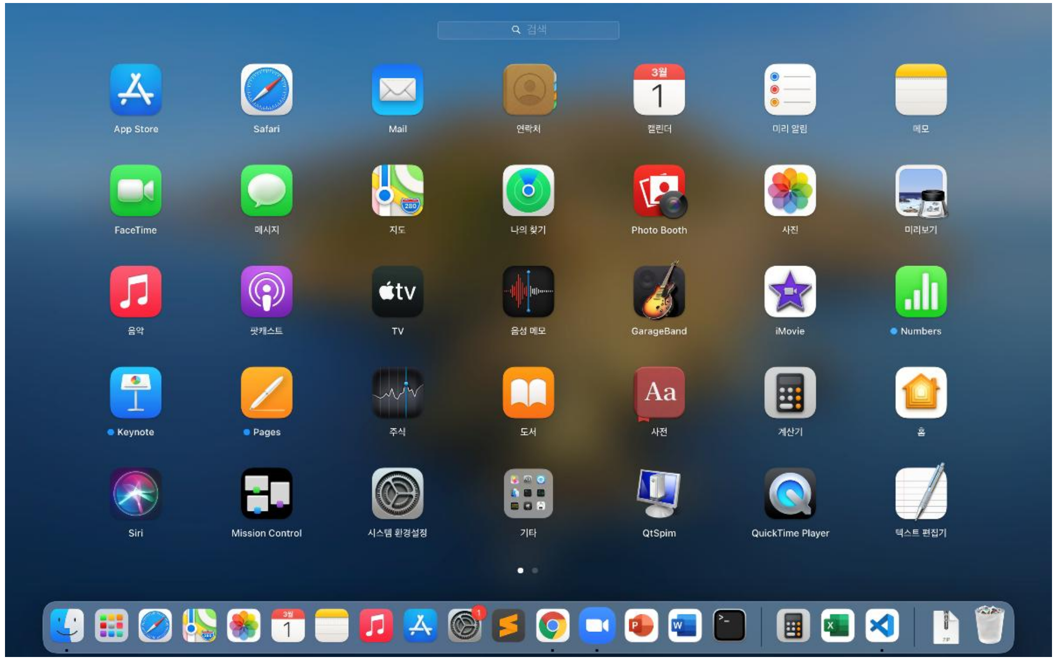


```
-rwxr-xr-x 1 root root      63720 Apr  6  2021  yelp
-rwxr-xr-x 1 root root      39256 Sep  5  2019  yes
lrwxrwxrwx 1 root root          8 Nov 16 04:00  ypdomainname -> hostname
-rwxr-xr-x 1 root root       1984 Dec 13  2019  zcat
-rwxr-xr-x 1 root root       1678 Dec 13  2019  zcmp
-rwxr-xr-x 1 root root       5880 Dec 13  2019  zdiff
-rwxr-xr-x 1 root root      26840 Dec 16  2020  zdump
-rwxr-xr-x 1 root root         29 Dec 13  2019  zegrep
-rwxr-xr-x 1 root root     135960 Feb 27  2020  zenity
-rwxr-xr-x 1 root root         29 Dec 13  2019  zfgrep
-rwxr-xr-x 1 root root       2081 Dec 13  2019  zforce
-rwxr-xr-x 1 root root       7585 Dec 13  2019  zgrep
-rwxr-xr-x 1 root root     216256 Apr 21  2017  zip
-rwxr-xr-x 1 root root      93816 Apr 21  2017  zipcloak
-rwxr-xr-x 1 root root      50718 Oct 19  2020  zipdetails
-rwxr-xr-x 1 root root       2953 Aug 16  2019  zipgrep
-rwxr-xr-x 1 root root     186664 Aug 16  2019  zipinfo
-rwxr-xr-x 1 root root      89488 Apr 21  2017  zipnote
-rwxr-xr-x 1 root root      93584 Apr 21  2017  zipsplit
-rwxr-xr-x 1 root root      26952 Jan 30  2020  zjsdecode
-rwxr-xr-x 1 root root       2206 Dec 13  2019  zless
-rwxr-xr-x 1 root root       1842 Dec 13  2019  zmore
-rwxr-xr-x 1 root root       4553 Dec 13  2019  znew
yunmin@mcnl-server:/usr/bin$
```

<Linux bash>

User Interfaces: GUI

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
  - Invented at Xerox PARC



<mac OS>

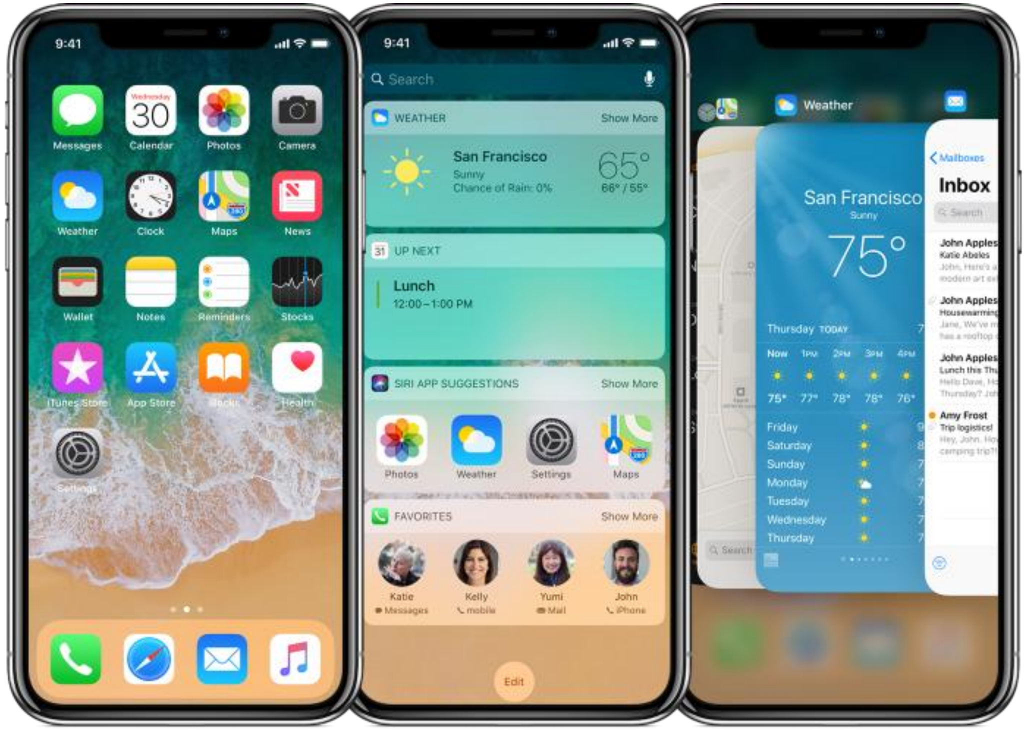
User Interfaces: Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands (+AI)

What can I ask Siri?

