Operating Systems

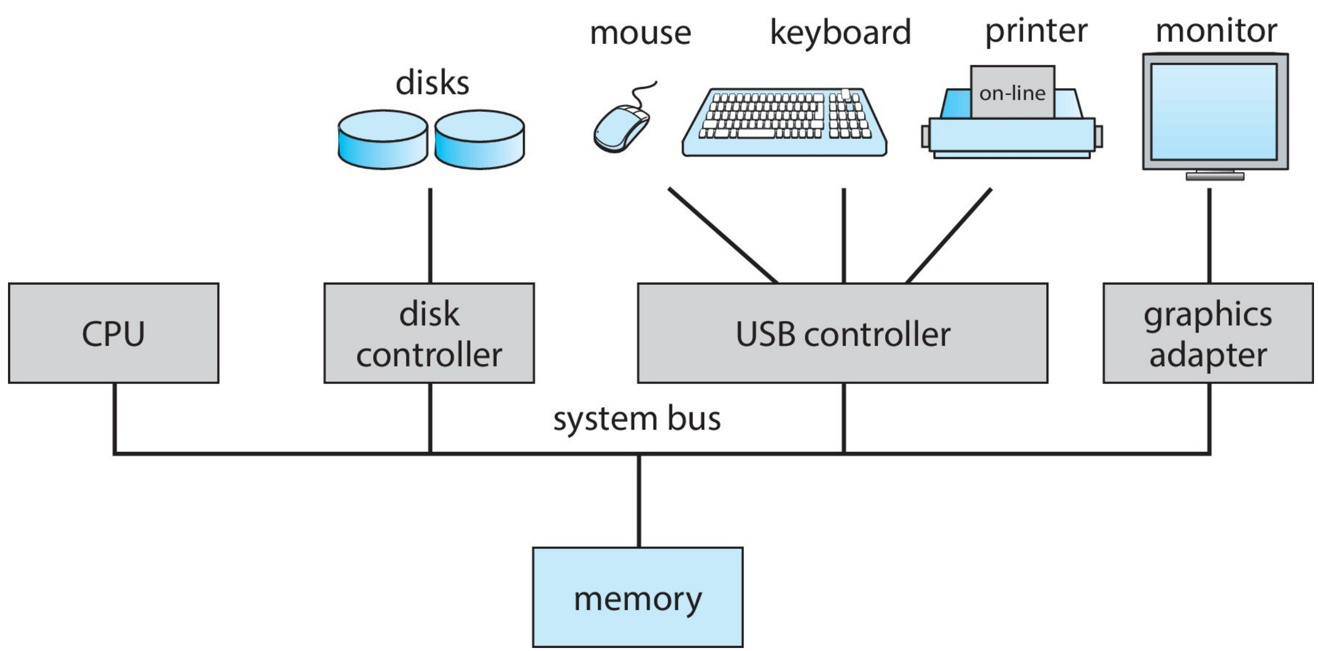
Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

Session 2: Computer and Operating System structure

Computer System Organization

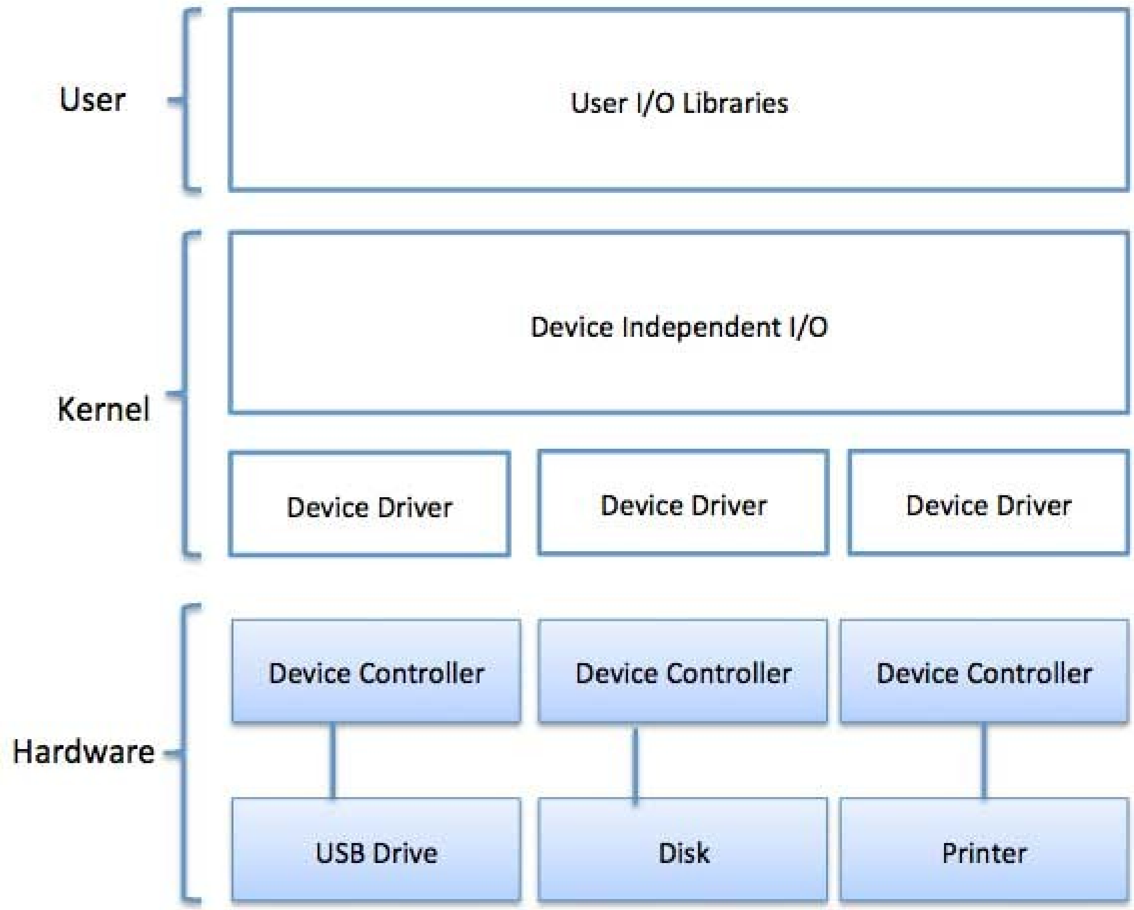
- One or more CPUs, device controllers connect through common **bus** providing access to shared memory
- Concurrent execution of CPUs and devices competing for memory cycles



