COMP 4384 Software Security

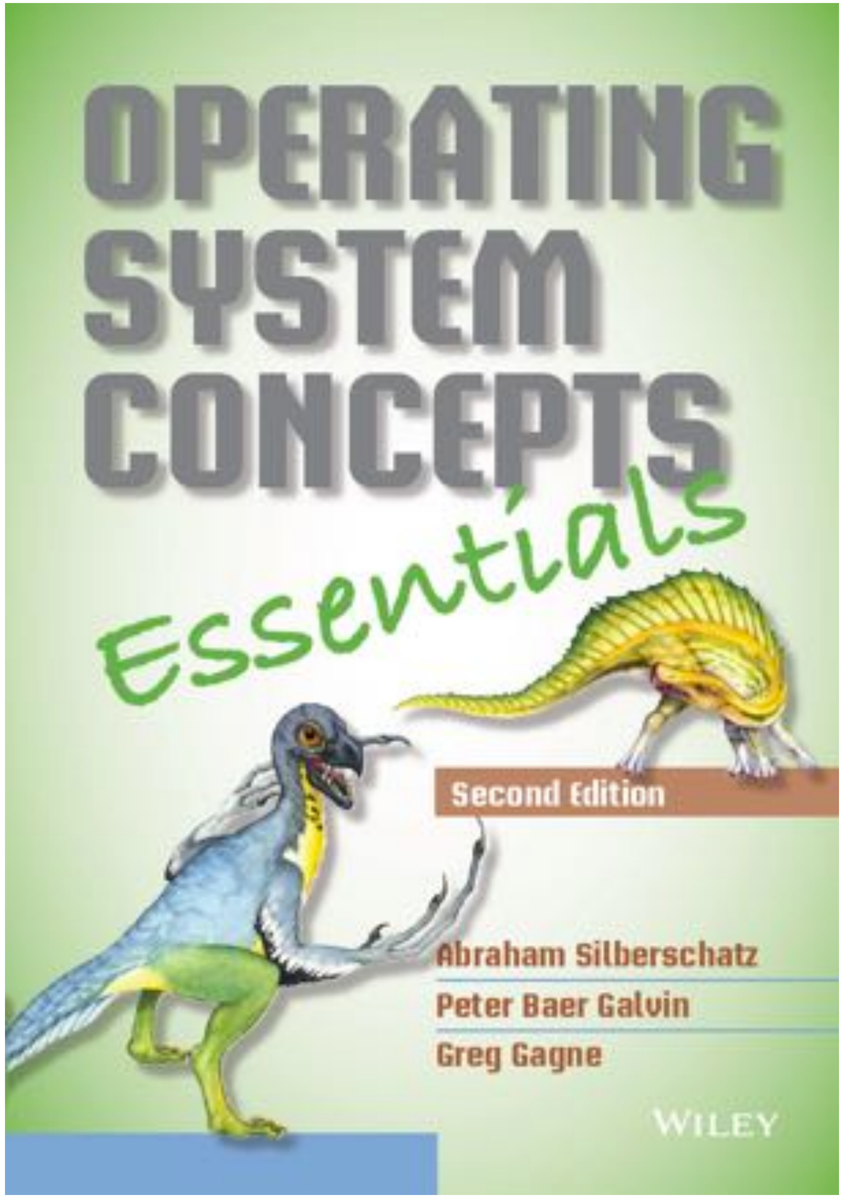
Module 4: *Operating Systems Concepts*

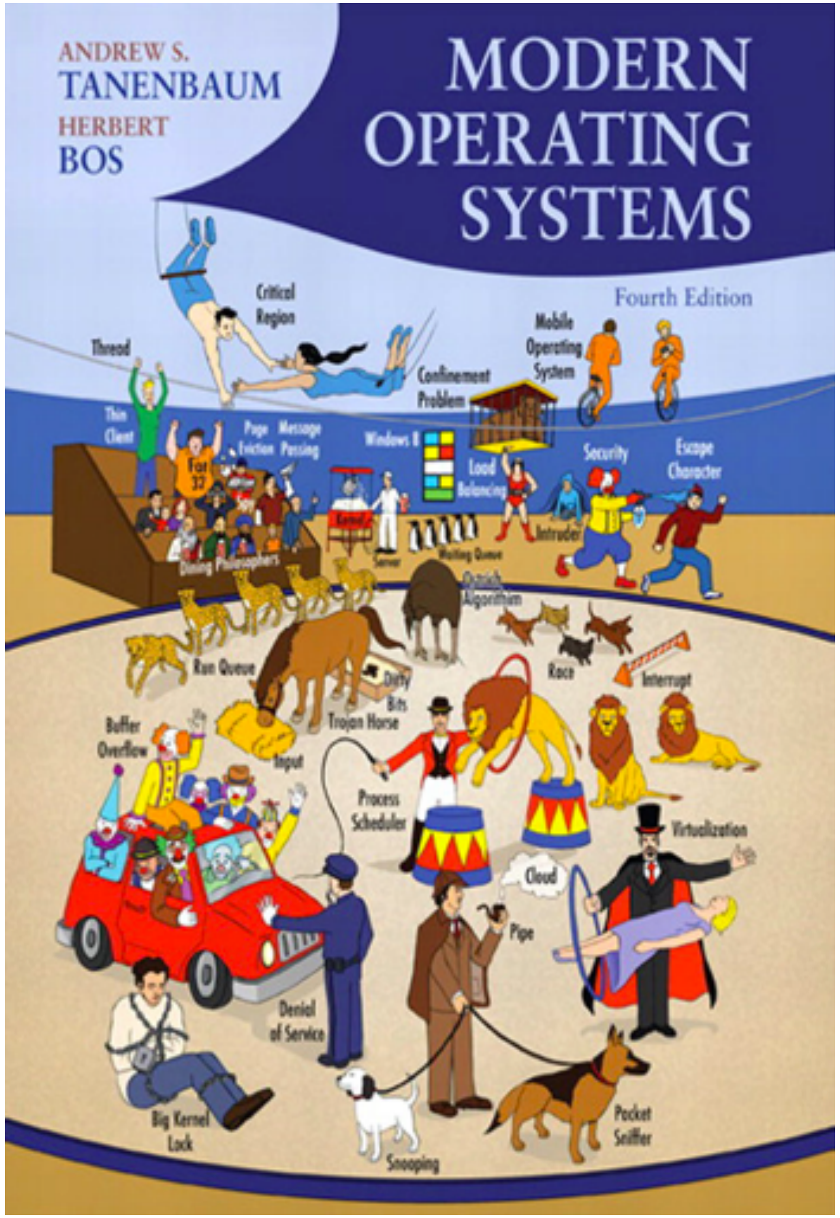
Ahmed Tamrawi

in atamrawi atamrawi.github.io ahmedtamrawi@gmail.com

An **operating system** is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be ***convenient,*** others to be ***efficient,*** and others to be some combination of the two.

A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the **kernel**. (Along with the kernel, there are two other types of programs: **system programs**, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.)





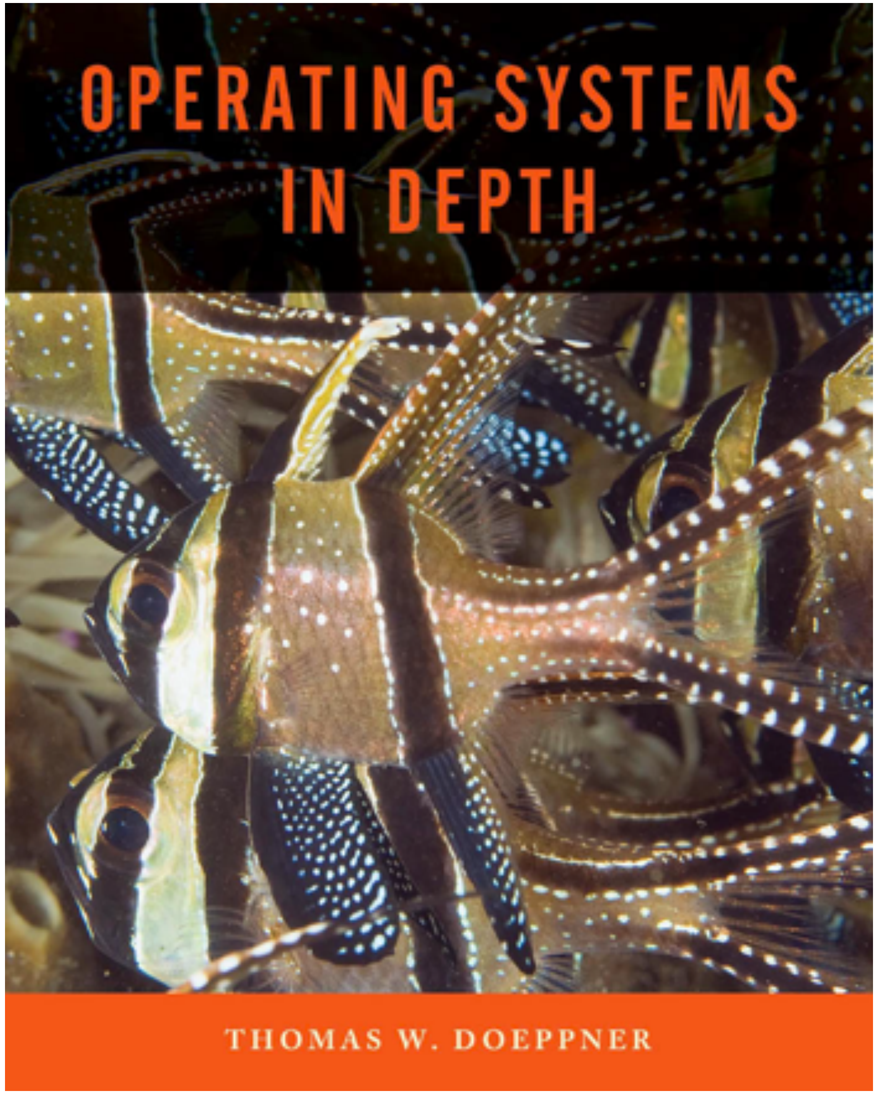
1.1 WHAT IS AN OPERATING SYSTEM?