Siri can help you do the things you do every day — and on any of your Apple devices. All you have to do is ask.

https://support.apple.com/ko-kr/siri



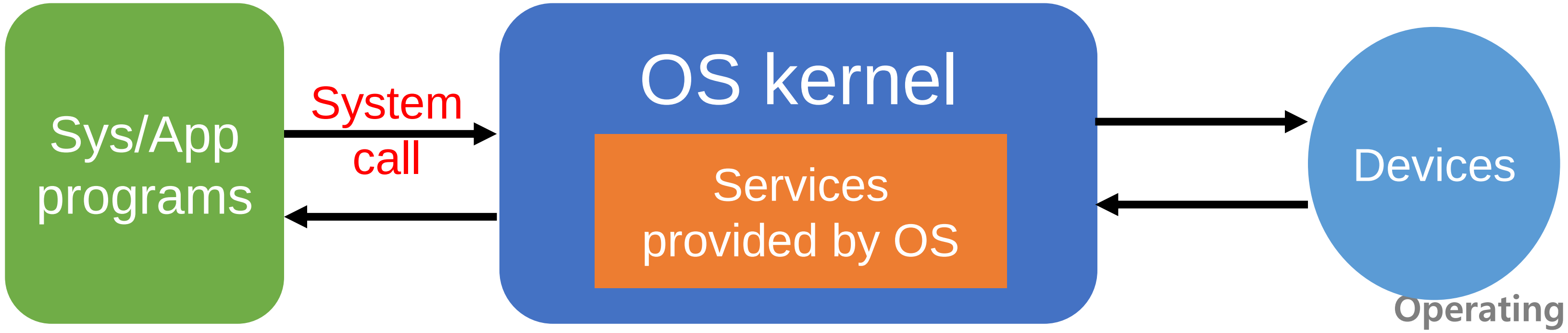
https://medium.com/dsgnrs/iphone-x-ui-guidelines-screen-details-and-layout-9ecb795c205a

Agenda

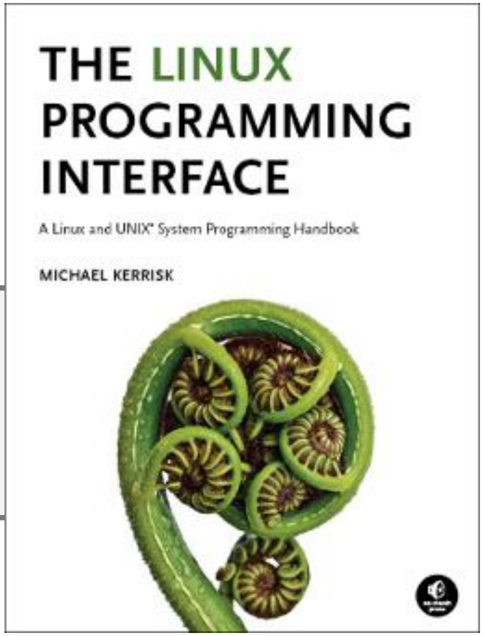
- Operating System Services
- System Calls
- Operating System Operations
- Operating System Structure

System Calls

- System calls provide an interface to the services made available by an operating system
    - “Function calls to OS kernel available through interrupt”
    - Generally, provided as interrupt handlers written in C/C++ or assembly.
    - A mechanism to transfer control safely from lesser privileged modes to higher privileged modes.
        - Ex) POSIX system calls: open, close, read, write, fork, kill, wait, …
    - Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use



Example of System Calls


THE LINUX PROGRAMMING INTERFACE
A Linux and UNIX® System Programming Handbook
MICHAEL KERRISK

- open()

```
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Open and possibly create a file

- Open the file specified by pathname
- If the specified file does not exist, it may optionally be created
- Return value
  - Success: file descriptor (nonnegative integer)
  - Error: -1
    - *errno* is set to indicate error

- flags
  - it must include one of access modes: O_RDONLY, O_WRONLY, and O_RDWR
  - zero or more file creation flags and file status flags can be bitwise-or'd in flags.
    - O_CREAT, O_TRUNC, O_DIRECTORY,..
- mode
  - specifies the file mode bits (permission) to be applied when a new file is created

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

Example of System Calls

- read()

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Read from a file descriptor

- Reads up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*
- Return value
  - Success: the number of bytes read
    - file position is advanced by this number
  - Fail: -1
    - *errno* is set to indicate error

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

Example of System Calls

- write()

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

Write to a file descriptor

- Writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*
- Return value
  - Success: the number of bytes written is returned
    - file position is advanced by this number
  - Fail: -1
    - *errno* is set to indicate error

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

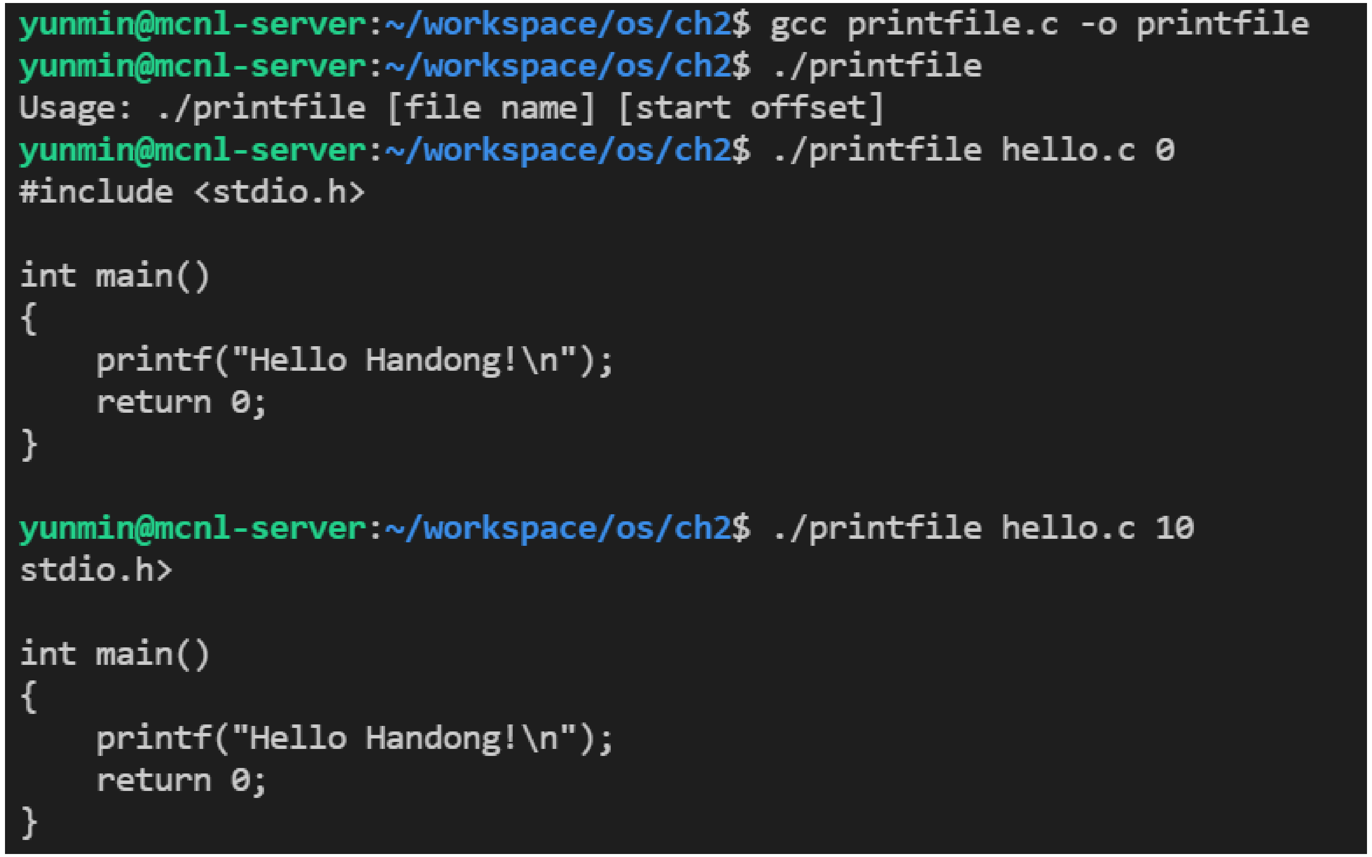
Example of System Calls

- System call sequence to print the contents of file