Kernel

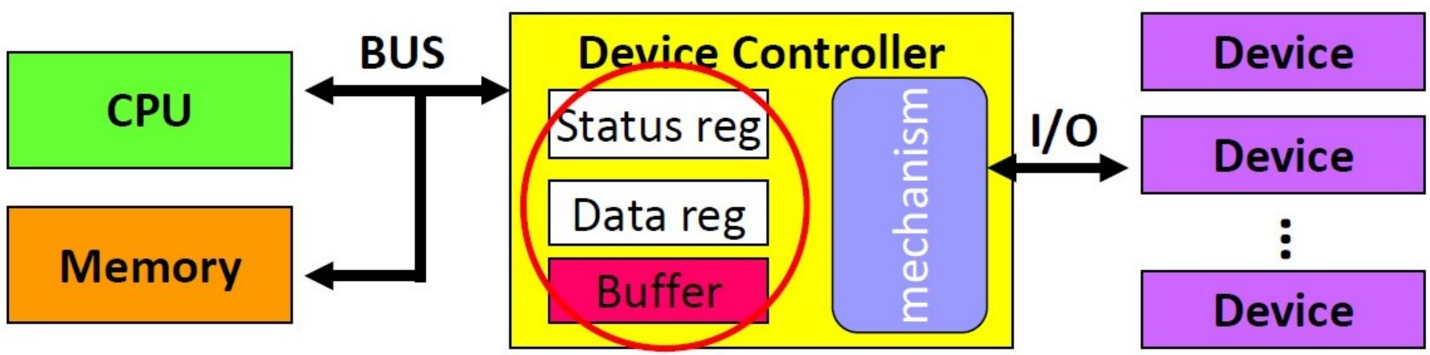
- **"The one program running at all times on the computer" is the kernel, part of the operating system**
- Everything else is either
  - A ***system program*** (ships with the operating system, but not part of the kernel) , or
  - An ***application program***, all programs not associated with the operating system
- Today's OSes for general purpose and mobile computing also include ***middleware*** – a set of software frameworks that provide additional services to application developers such as databases, multimedia, graphics

Device Controller

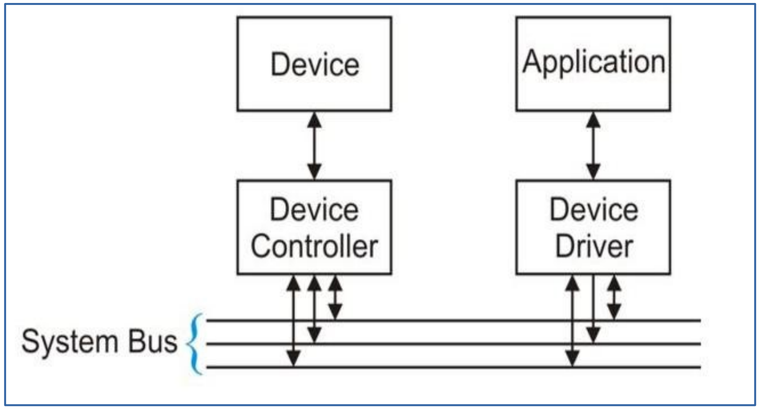


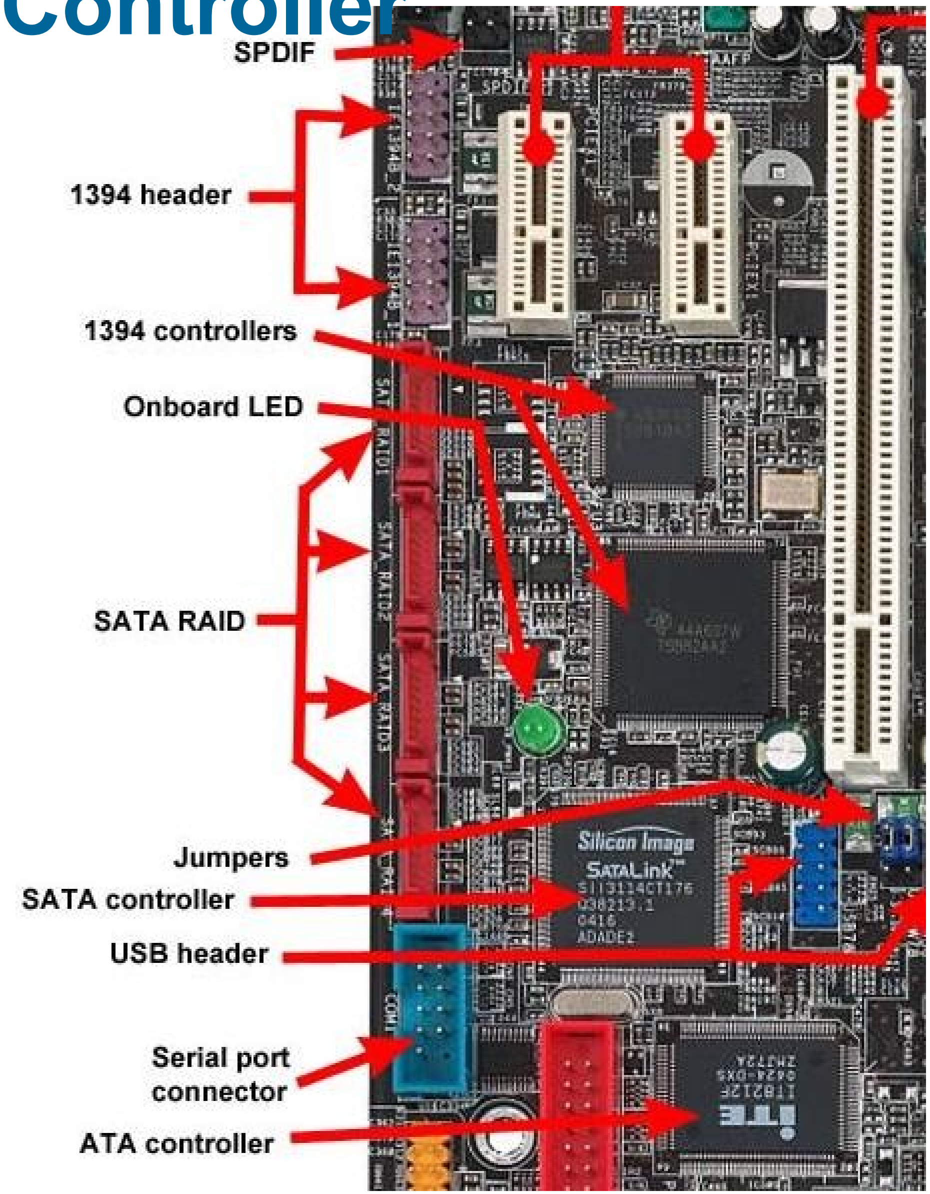
Device Controller

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- Each device controller type has an operating system **device driver** to manage it
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an **interrupt**