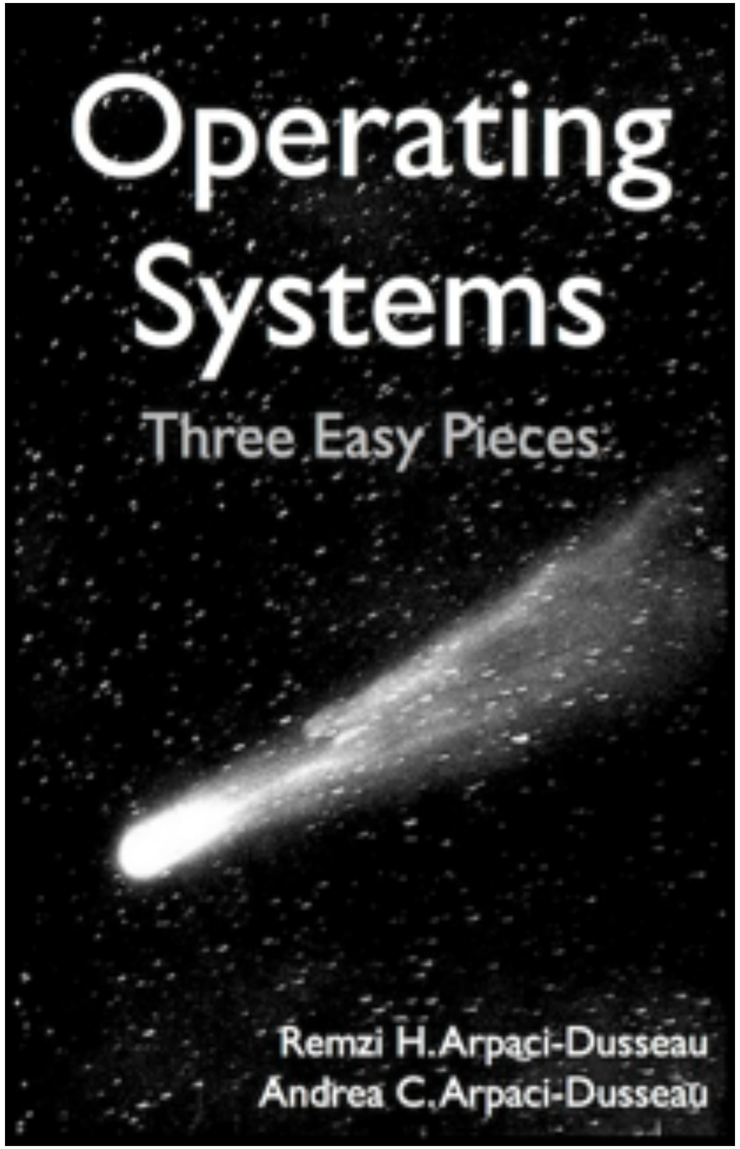
It is hard to pin down what an operating system is other than saying it is the software that runs in kernel mode—and even that is not always true. Part of the problem is that operating systems perform two essentially unrelated functions: providing application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources. Depending on who is doing the talking, you might hear mostly about one function or the other. Let us now look at both.

1.1 OPERATING SYSTEMS

What's an operating system? You might say it's what's between you and the hardware, but that would cover pretty much all software. So let's say it's the software that sits between your software and the hardware. But does that mean that the library you picked up from some web site is part of the operating system? We probably want our operating-system definition to be a bit less inclusive. So, let's say that it's that software that almost everything else depends upon. This is still vague, but then the term is used in a rather nebulous manner throughout the industry.

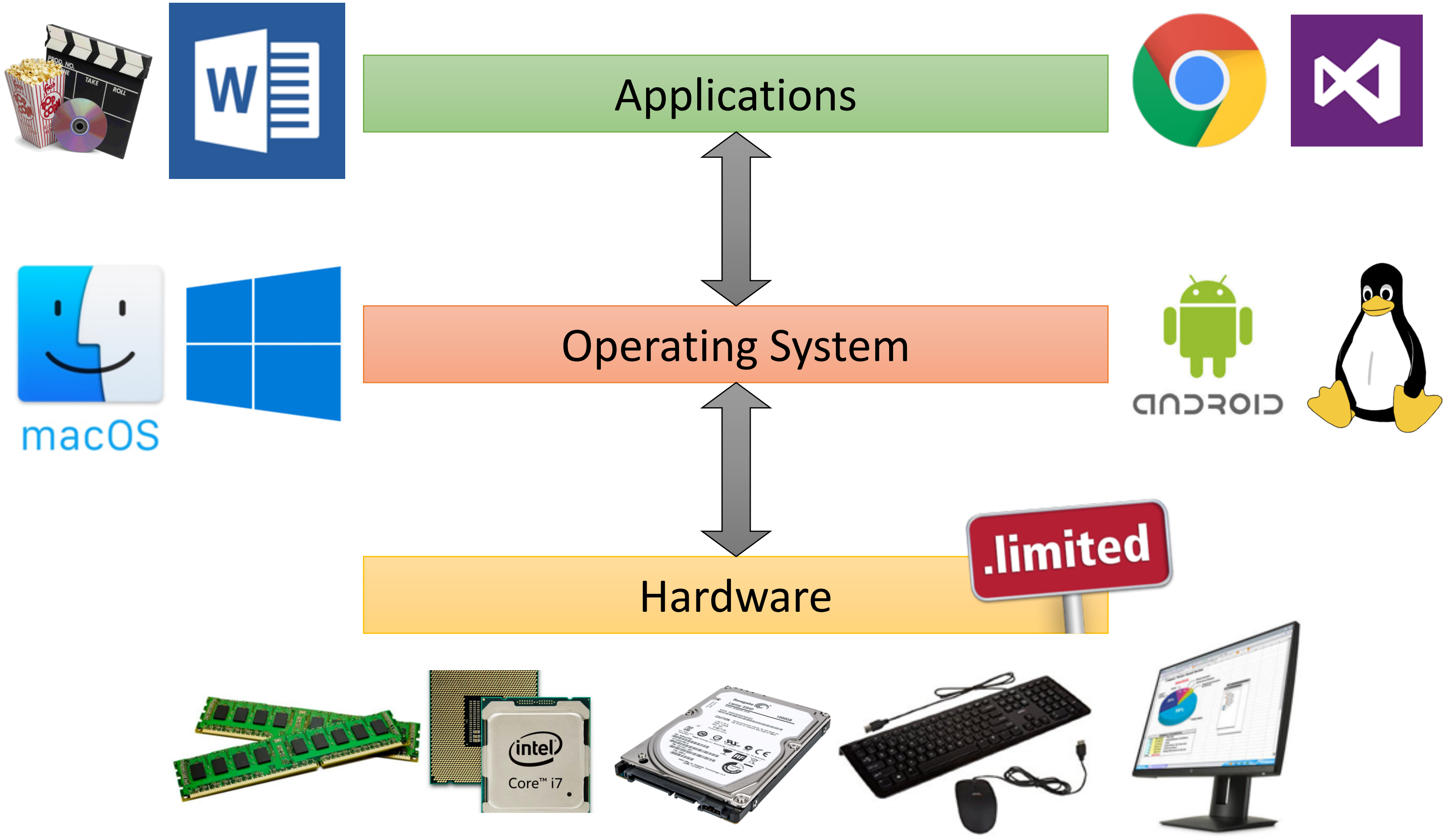
Perhaps we can do better by describing what an operating system is actually supposed to do. From a programmer's point of view, operating systems provide useful abstractions of the underlying hardware facilities. Since many programs can use these facilities at once, the operating system is also responsible for managing how these facilities are shared.





There is a body of software, in fact, that is responsible for making it easy to run programs (even allowing you to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. That body of software is called the **operating system** (**OS**)[3], as it is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner.

Realistic View of Operating System



Our Definition

An **operating system** is a program that
manages resources and ***provide abstractions***

Main Ideas in OS

Manage Resources

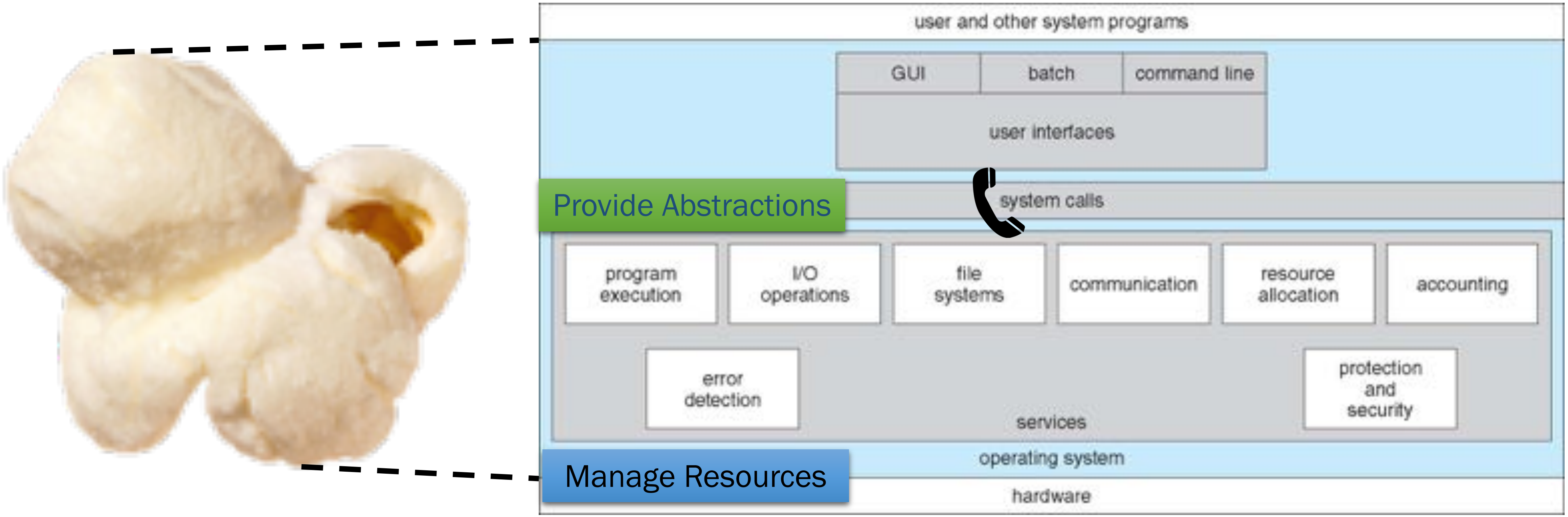
How do you *share* **processors**, **memory**, and **hardware devices** among programs?