```
int main(int argc, char *argv[]) {
    int fd;
    int start_offset;
    int read_bytes;
    char filename[FILELEN];
    char buf[BUFSIZE];

    if (argc < 3) {
        fprintf(stderr ,"Usage: %s [file name]
                [start offset] \n", argv[0]);
        exit(1);
    }

    strcpy(filename, argv[1]);      // set file name
    start_offset = atoi(argv[2]);   // set start offset

    // open source file
    fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror(filename);
        exit(1);
    }

    // set file position
    lseek(fd, start_offset, SEEK_SET);

    // read data from file
    while ((read_bytes = read(fd, buf, BUFSIZE)) > 0) {
        // print file data
        write(1, buf, read_bytes);
    }
    printf("\n\n");

    // close file
    close(fd);

    return 0;
}
```

Example of System Calls

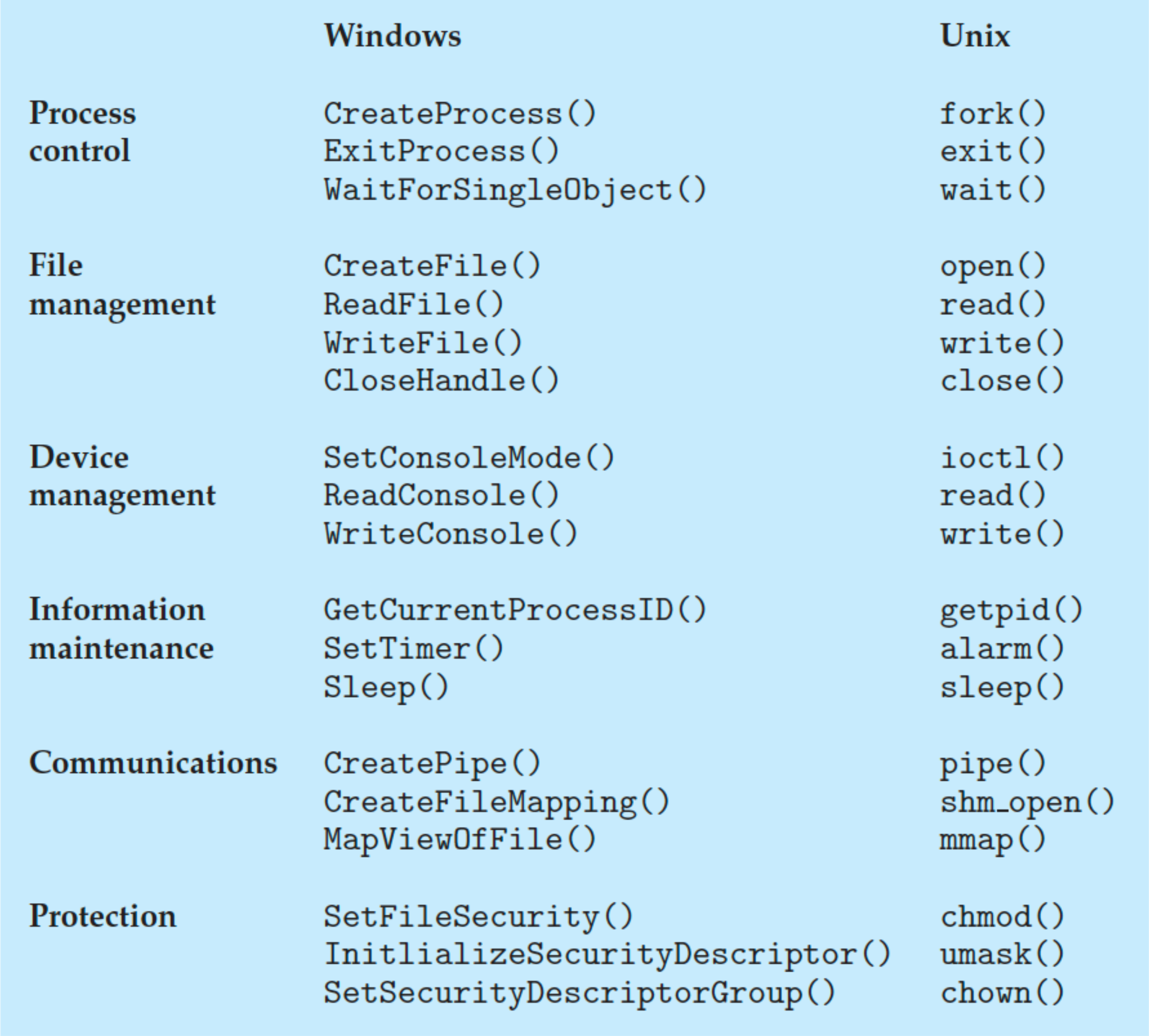
- System call sequence to print the contents of file
  - Results



```
yunmin@mcnl-server:~/workspace/os/ch2$ gcc printfile.c -o printfile
yunmin@mcnl-server:~/workspace/os/ch2$ ./printfile
Usage: ./printfile [file name] [start offset]
yunmin@mcnl-server:~/workspace/os/ch2$ ./printfile hello.c 0
#include <stdio.h>