Device Controller

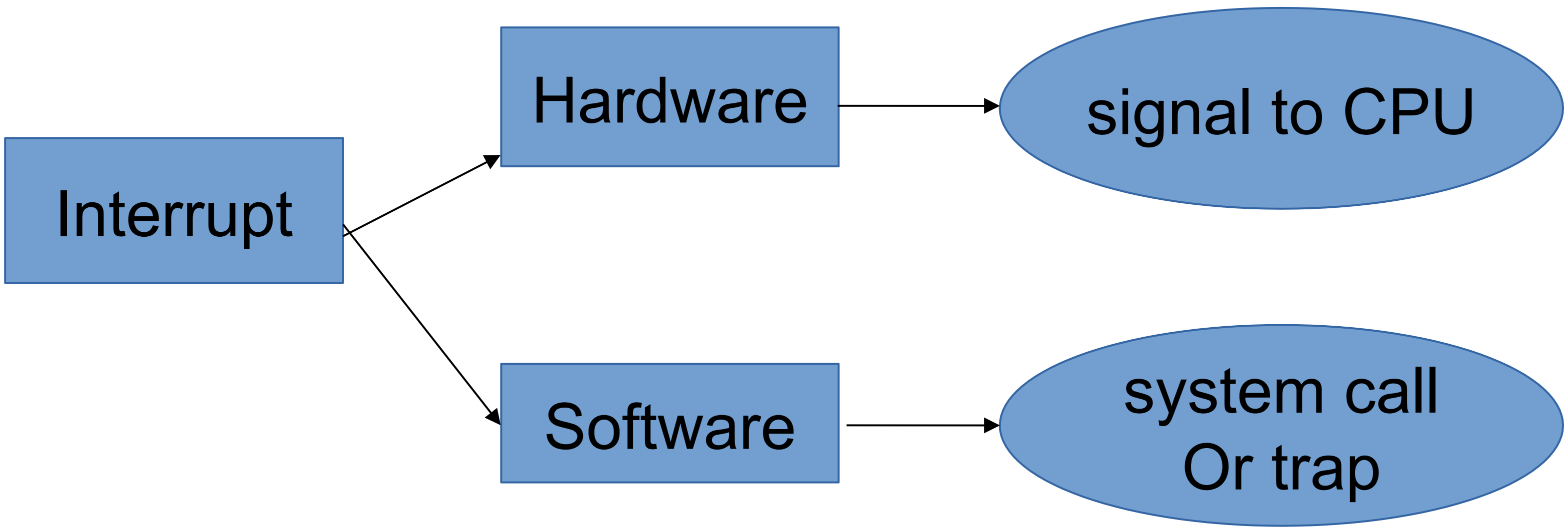




Common Functions of Interrupts

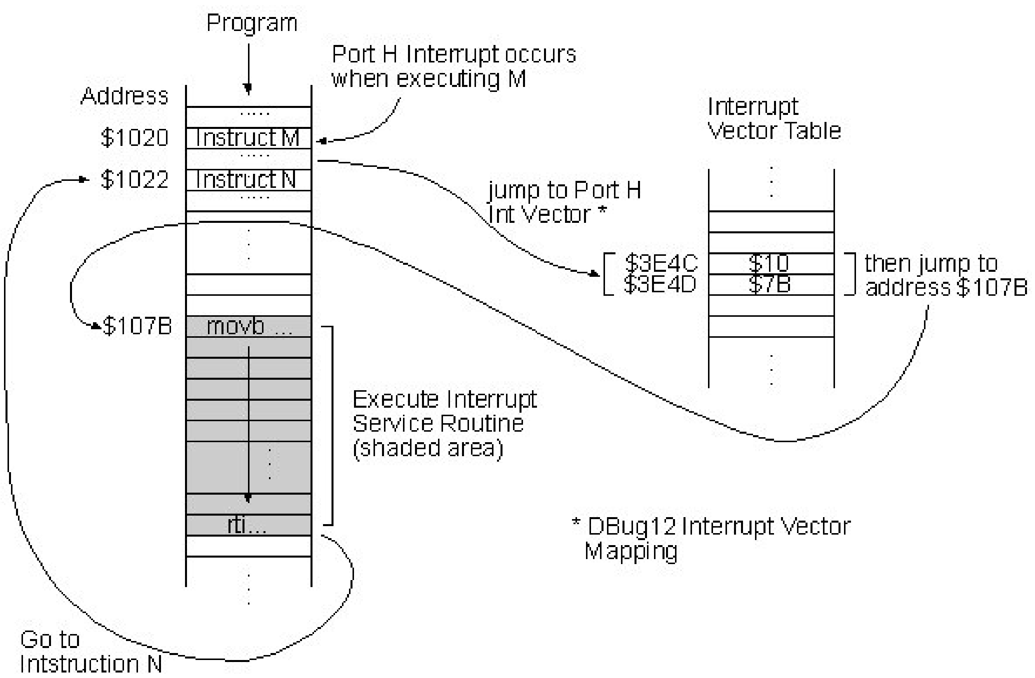
- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request
- **An operating system is interrupt driven**