Provide Abstractions

How do you provide programs with **clean** and **easy to use** interfaces to resources, without sacrificing (too much) **efficiency and flexibility?**

A View of Operating System Services

Operating systems provide an environment for execution of programs and services to programs and users



Does it have an Operating System?

Introduction

- An operating system (OS) **provides the interface** between the users of a computer and that computer's hardware.
- In particular, an operating system **manages** the ways applications access the **resources** in a computer, including its disk drives, CPU, main memory, input devices, output devices, and network interfaces.
- It is the "glue" that allows users and applications to **interact** with the hardware of a computer.

Introduction

- Operating systems allow application developers to write programs without having to handle low-level details (**provide abstractions**) such as how to deal with every possible hardware device, like the hundreds of different kinds of printers that a user could possibly connect to his or her computer.
- Operating systems handle a staggering number of complex tasks, many of which are directly related to *fundamental security problems*.
  - For example, operating systems must allow for multiple users with potentially different levels of access to the same computer.

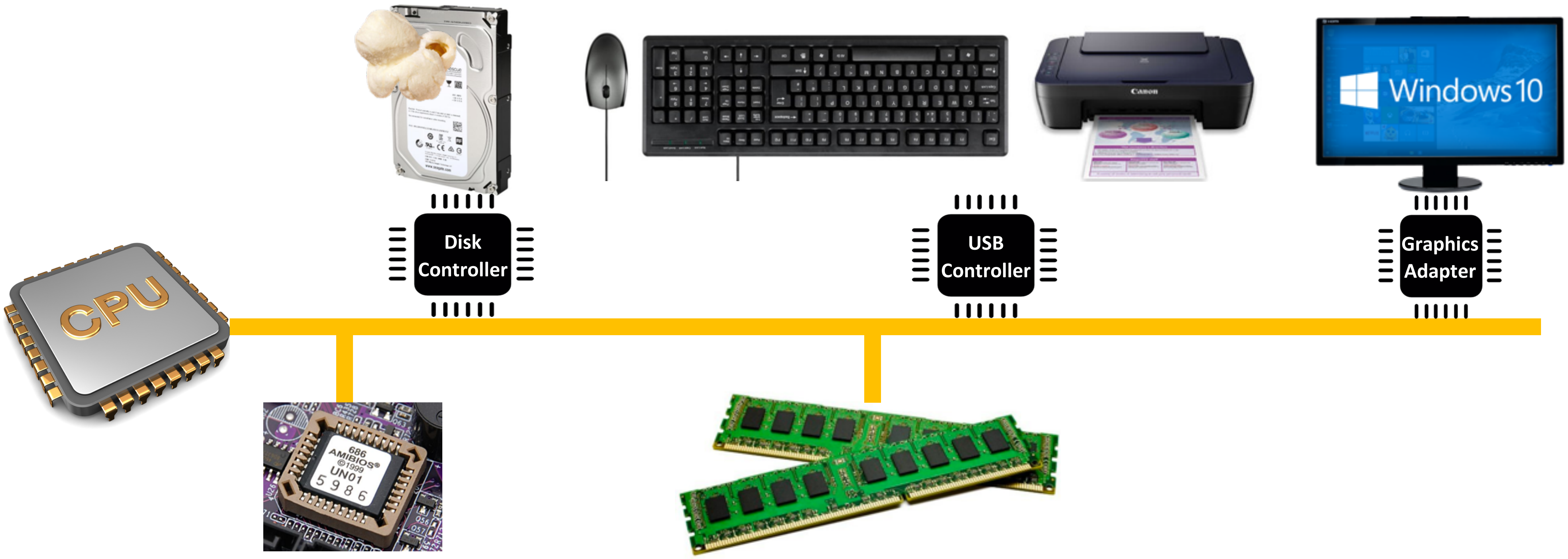
Introduction: *A University Lab*

- A university lab typically allows multiple users to access computer resources, with some of these users, for instance, being students, some being faculty, and some being administrators that maintain these computers.
- Each different type of user has potentially unique needs and rights with respect to computational resources, and it is the operating system’s job to make sure these rights and needs are respected while also avoiding malicious activities.

Introduction: *Multitasking*

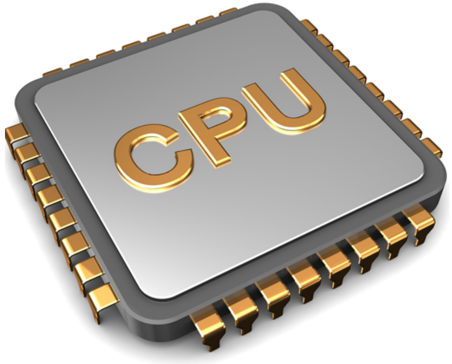
- In addition to allowing for multiple users, operating systems also allow multiple application programs to run at the same time, which is a concept known as **multitasking**.
- This technique is extremely useful; however, this ability has an implied security need of protecting each running application from interference by other, potentially malicious, applications.
- Applications running on the same computer, even if not running simultaneously might have access to **shared resources**, like the filesystem.
- Thus, the operating system should have **measures** in place so that applications can’t **maliciously or mistakenly damage resources** needed by other applications.

Our Computer System



What happens at Computer Startup?