int main()
{
    printf("Hello Handong!\n");
    return 0;
}

yunmin@mcnl-server:~/workspace/os/ch2$ ./printfile hello.c 10
stdio.h>

int main()
{
    printf("Hello Handong!\n");
    return 0;
}
```

Types of System Calls

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



	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()

System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation: create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
  - Status information: date, time, system information, performance, etc.
  - Programming language support: compiler, assembler, debuggers, interpreter
  - Program loading and execution: loader, linkage editors, etc.
  - Communications: networks
  - Background services: launch at boot time, process scheduling, logging, etc
  - Application programs: run by users
- Most users' view of the operation system is defined by system programs, not the actual system calls

Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
  - Own file formats, etc
- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- Application Binary Interface (ABI) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc

Agenda

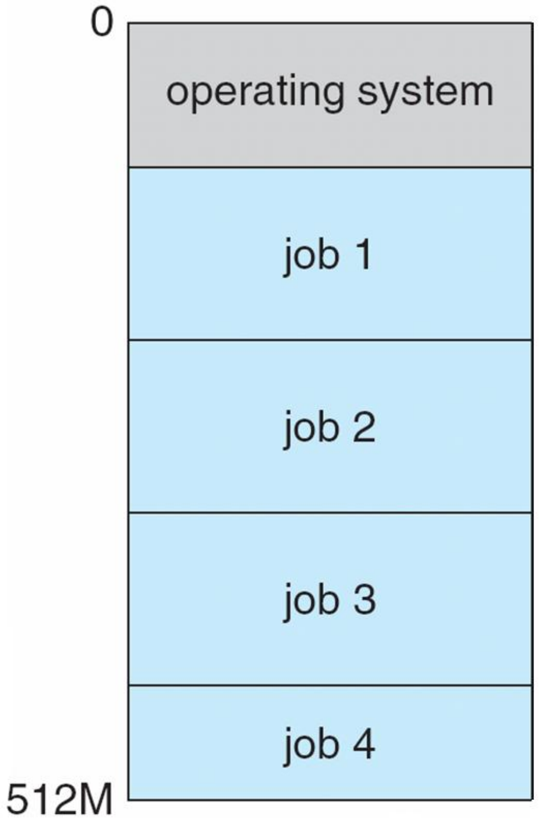
- Operating System Services
- System Calls
- Operating System Operations
- Operating System Structure

Operating-System Operations

- Bootstrap program
  - simple code to initialize the system, load the kernel
- Kernel loads
- Starts system daemons (services provided outside of the kernel)
- Kernel interrupt driven (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (exception or trap):
    - Software error (e.g., division by zero)
    - Request for operating system service: system call
    - Other process problems include infinite loop, processes modifying each other or the operating system

Multiprogramming