Interrupt types



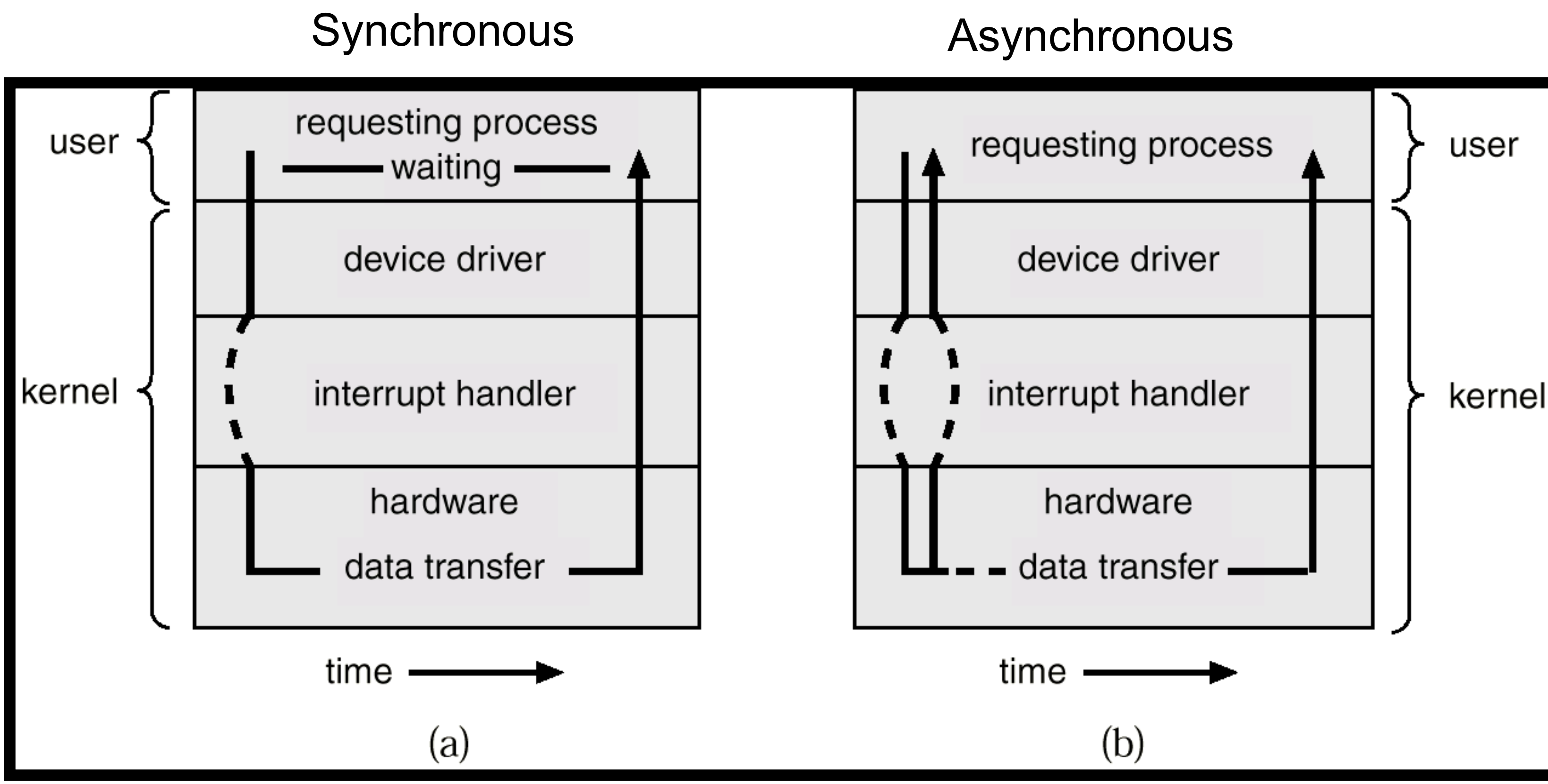
Interrupt Handling

- The operating system preserves the state of the CPU by storing the registers and the program counter
- Determines which type of interrupt has occurred:
- Separate segments of code determine what action should be taken for each type of interrupt



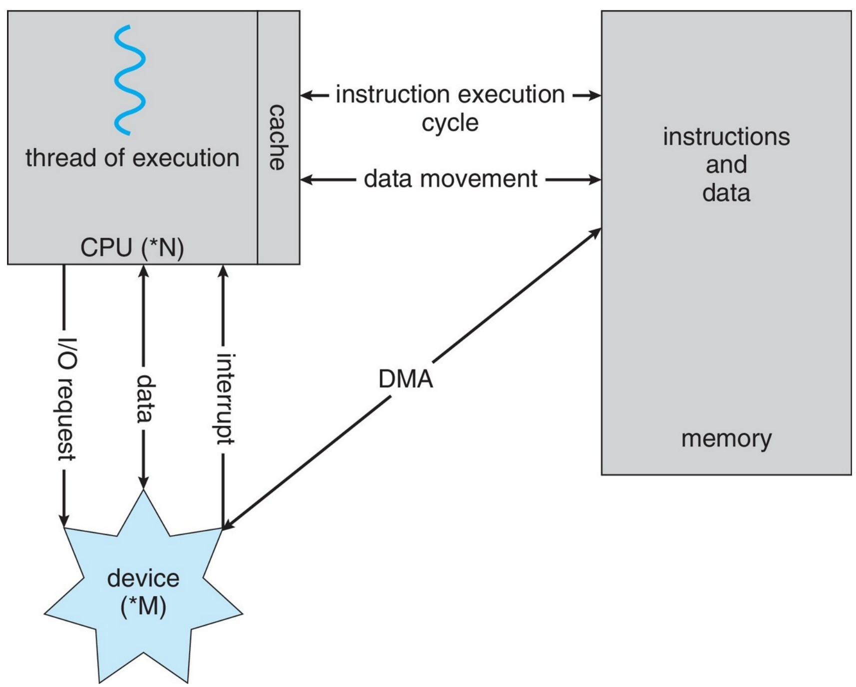
Two I/O Methods

Direct Memory Access (DMA)