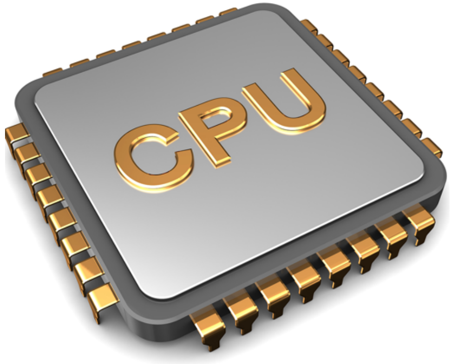

4GIFs.com



Finds itself in **Real Mode**

Power-On Self-Test

Executes the code at address 0xFFFF0 which corresponds to **BIOS**

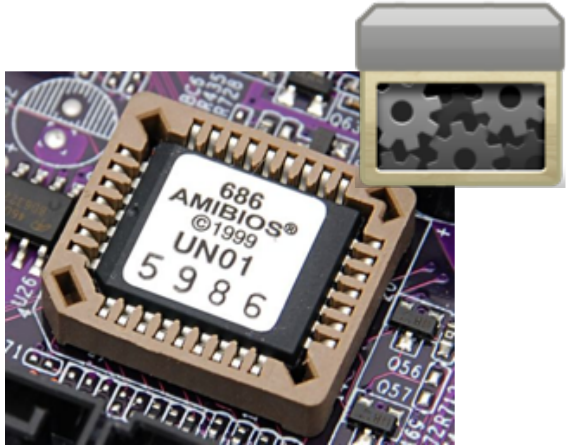


Finds itself in **Real Mode**

Power-On Self-Test

Executes the code at address 0xFFFF0 which corresponds to **BIOS**

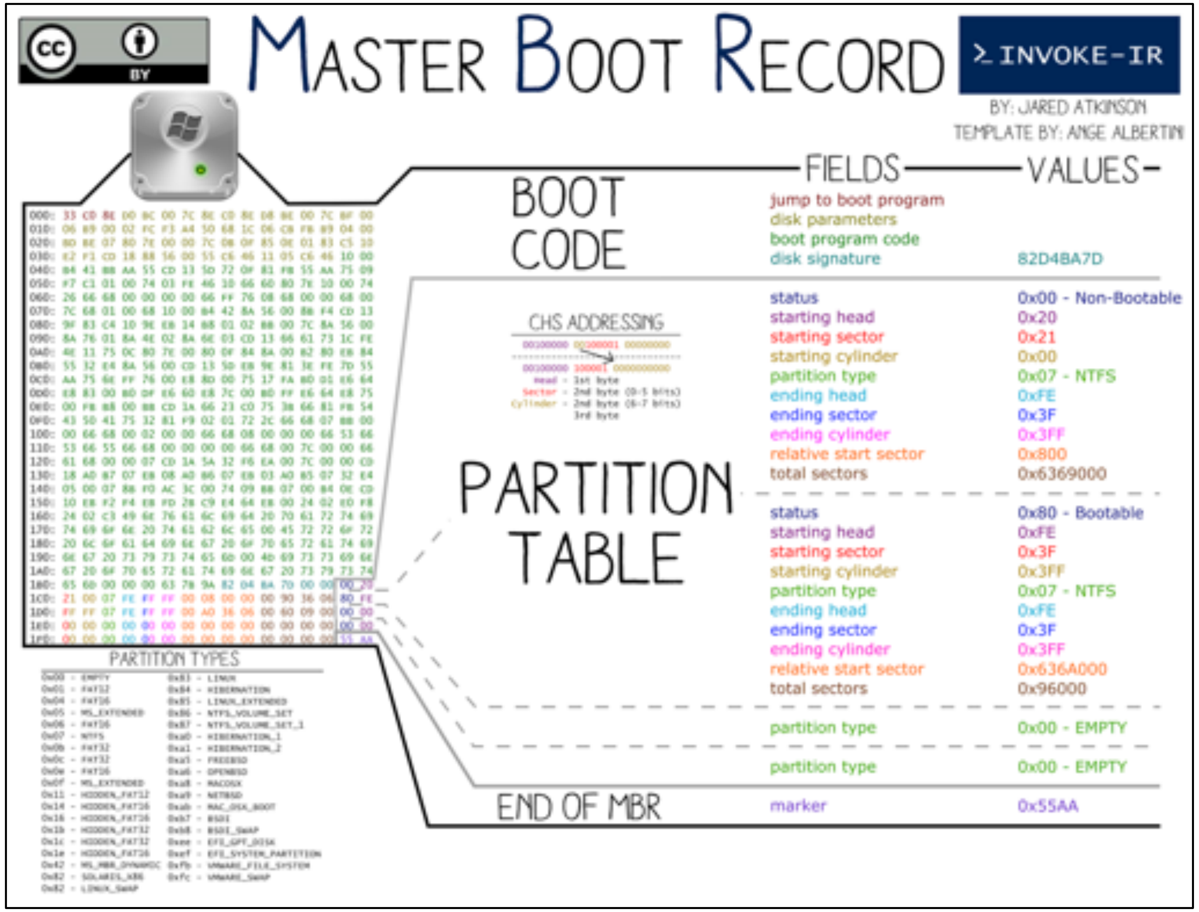
Bootstrap Program

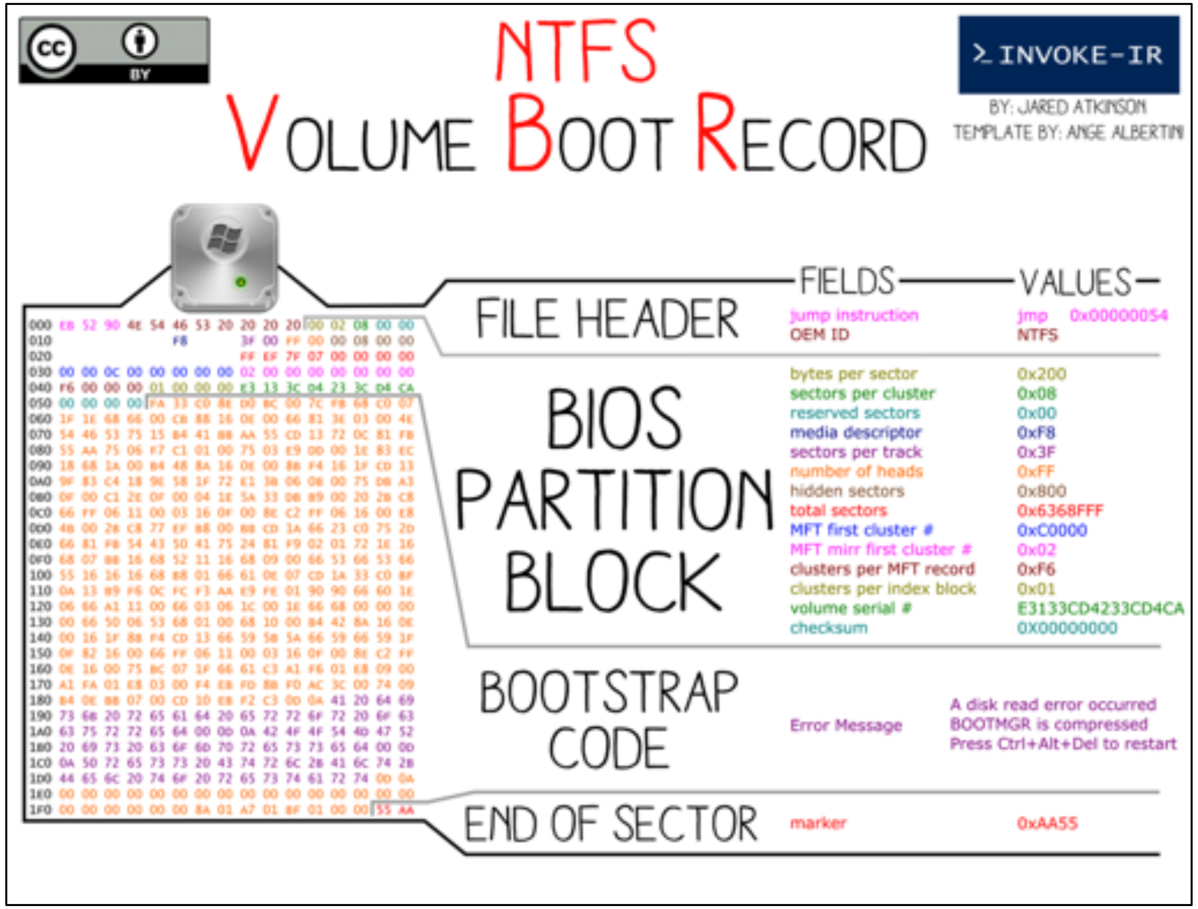


Autoprobing I/O ports

Looks for **bootloader** in Boot Device

It loads the first sector of a bootable device at 0x7C00 and jumps to it. Then it executes the MBR bootloader located in the first sector of a bootable disk (/dev/hda or /dev/sda)

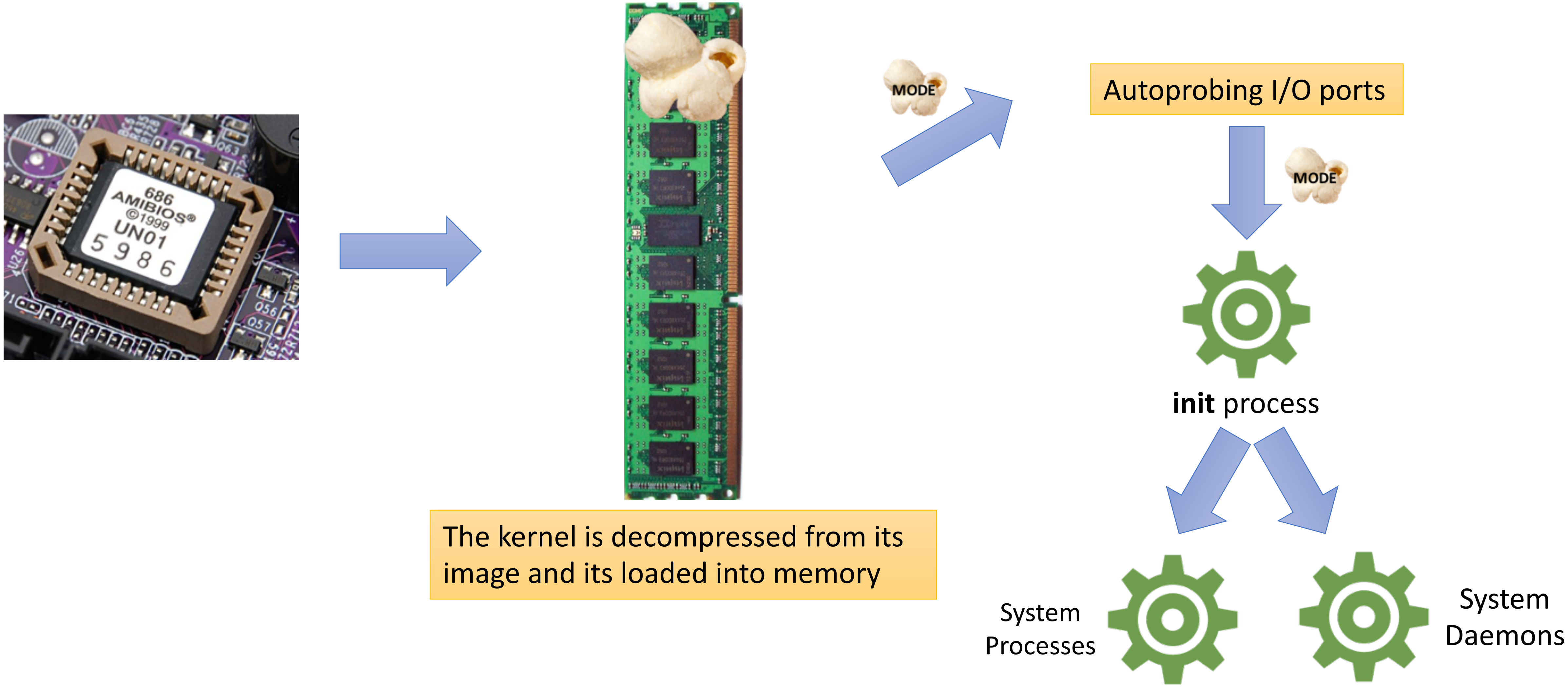


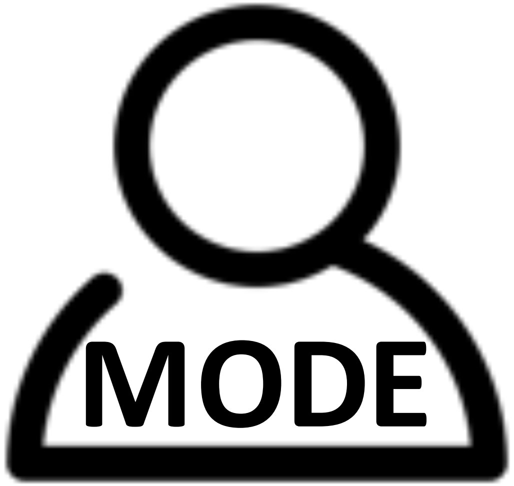


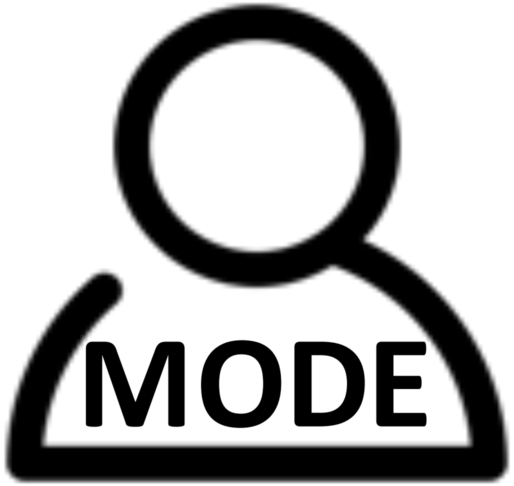
http://www.invoke-ir.com/2015/05/ontheforensictrail-part2.html

Any program to run **must** be loaded in memory

The kernel is decompressed from its image and its loaded into memory


686
AMIBIOS®
©1999
UN01
5986
The kernel is decompressed from its image and its loaded into memory
MODE
Autoprobing I/O ports
MODE
init process
System Processes
System Daemons


MODE



Wait for Event to Occur

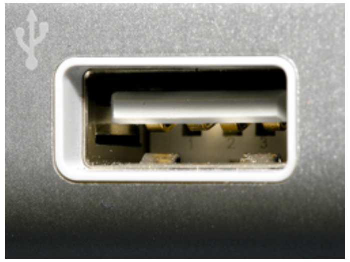

انا مش فاهم حاجة خالص

What happens when you move the cursor?