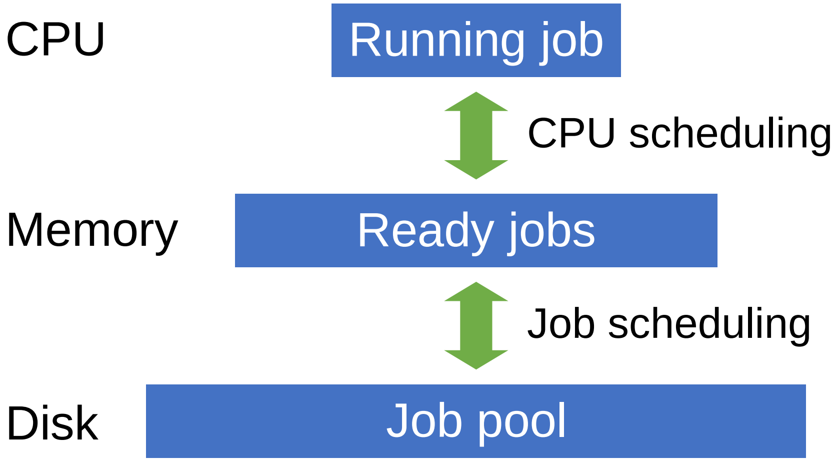
- Multiprogramming needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via job scheduling
  - When it has to wait (for I/O for example), OS switches to another job



<Memory layout for a multiprogramming system>

Multitasking

- Multitasking is logical extension of multiprogramming
  - CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing
  - Response time should be < 1 second
  - Each user has at least one program executing(process) in memory
  - CPU scheduler selects a job that is ready to run



Requirement for Modern OS

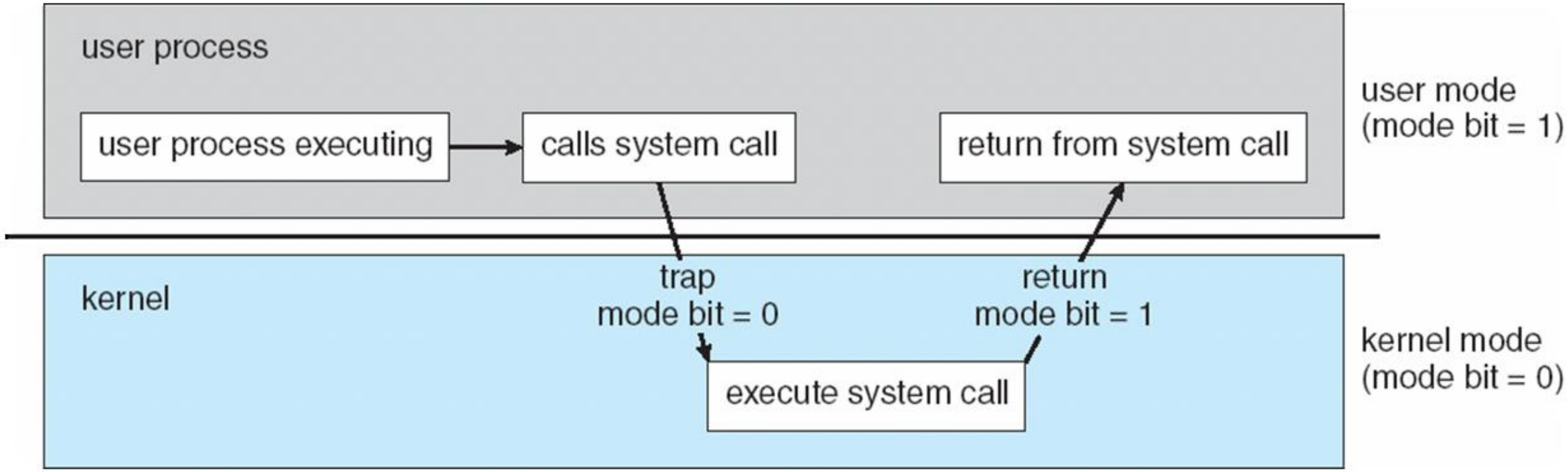
- Modern OS's are interrupt-driven programs
  - Events are signaled by interrupts, which are handled by interrupt handlers.
  - H/W and S/W resources are shared.

  ➔ **Problem: An error from a process can corrupt whole system**

- Essential requirement for multi-user OS: Error of a program should not affect to other program
  - Dangerous instructions
  - Preventing long execution of user process

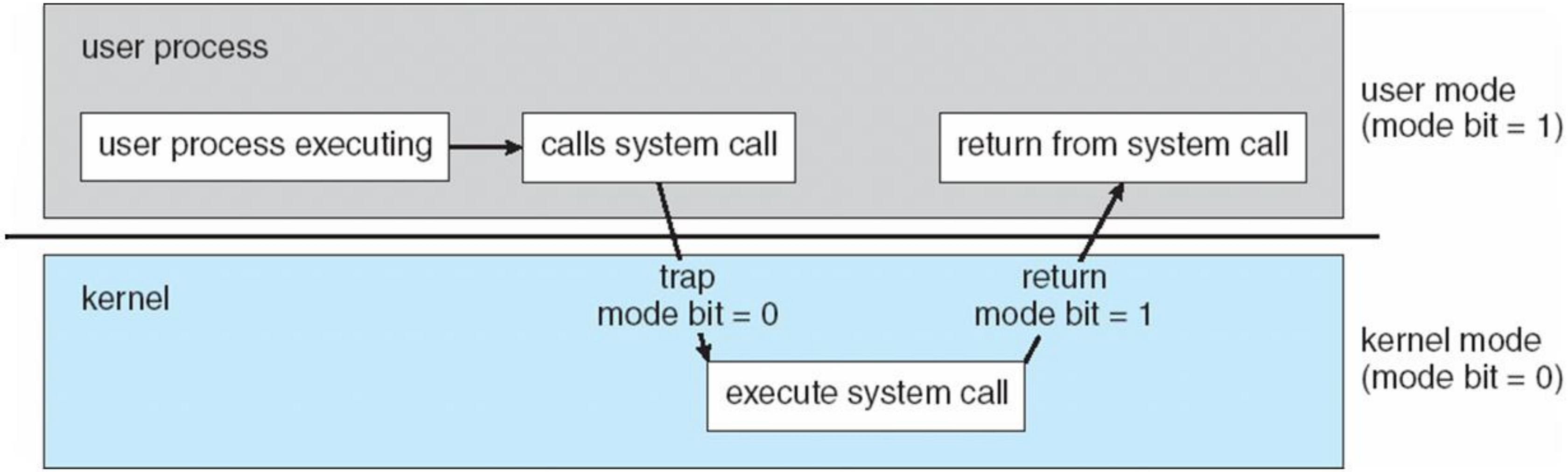
Dual-Mode Operation

- Dual-mode operation allows OS to protect itself and other system components (requires H/W support)
  - User mode
    - User defined code (application)
    - Privileged instructions, which can cause harm to other system, are prohibited
  - Kernel mode (supervisor mode, system mode, privileged mode)
    - OS code, System call → Privileged instructions are permitted



Dual-Mode Operation

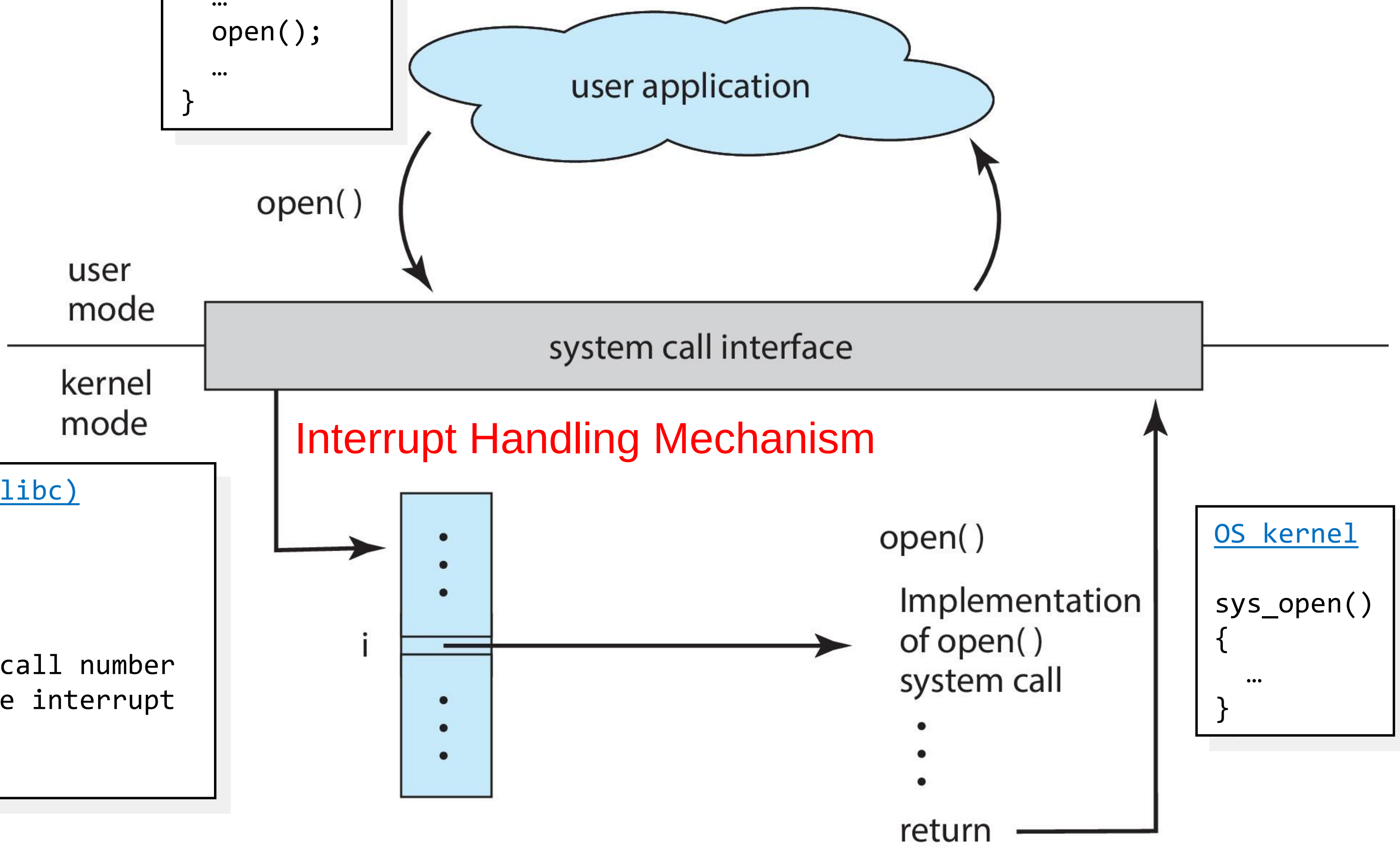
- Dual-mode operation allows OS to protect itself and other system components (requires H/W support)
  - Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code.
    - When a user is running → mode bit is “user” (1)
    - When kernel code is executing → mode bit is “kernel” (0)



System Call – OS Relationship

User program