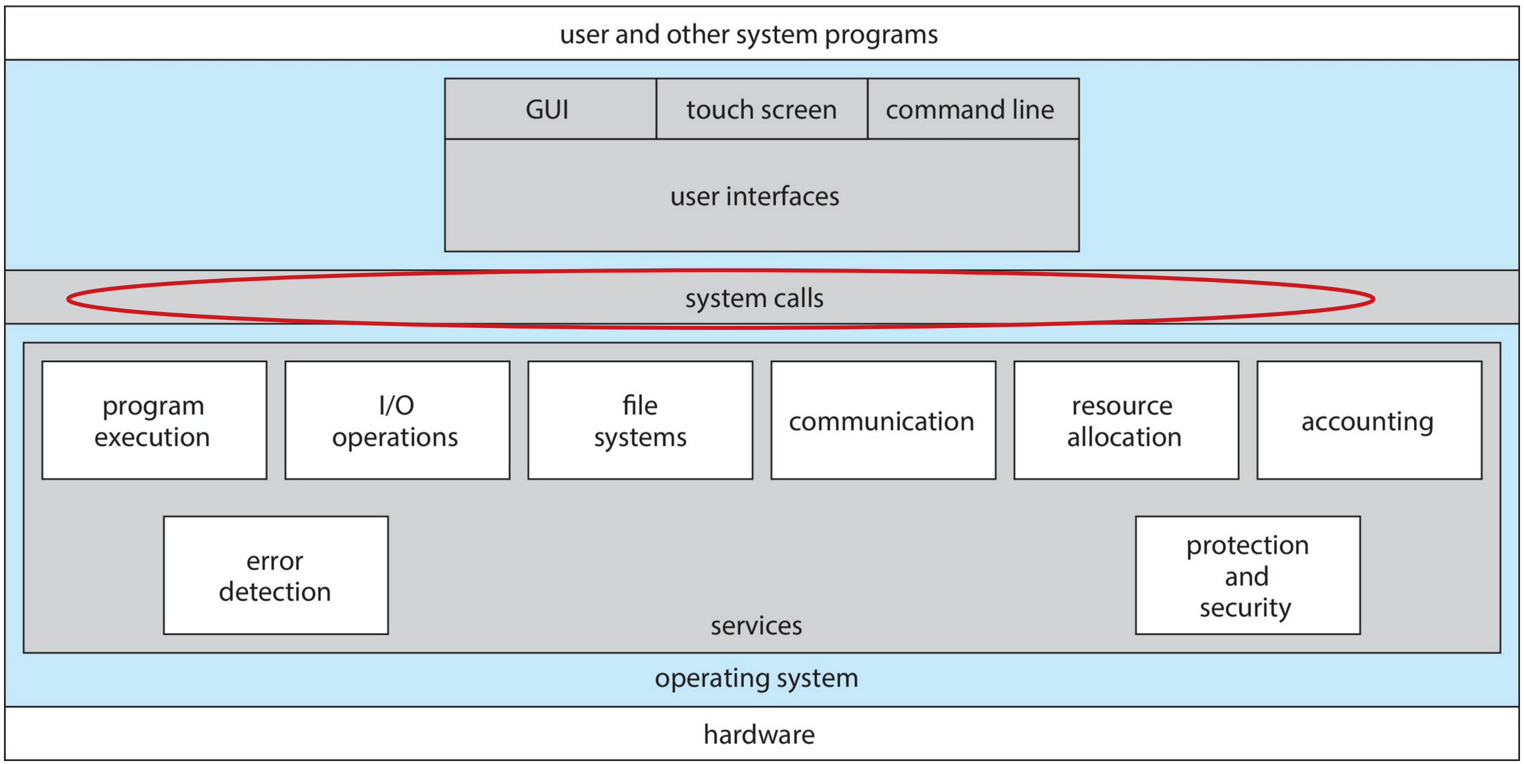


Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte



System Calls



System Calls

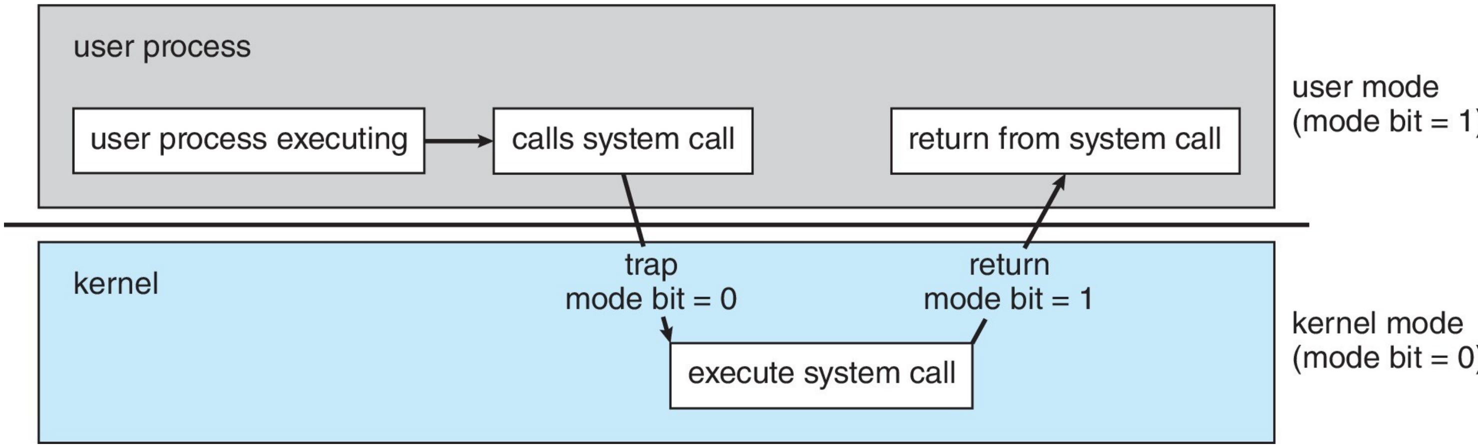
- **Programming interface to the services provided by the OS**
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
- Three most common APIs
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
  - Java API for the Java virtual machine (JVM)

System call is the method used by a process to request action by the OS. The system call service routine is a part of the OS

Dual-mode Operation

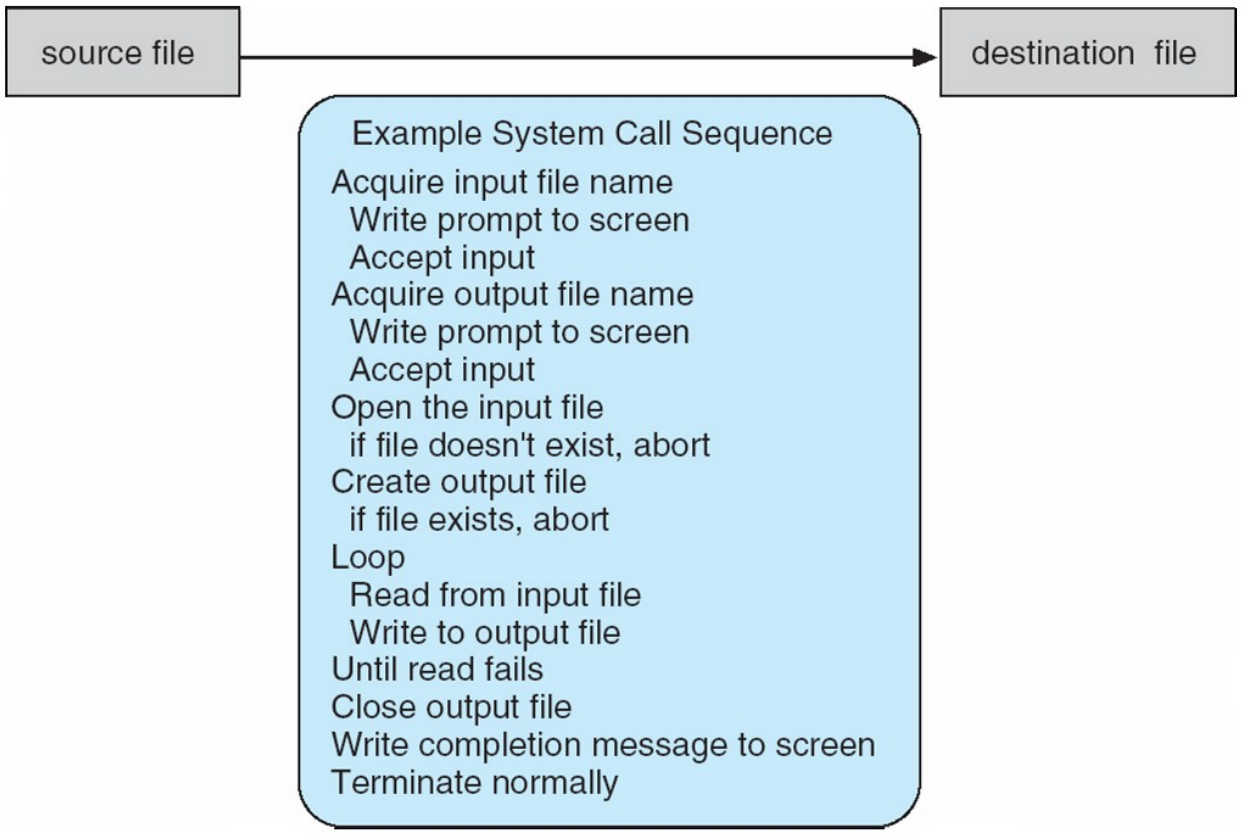
- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
- **Mode bit** provided by hardware
  - Provides ability to distinguish when system is running user code or kernel code.
  - When a user is running the mode bit is “user”
  - When kernel code is executing the mode bit is “kernel”
- How do we guarantee that user does not explicitly set the mode bit to “kernel”?
  - System call changes mode to kernel, returns from call, resets it to user
- Some instructions designated as **privileged**, only executable in kernel mode