Mouse sends out pulses, one pulse for every 1000th of an inch or so

What happens when you move the cursor?

Mouse sends out pulses, one pulse for every 1000th of an inch or so

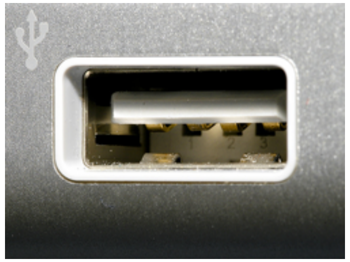


The pulses are received through a USB packet or through an old serial line

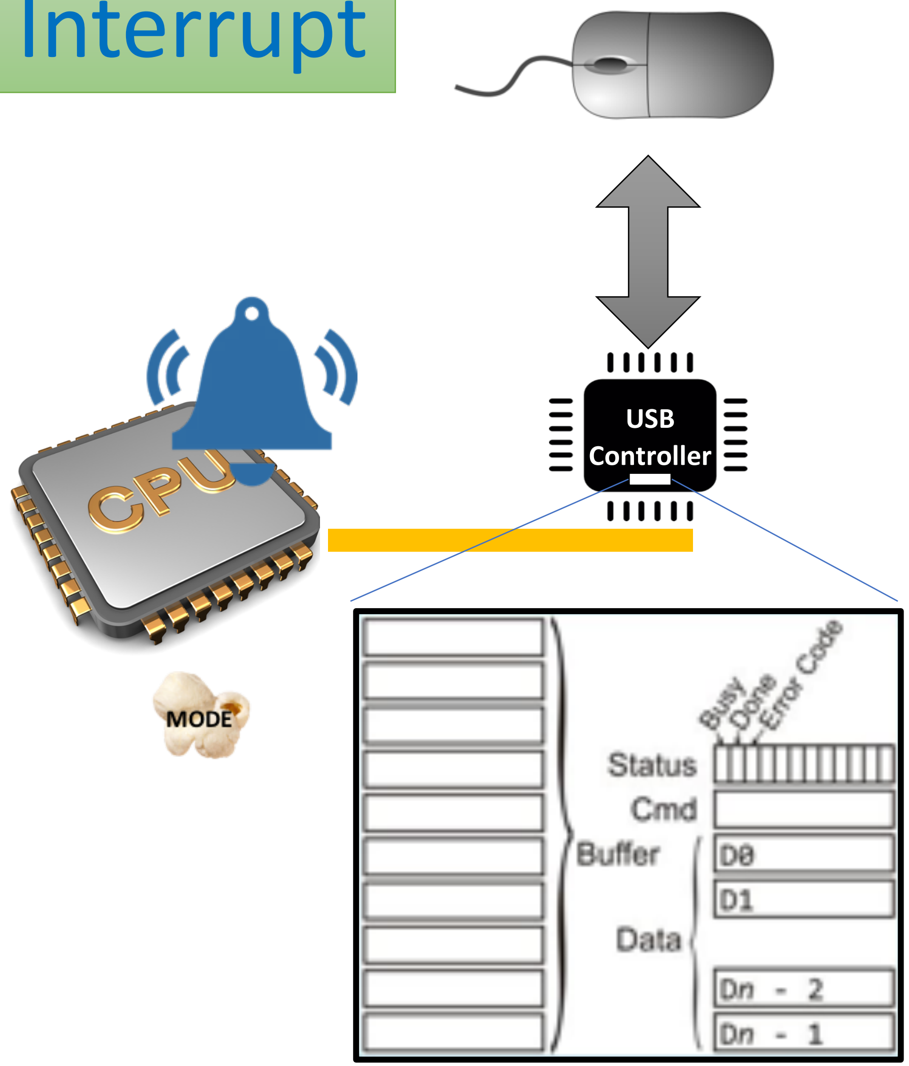
What happens when you move the cursor?

Hardware Interrupt

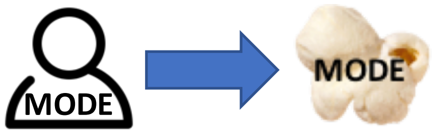
Mouse sends out pulses, one pulse for every 1000th of an inch or so

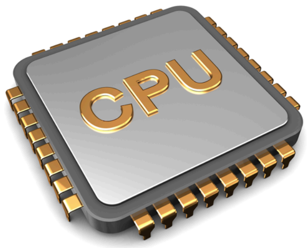


The pulses are received through a USB packet or through an old serial line



What happens when CPU is interrupted?

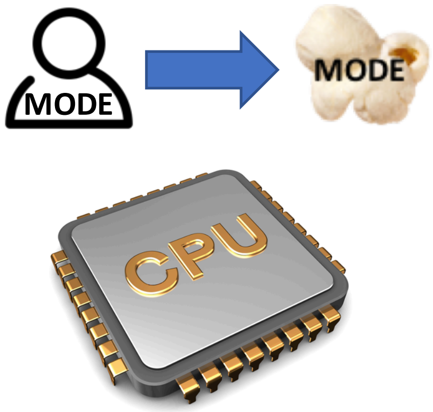


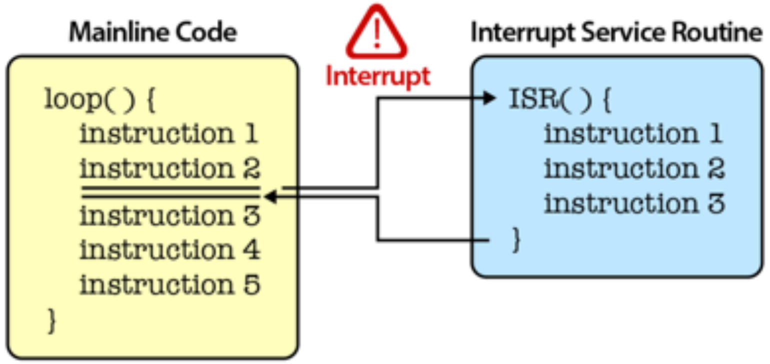


CPU preserves the current state of the CPU by storing registers and the program counter

Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**

What happens when CPU is interrupted?





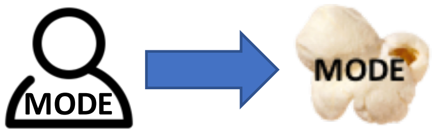
CPU preserves the current state of the CPU by storing registers and the program counter

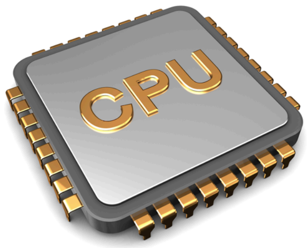
Separate segments of code determine what action should be taken for each type of interrupt

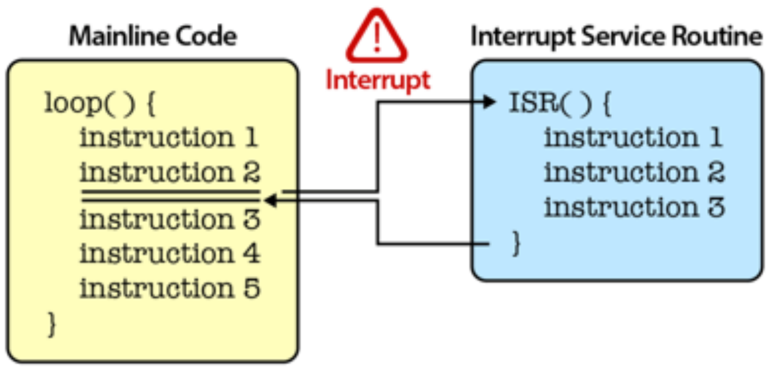
Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**

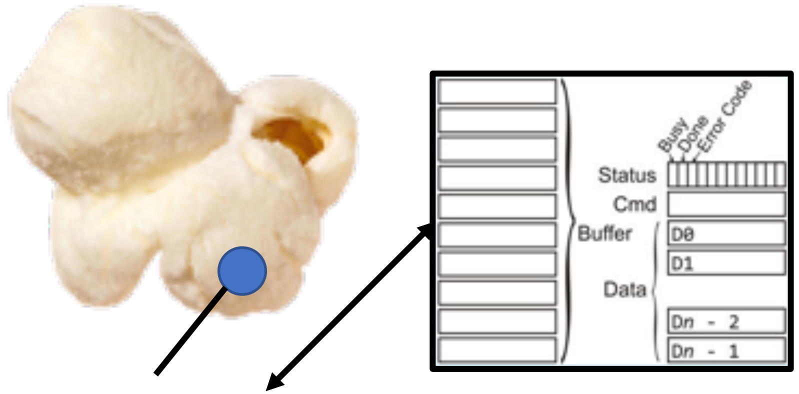
Reads the interrupt and realizes it's from the mouse, and calls the proper ISR which calls the mouse driver.

What happens when CPU is interrupted?