```
int main()
{
  …
  open();
  …
}
```



System-call Interface (libc)

```
open()
{
  …
  movl 5, %eax ‘system call number
  int $0x80    ‘generate interrupt
  …
}
```

OS kernel

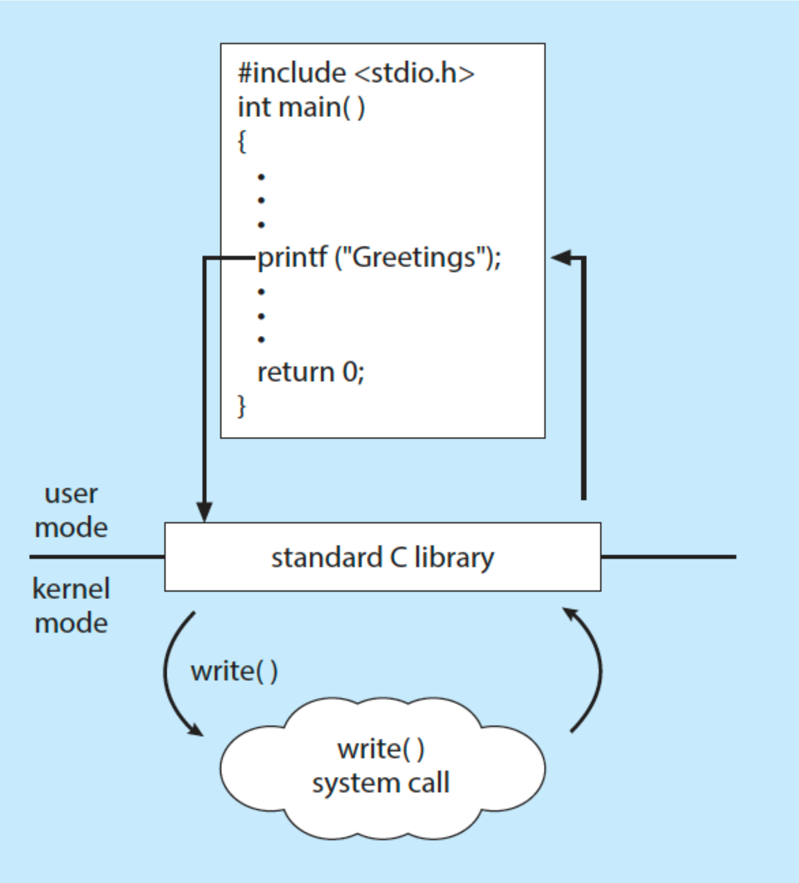
```
sys_open()
{
  …
}
```

System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
  - Passing information to the kernel
  - Switch to kernel mode
  - Any data processing and preparation for execution in kernel mode
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API

System Call Implementation

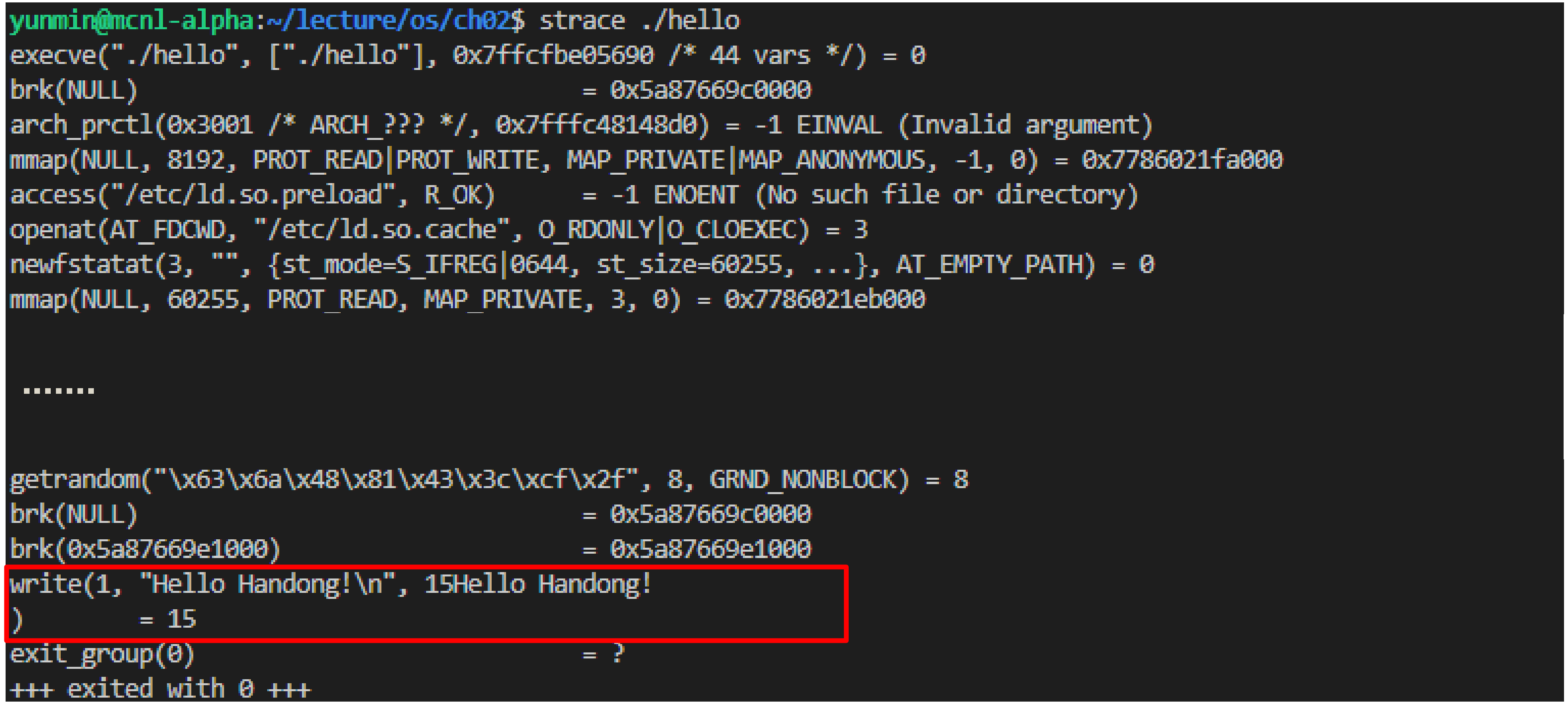
- System call vs. I/O functions in programming language
  - write() vs. printf()
    - printf() is implemented using write()
      - write(): provided by OS
      - printf(): standard function defined in C language
  - read() vs. fread()
    - fread() is implemented using read()



```
#include <stdio.h>
int main( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

System Call Implementation

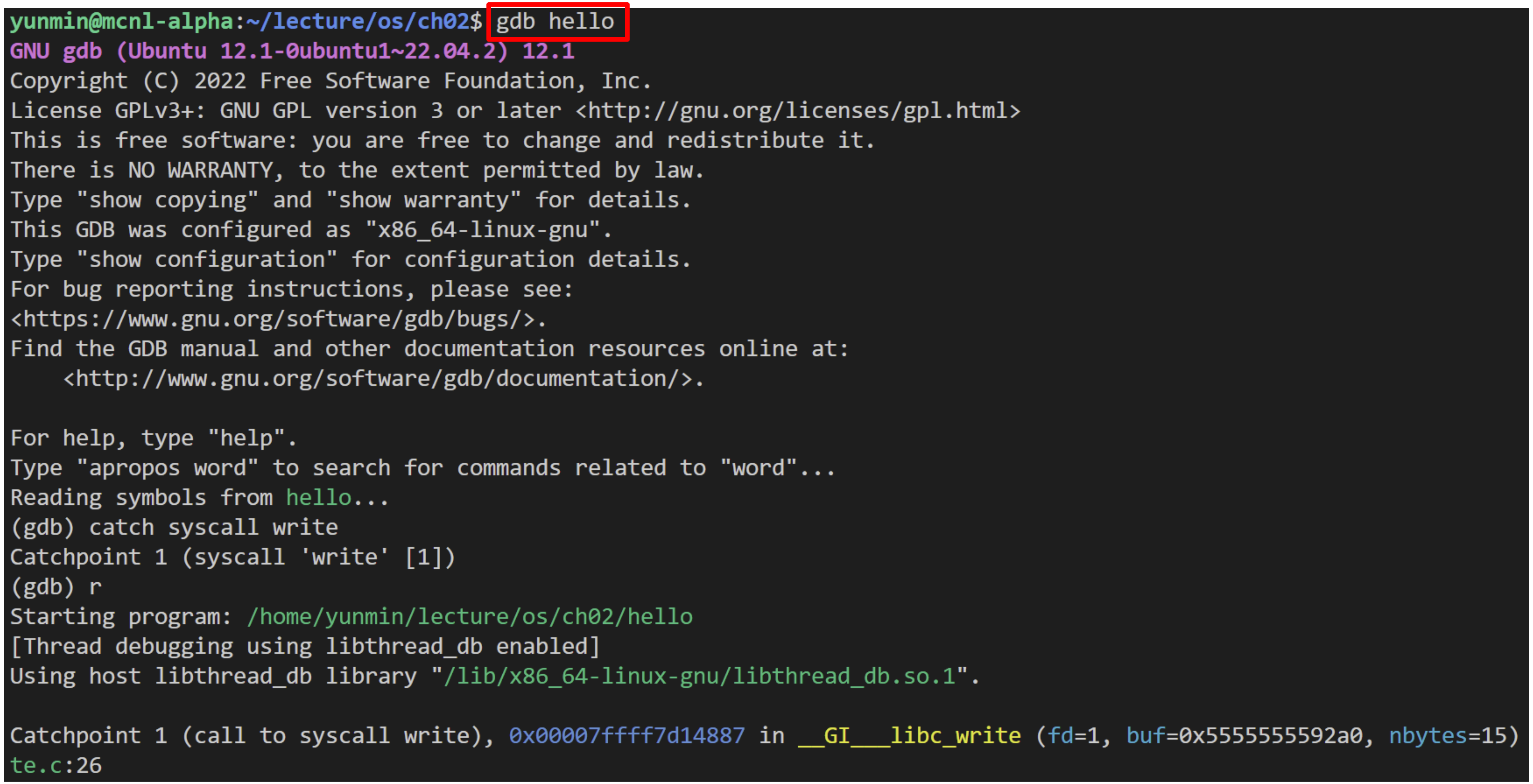
- System call vs. I/O functions in programming language
  - `strace`: traces system calls invoked by a process



```
yunmin@mcnl-alpha:~/lecture/os/ch02$ strace ./hello
execve("./hello", ["./hello"], 0x7ffcfbe05690 /* 44 vars */) = 0
brk(NULL)                               = 0x5a87669c0000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fffc48148d0) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7786021fa000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=60255, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 60255, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7786021eb000


 .......


getrandom("\x63\x6a\x48\x81\x43\x3c\xcf\x2f", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x5a87669c0000
brk(0x5a87669e1000)                     = 0x5a87669e1000
write(1, "Hello Handong!\n", 15Hello Handong!
)       = 15
exit_group(0)                           = ?
+++ exited with 0 +++
```