Transition from User to Kernel Mode



Example of System Calls

- System call sequence to copy the contents of one file to another file



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value	function name	parameters
`ssize_t`	`read`	`(int fd, void *buf, size_t count)`

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

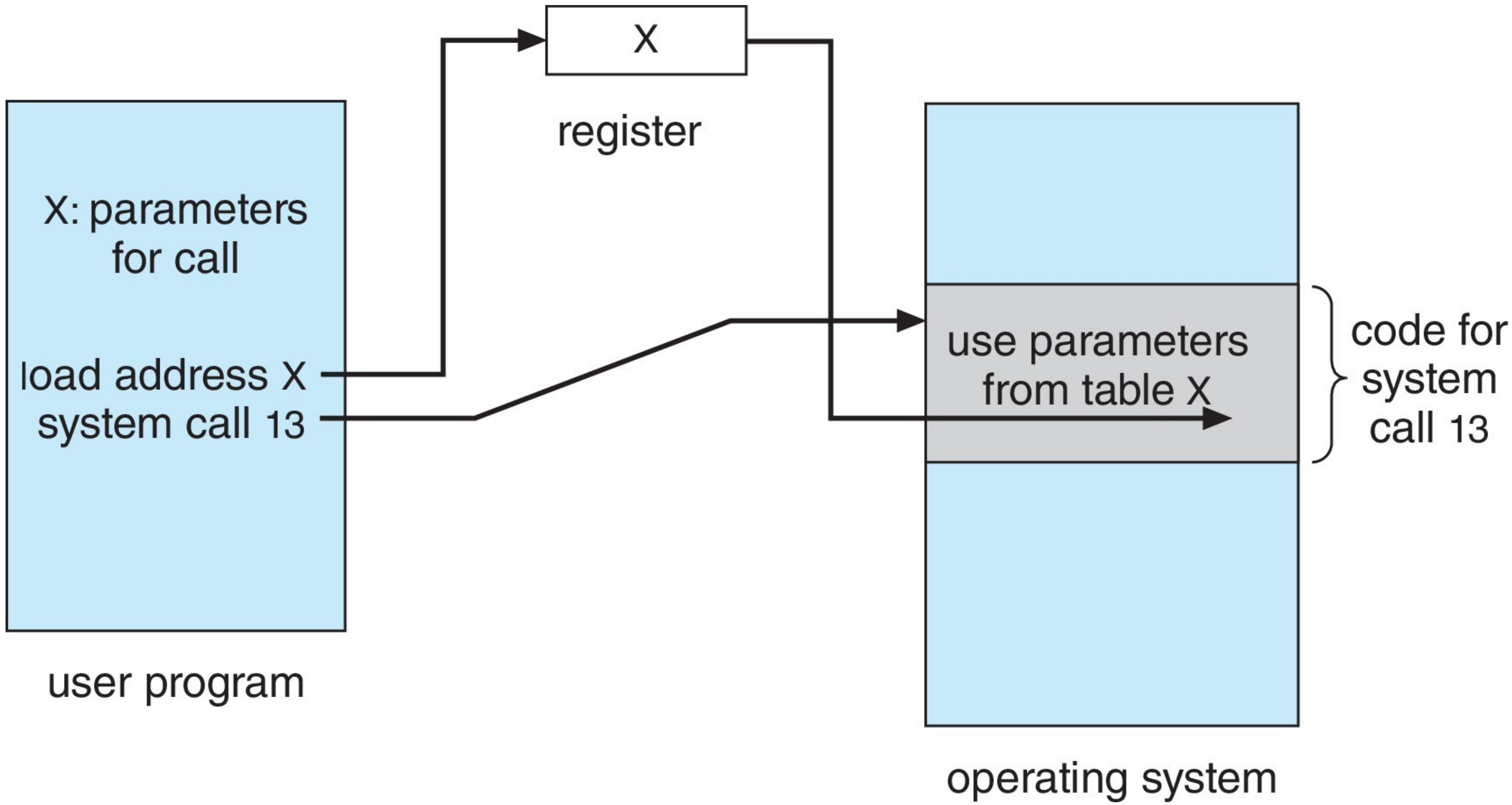
System Call Implementation

- Typically, a number is 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
- 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
    - **Managed by run-time support library (set of functions built into libraries included with compiler)**