Mouse Driver

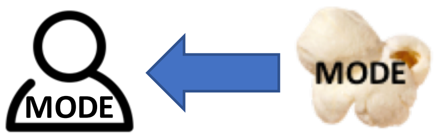
CPU preserves the current state of the CPU by storing registers and the program counter

Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**

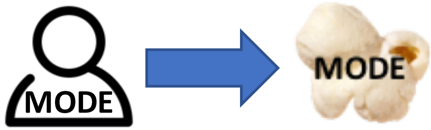
Separate segments of code determine what action should be taken for each type of interrupt

Reads the interrupt and realizes it’s from the mouse, and calls the proper ISR which calls the mouse driver.

Mouse driver adds the x and y increments to its current cursor position and return the result to OS

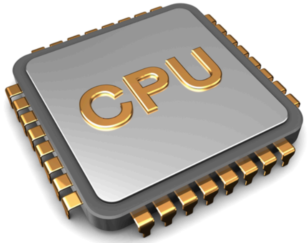


How to notify Monitor of cursor movement?

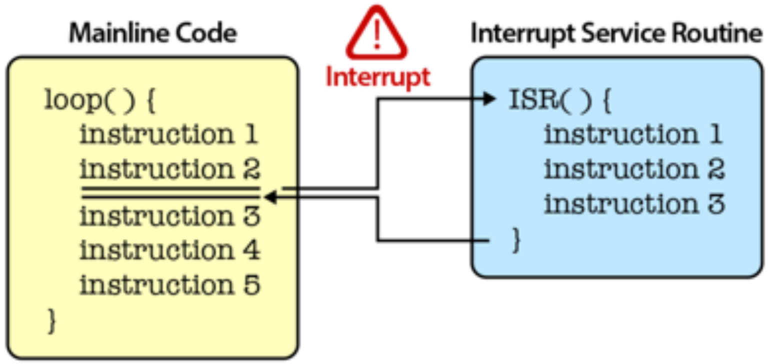


Software Interrupt (Trap)

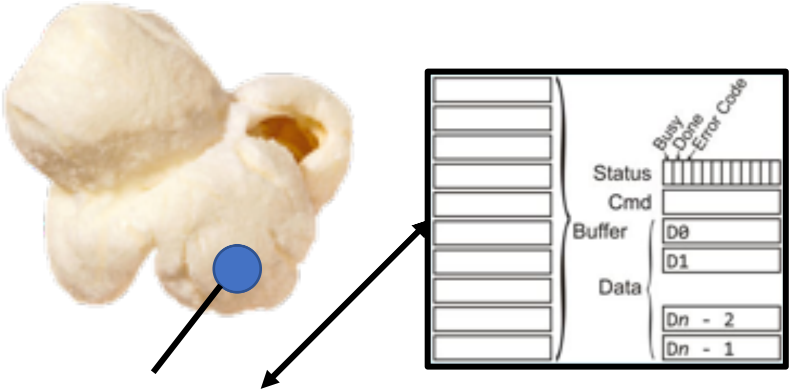
OS gets interrupted through a **system call** to update the screen



OS preserves the current state of the CPU by storing registers and the program counter

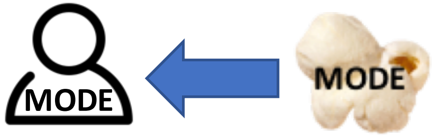


Reads the interrupt and realizes it's from OS to monitor. It calls the display driver with the updated screen



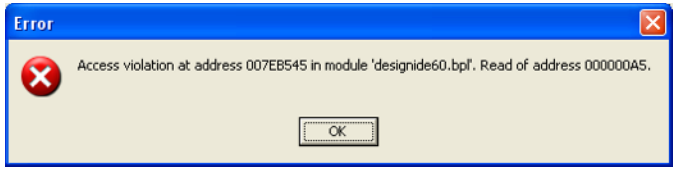
Display Driver

Monitor device drivers sets the proper registers and buffer data in the graphics adapter



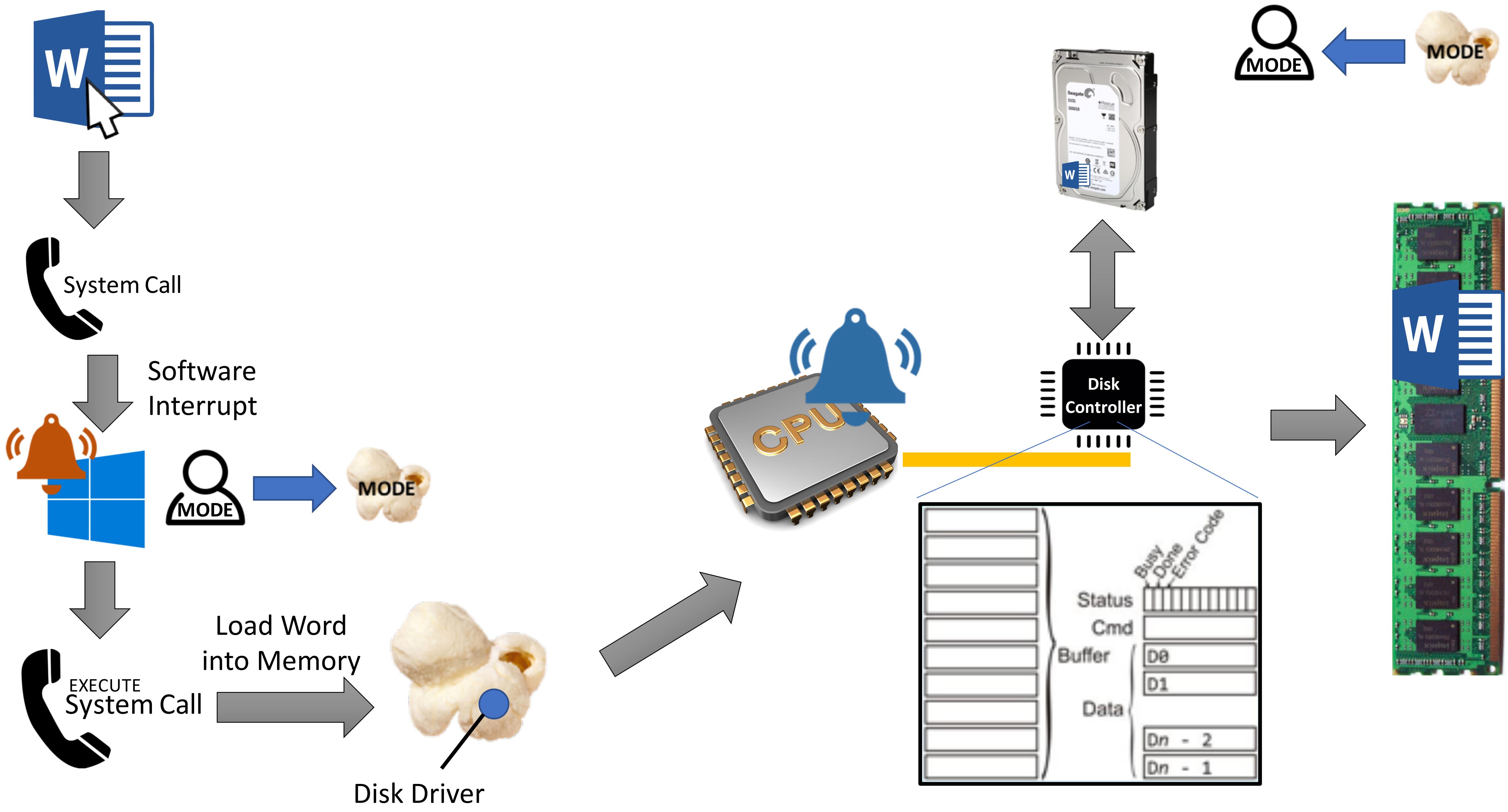
Software Interrupt (Trap)

EXECUTE
System Call



MODE

Any program to run **must** be loaded in memory



System Call
Software
Interrupt
MODE
MODE
EXECUTE
System Call
Load Word
into Memory
Disk Driver
CPU
Disk
Controller
Status
Cmd
Buffer
Data
Busy
Done
Error Code
D0
D1
Dn - 2
Dn - 1
MODE
MODE

An operating system is interrupt driven


انا مش فاهم حاجة خالص

As long as their processes fit in memory, we do not have a memory problem

Each process needs resources to accomplish its task: CPU, memory, I/O, files, etc.

Process termination requires reclaim of any reusable resources

Typically system has many processes running concurrently, how this is achieved?

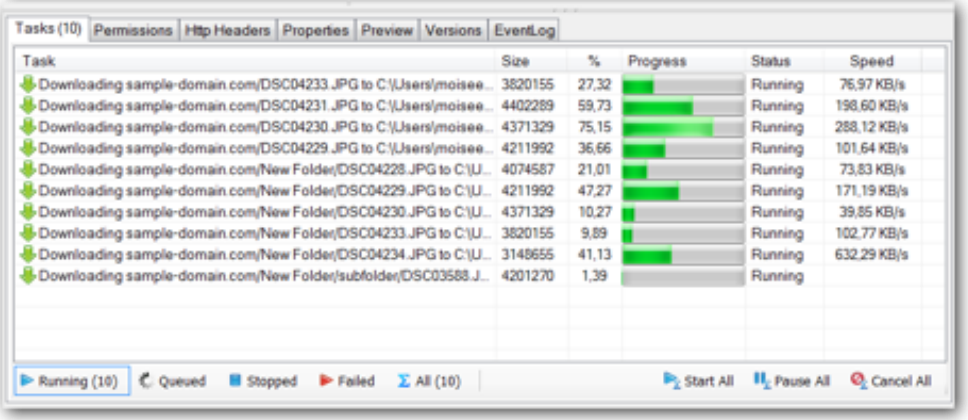
Many Processes

Creating/deleting user and system processes

Suspending/resuming processes

Process Synchronization & Communication

Process Management



The memory is not enough memory for all my processes!