System Call Implementation

- Let’s try running gdb!



```
yunmin@mcnl-alpha:~/lecture/os/ch02$ gdb hello
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...
(gdb) catch syscall write
Catchpoint 1 (syscall 'write' [1])
(gdb) r
Starting program: /home/yunmin/lecture/os/ch02/hello
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Catchpoint 1 (call to syscall write), 0x00007ffff7d14887 in __GI___libc_write (fd=1, buf=0x5555555592a0, nbytes=15)
te.c:26
```

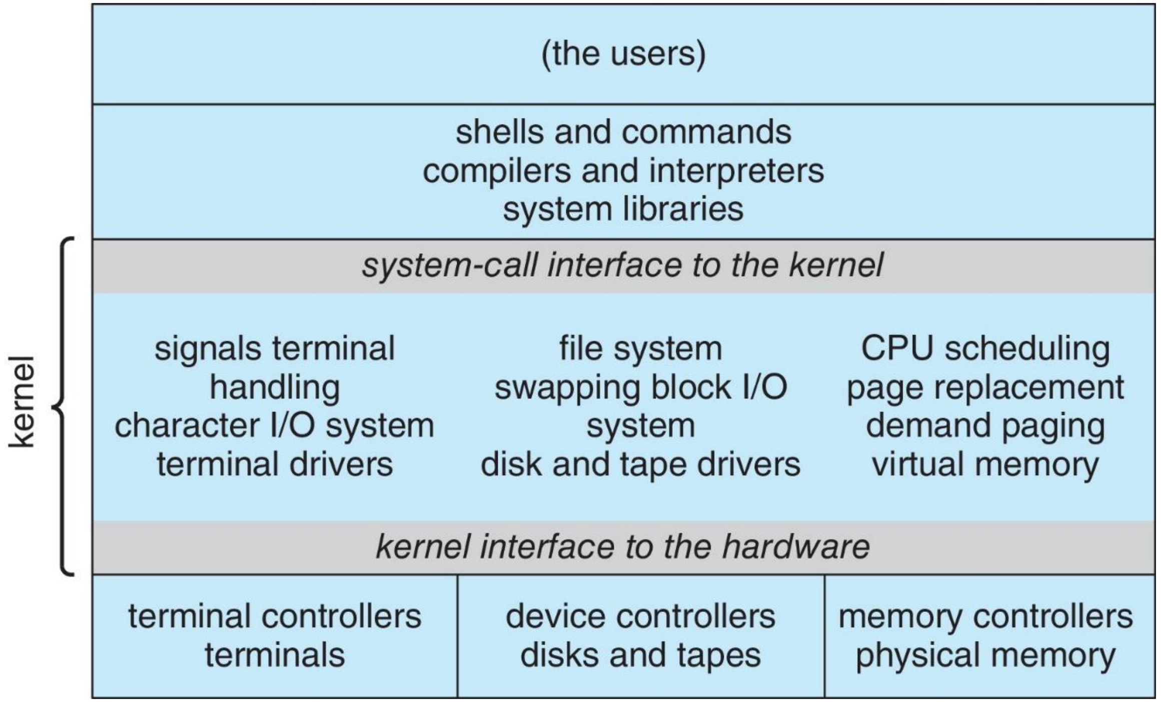
Agenda

- Operating System Services
- System Calls
- Operating System Operations
- Operating System Structures

Monolithic Structure

- Monolithic structure place all of the functionality of the kernel into a single, static library file that runs in a single address space.

<Traditional UNIX System>



(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

signals terminal handling
character I/O system
terminal drivers

file system
swapping block I/O system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