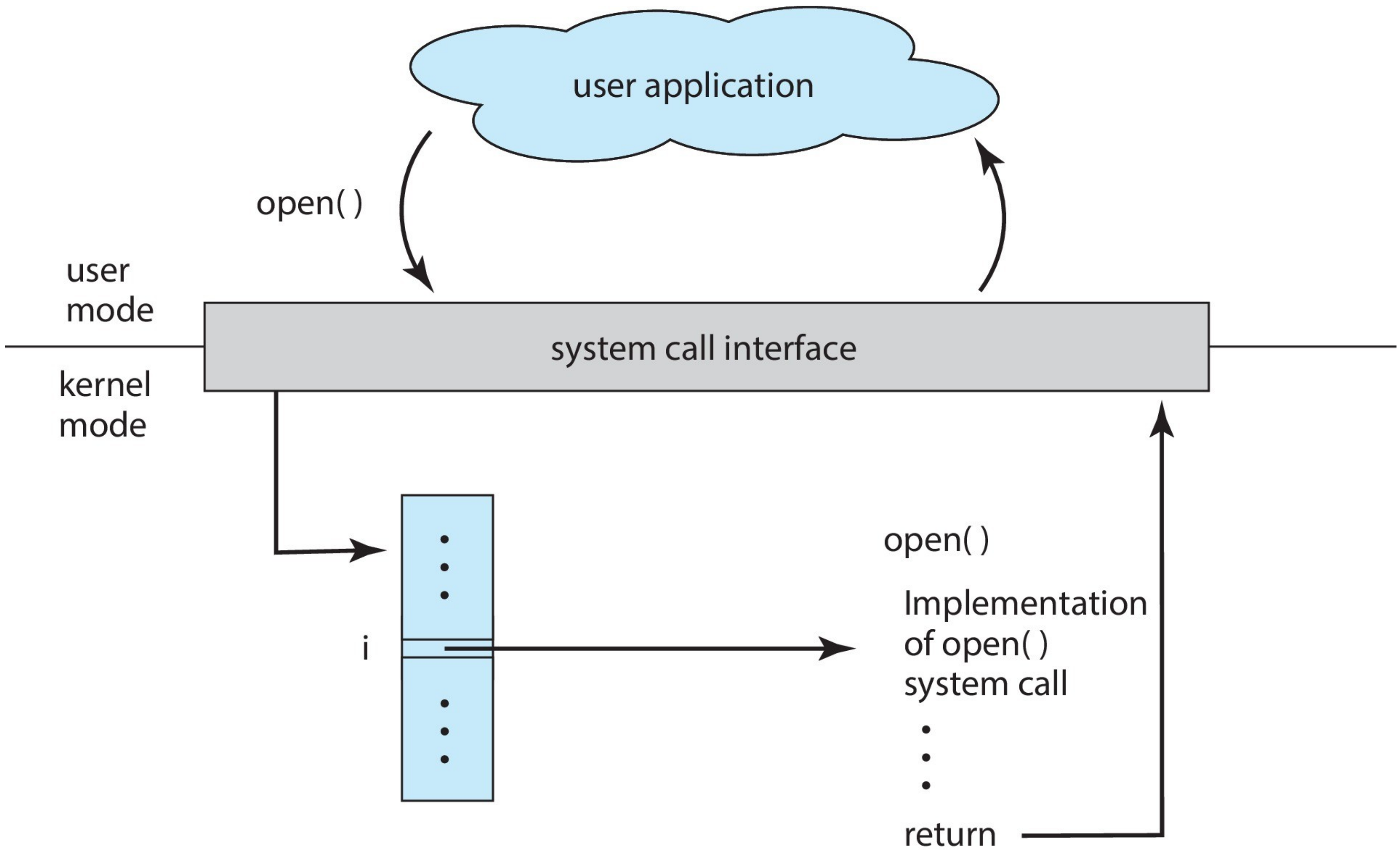
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



API – System Call – OS Relationship



Types of System Calls

- Process control
  - Create, terminate, end, abort, load, execute, ...
  - Debugger for determining bugs, single step execution
  - Locks for managing access to shared data between processes
- File management
  - create file, delete file, open, close, read, write, …
- Device management
  - request, release, read, write, attach or detach devices
- Information maintenance
  - get time or date, set time or date, get system data, …
- Communications
  - create, delete communication connection, …
- Protection
  - Control access to resources, Get and set permissions, Allow and deny user access, ...

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	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()`

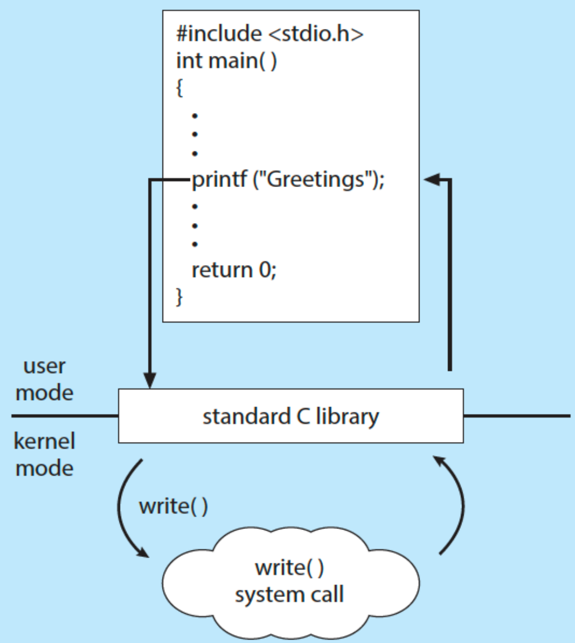
Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Standard C Library Example

- C program invoking printf() library call, which calls write() system call

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Logging -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

System Programs

- Once the kernel is loaded and executing, it can start providing services to the system and its users.
- If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen (a system call from user)
- Some services are provided outside of the kernel by system programs (system services or system utilities) that are loaded into memory at boot time to become system daemons, which run the entire time the kernel is running.
- system utilities provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls.