Memory is not Enough

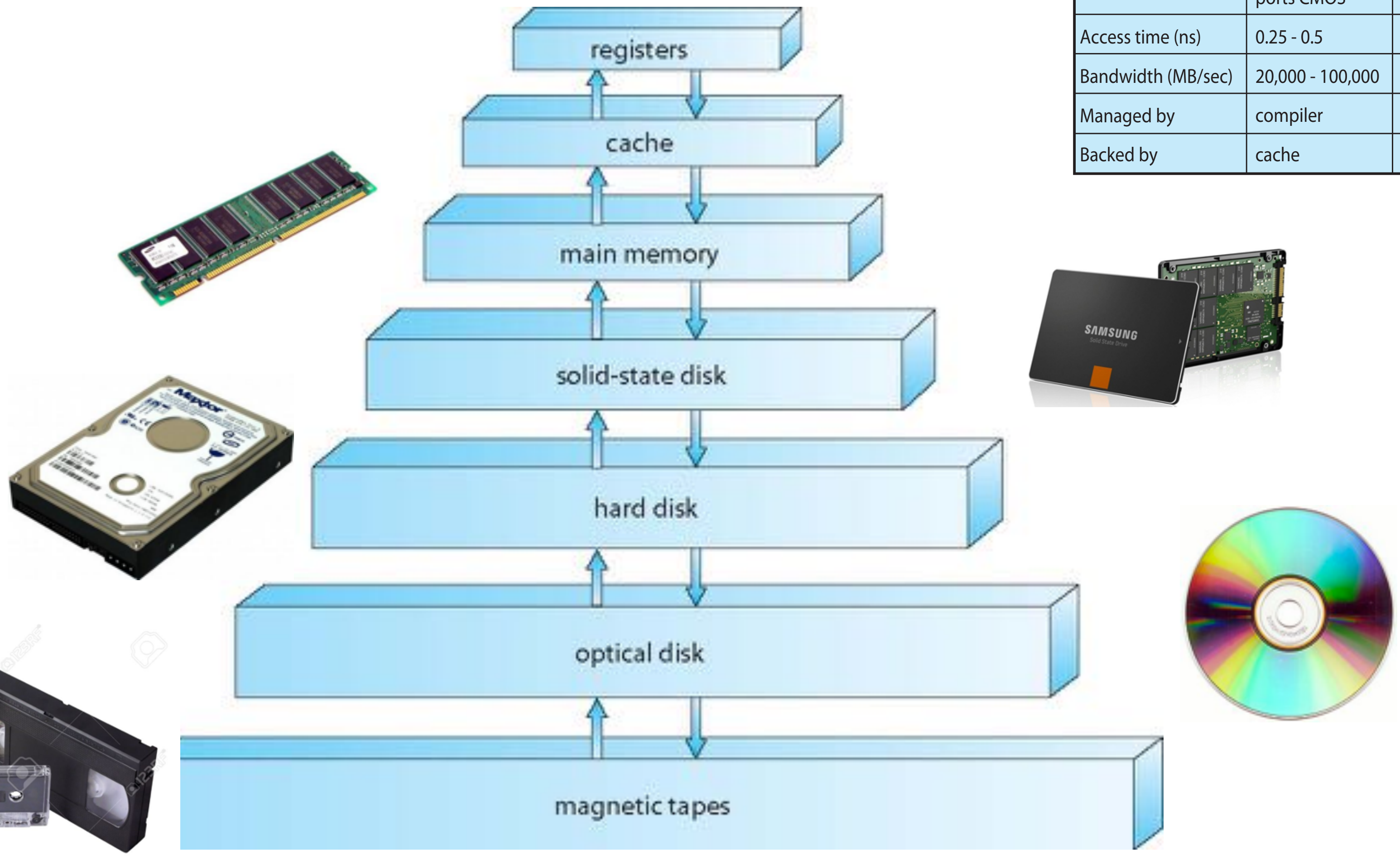
Keeping track of which parts of memory are currently being used and by whom

Deciding which processes and data to move into and out of memory

Allocating and deallocating memory space as needed

Memory Management

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape



Different Kinds of Storage Devices

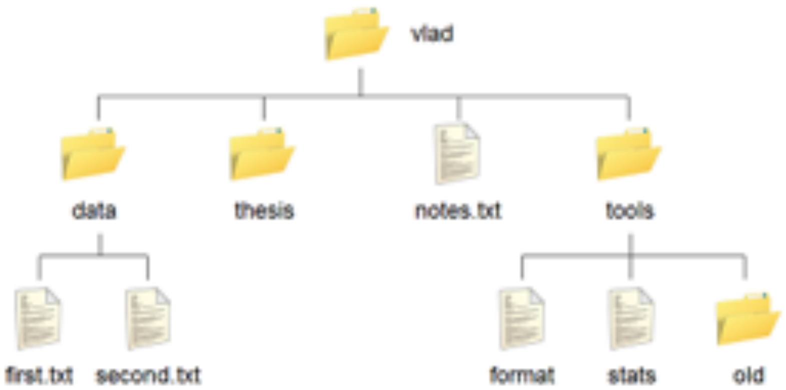
Usually disks is used to store data that does not fit in main memory or data that must be kept for a "long" period of time

Entire speed of computer operation hinges on disk subsystem and its algorithms

Free-space management, Storage Allocation, and Disk Scheduling

Mass-Storage Management

OS provides uniform, logical view of information storage



Abstracts physical properties to logical storage unit : files, directories

Bits, Bytes, and Files

Access control to determine who can access what

Creating and deleting files and directories

Mapping and Backing files onto secondary storage

File-System Management

Many I/O Devices

Hides peculiarities of hardware devices from the user

Memory management of I/O including buffering, caching, spooling

General device-driver interface

I/O Management

Protection – any mechanism for controlling access of processes or users to resources defined by the OS

Security – defense of the system against internal and external attacks including: denial-of-service, worms, viruses, identity theft, theft of service

Protection & Security

An operating system is interrupt driven


System Call

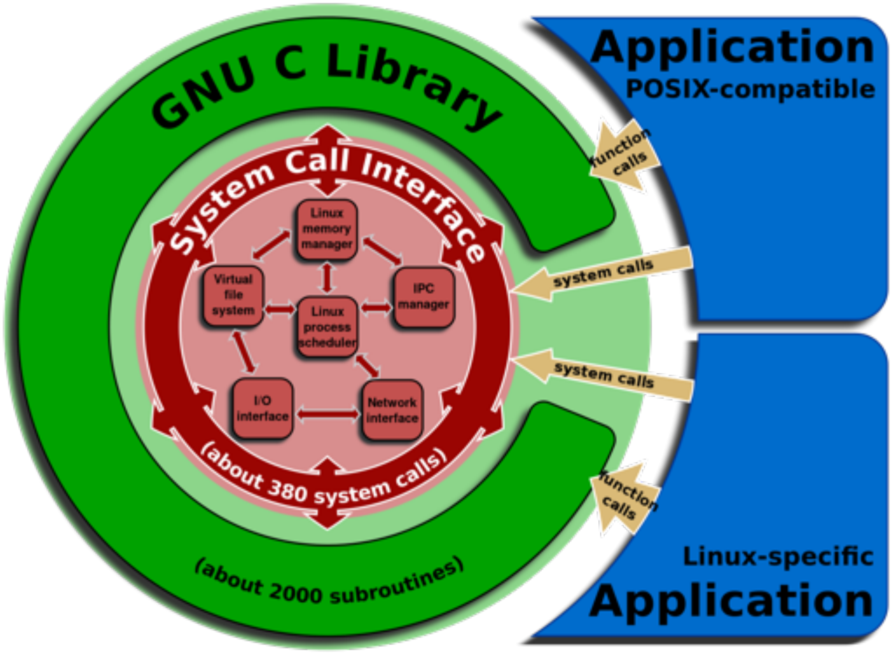

HELLO

Software Interrupt (Trap)

Programming interface to the services provided by the OS

Typically written in a high-level language (C or C++)

Accessible via a high-level **Application Programming Interface (API)** rather than direct system call use



Create, Delete Communication Connection
Message Passing Model Host/Process Name
Shared-Memory Model
Transfer Status Information
Attach/Detach Remote Devices

Create/Terminate/Load/Execute Process
Get/Set Process Attributes
Wait for Time/Event
wait event, signal event
Allocate/Free/Dump Memory
Locks for Process Synchronization

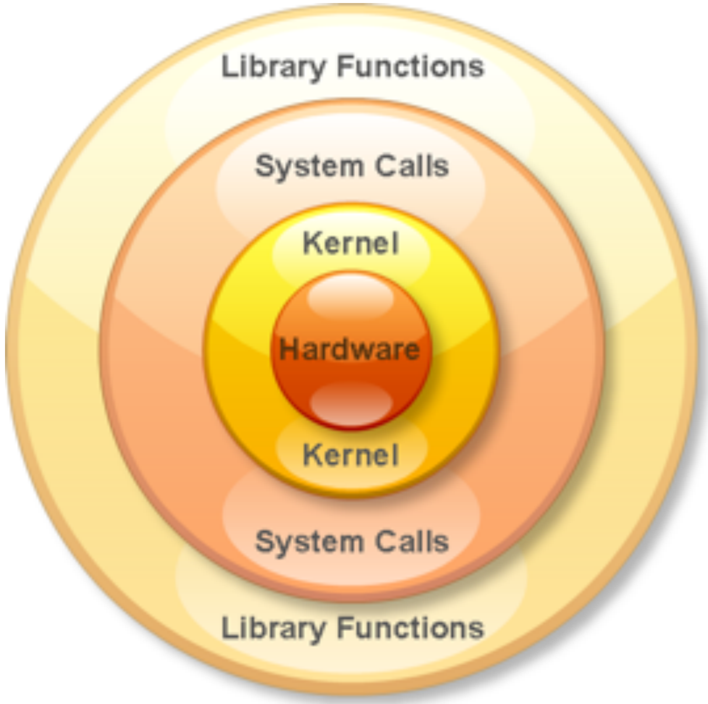
Control access to resources
Get and set permissions
Allow and deny user access