terminal controllers
terminals

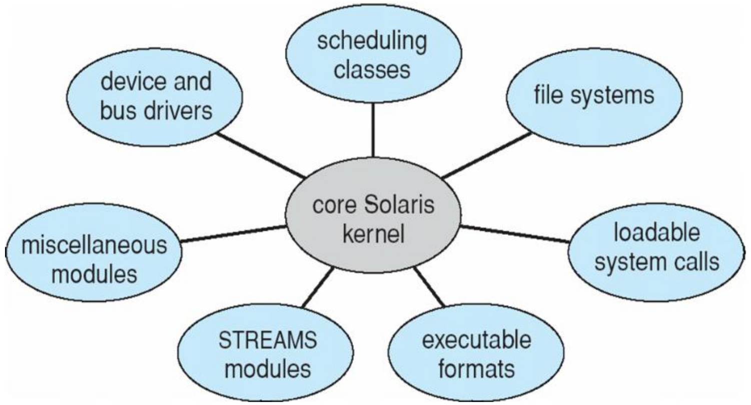
device controllers
disks and tapes

memory controllers
physical memory

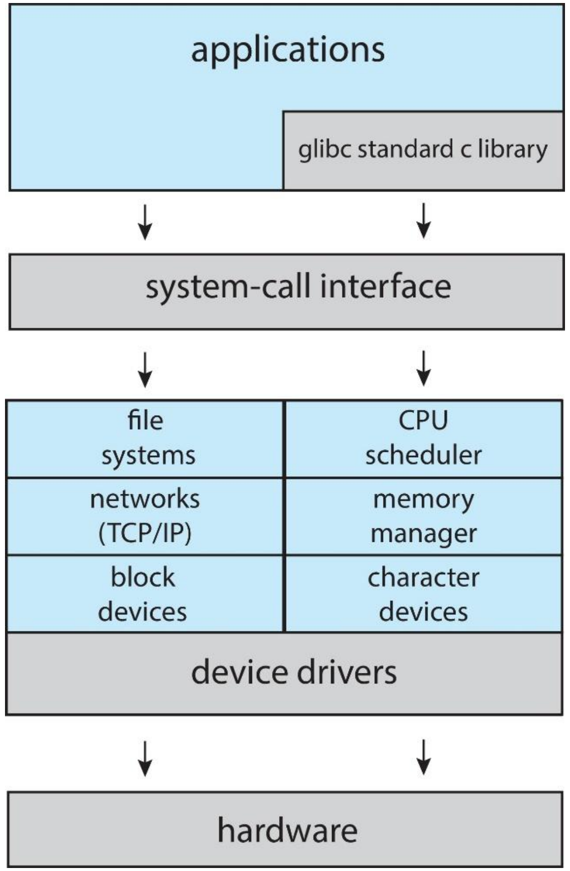
Modular Structure

- Many modern operating systems implement loadable kernel modules (LKM)
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
  - ex) Linux, Solaris, etc

<Solaris>



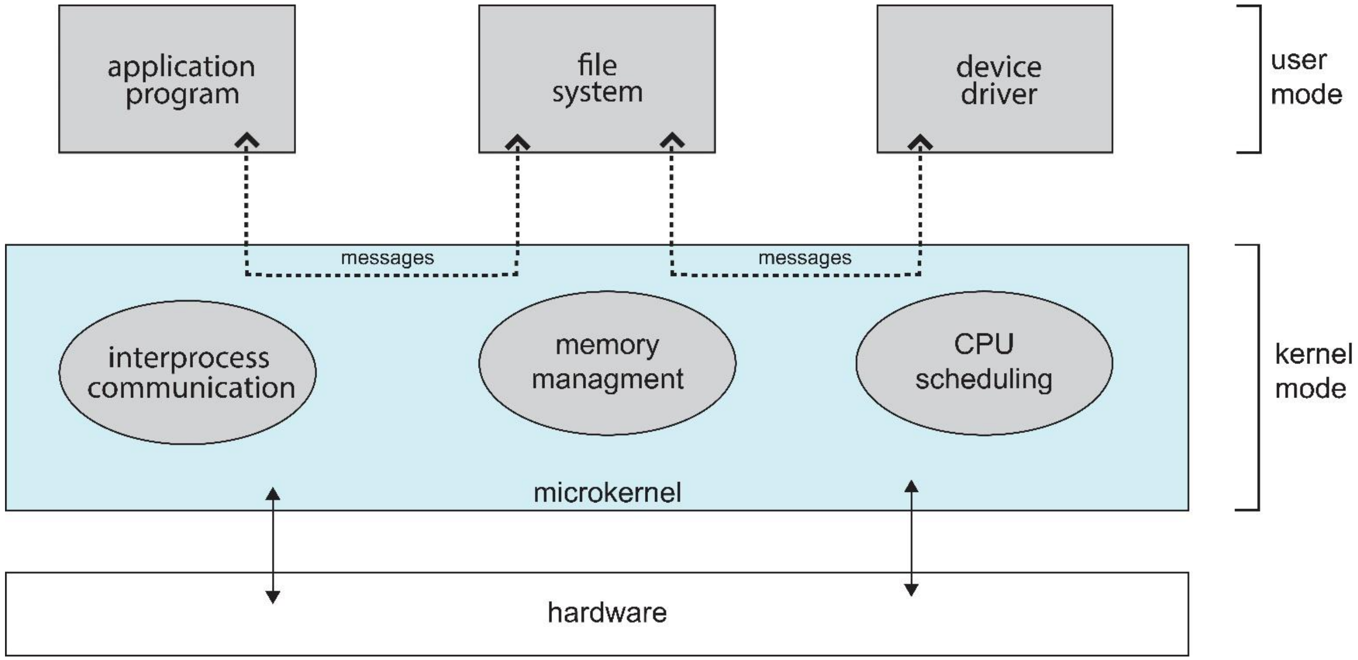
<Linux>



Monolithic + modular design (Hybrid)

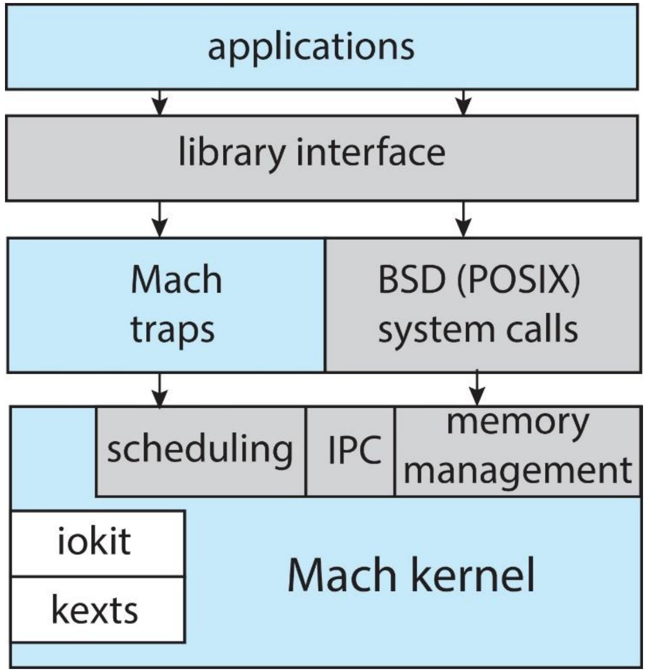
Microkernel

- Smaller kernel
  - All unessential components are not implemented in kernel but as system/user-level programs.
    - Generally, process/memory management, communication facility are in the kernel.
  - System calls are provided through message passing.



Hybrid Structure

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem personalities
  - macOS uses Mach microkernel with modules and integrates BSD subsystem.



<The structure of Darwin>

Summary of Operating System Structures

Structures	Description	Advantages	Disadvantages	Example OS
Monolithic	All OS services run in kernel space as a single program.	Fast execution, direct system call handling, efficient performance.	Hard to debug and maintain, a crash affects the whole system.	Linux, Unix, MS-DOS
Modular	A monolithic kernel with dynamically loadable modules.	Modular and flexible, allows adding/removing drivers without rebooting.	Still runs most services in kernel space, potential security issues.	Linux (with Loadable Kernel Modules), FreeBSD
Microkernel	Only essential services run in kernel space, while others run in user space.	Better security and stability, crashes in user space don't affect kernel.	Slower due to message passing between kernel and user space services.	Minix, QNX, seL4
Hybrid	A combination of monolithic and microkernel features for performance and flexibility.	Balances performance and modularity, reduces microkernel overhead.	Increased complexity, more difficult to optimize.	macOS (XNU), Windows NT, Google Fuchsia