System programs examples

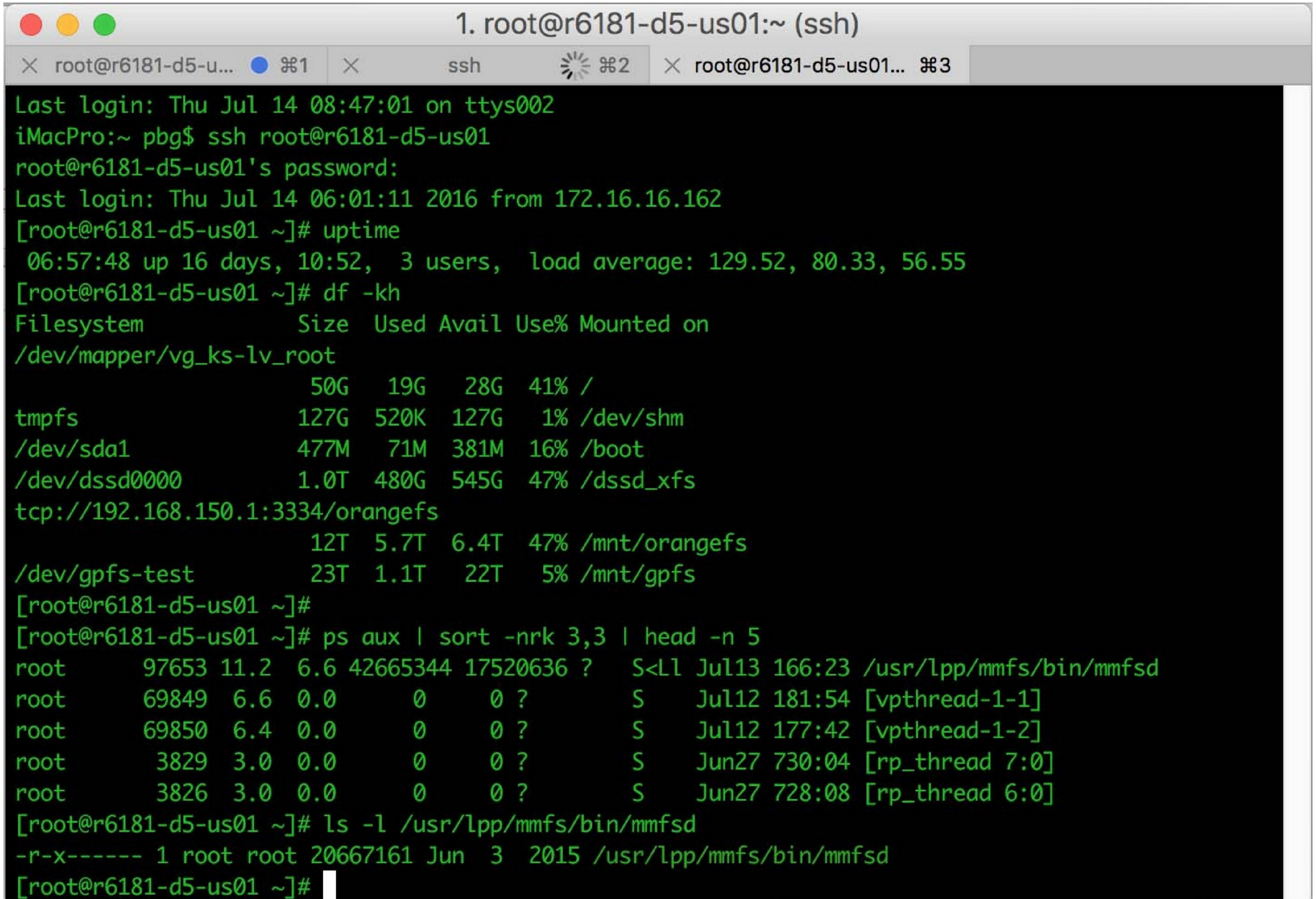
- **Systemd:** the first system program On Linux is“ systemd ,” and it starts many other daemons.
  - List all the services controlled by systemd: systemctl list-unit-files --type service
- **Cron service:** Cron is a system that helps Linux users to schedule any task. However, a cron job is any defined task to run in a given time period
- **Gnome display manager:** a program that manages graphical display servers and handles graphical user logins
- File management Utilities, Text Editors, Program Linkers and loaders

Command Line interpreter

- CLI or **command interpreter** allows direct command entry
- **Function:** Primarily fetches a command from user and executes it
- **Organization:** Sometimes implemented in kernel, sometimes by systems program
- **Multiple flavors** implemented – **shells**
  - Bourne shell, bash, third-party shells
- **Implementation approaches:**
  - commands are built-in
  - Commands are just names of programs
    - Benefit: flexible for adding new features without shell modification

```
echo $SHELL
```

Bourne Shell Command Interpreter



1. root@r6181-d5-us01:~ (ssh)

root@r6181-d5-u... ⌘1 | ssh ⌘2 | root@r6181-d5-us01... ⌘3

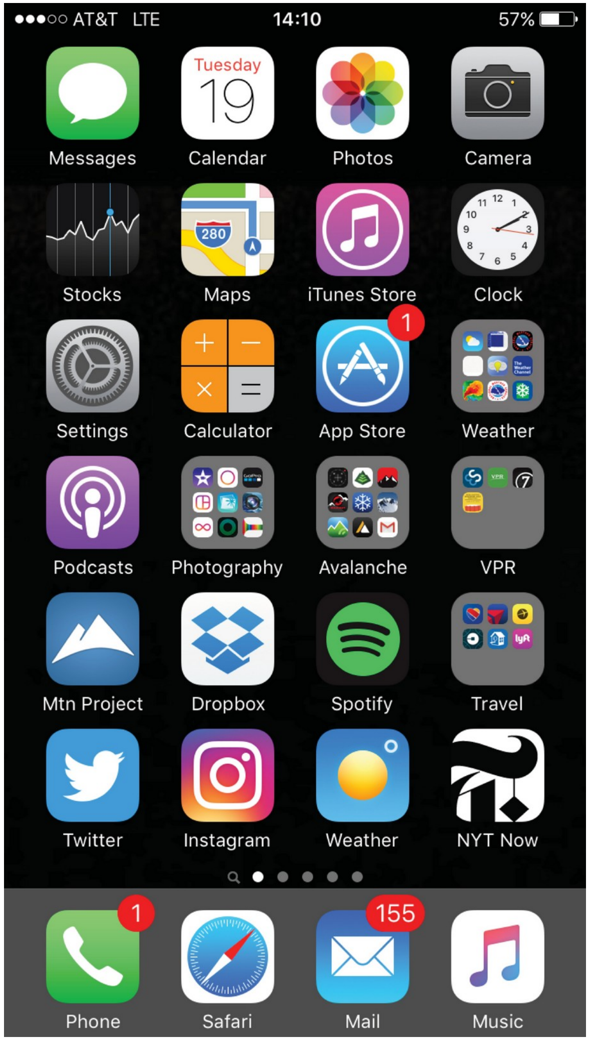
```
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                       50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1             477M   71M  381M  16% /boot
/dev/dssd0000         1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                       12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?   S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0      0     0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0      0     0 ?        S    Jul12 177:42 [vpthread-1-2]
root       3829  3.0  0.0      0     0 ?        S    Jun27 730:04 [rp_thread 7:0]
root       3826  3.0  0.0      0     0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x------ 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

User Operating System Interface - 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
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

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



Quick student research

Please find
system call table in the kernel of your linux
Provide two samples of system call interface code