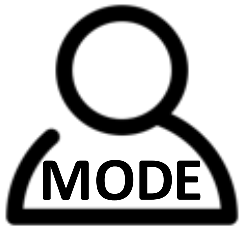
Create/Delete/Open/Close/Read/Write File
Get/Set File Attributes

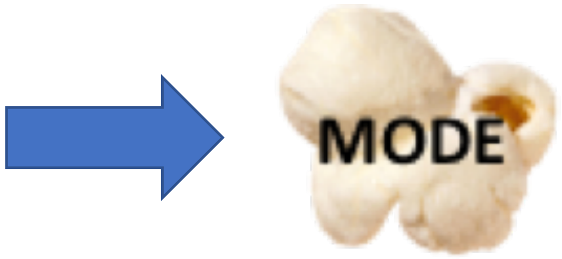
Get/Set Time or Date
Get/Set System Data

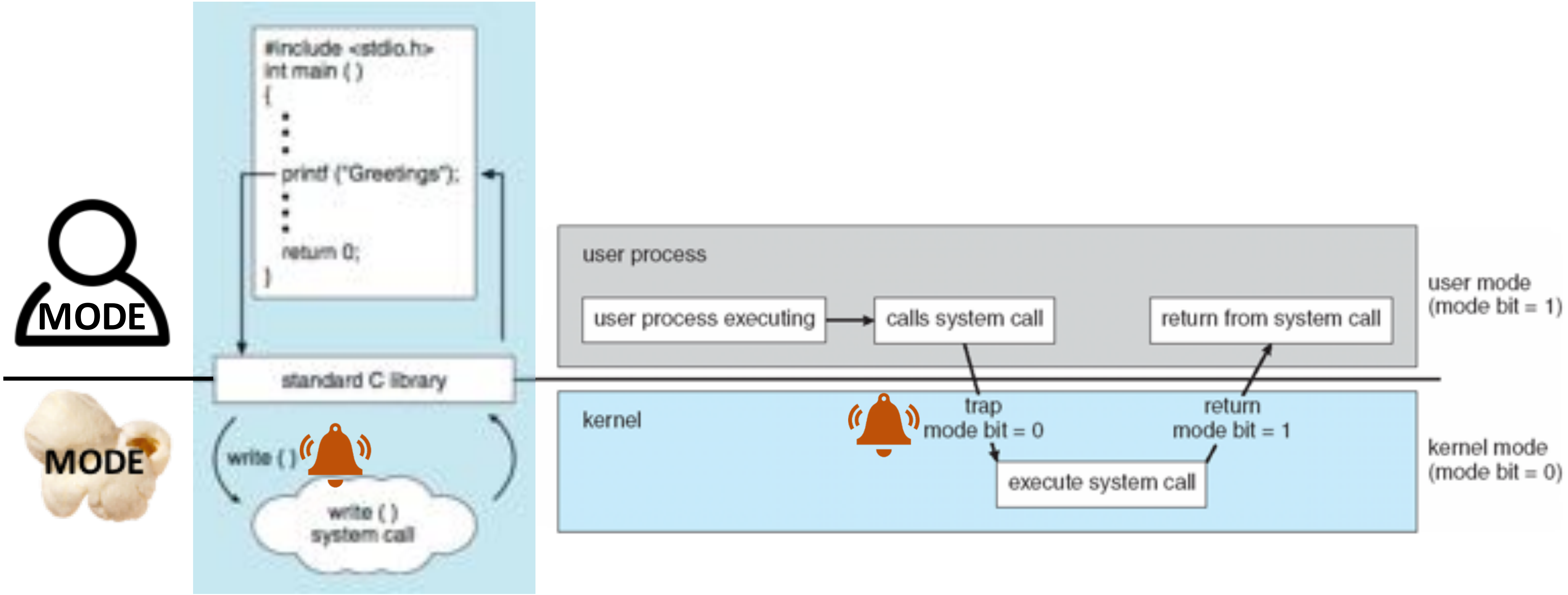
Request/Release/Read/Write Device
Get/Set Device Attributes
Logically Attach/Detach devices

User processes **cannot** perform *privileged operations* themselves





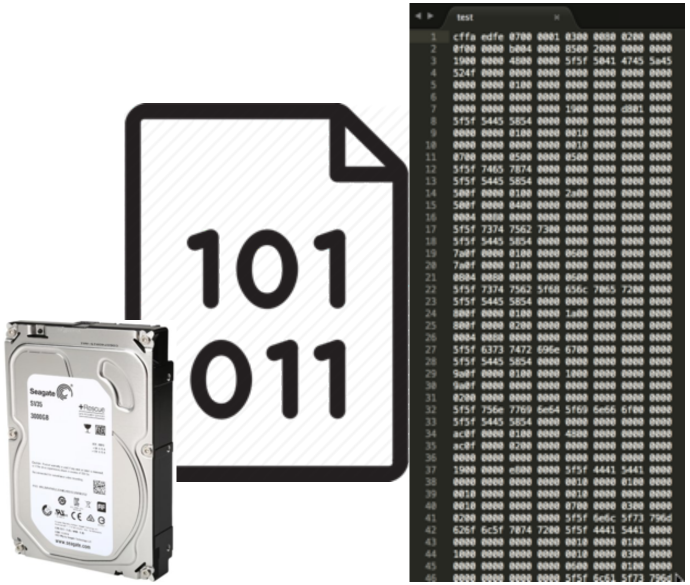


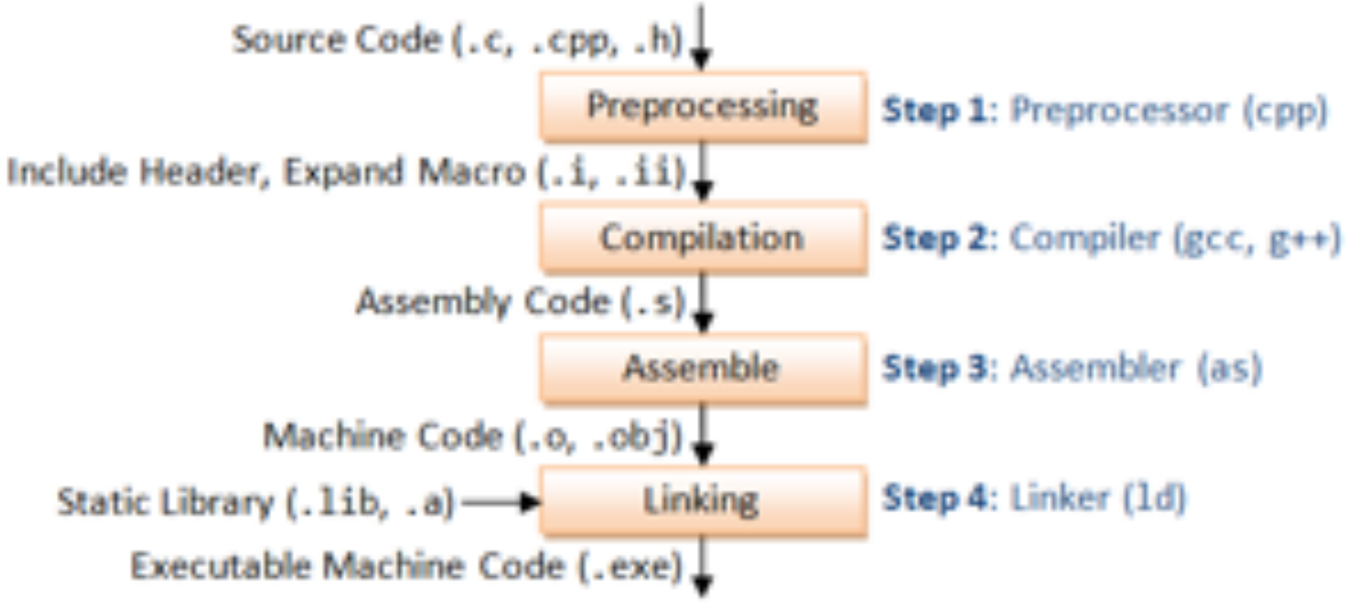

MODE
MODE
#include <stdio.h>
int main ( )
{
printf ("Greetings");
return 0;
}
standard C library
write ( )
write ( )
system call
user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
kernel
trap
mode bit = 0
return
mode bit = 1
execute system call
kernel mode
(mode bit = 0)

Any program to run **must** be loaded in memory

```c
// File: test.c
#include <stdio.h>

int main() {
    printf("I love Mansaf!\n");
    return 0;
}
```

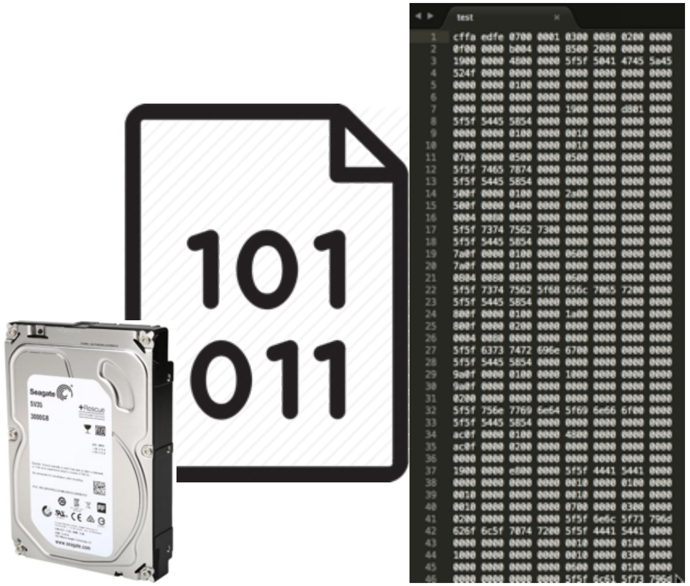
```
gcc -o test test.c
```





```
// File: test.c
#include <stdio.h>

int main() {
    printf(“I love Mansaf!\n”);
    return 0;
}
```

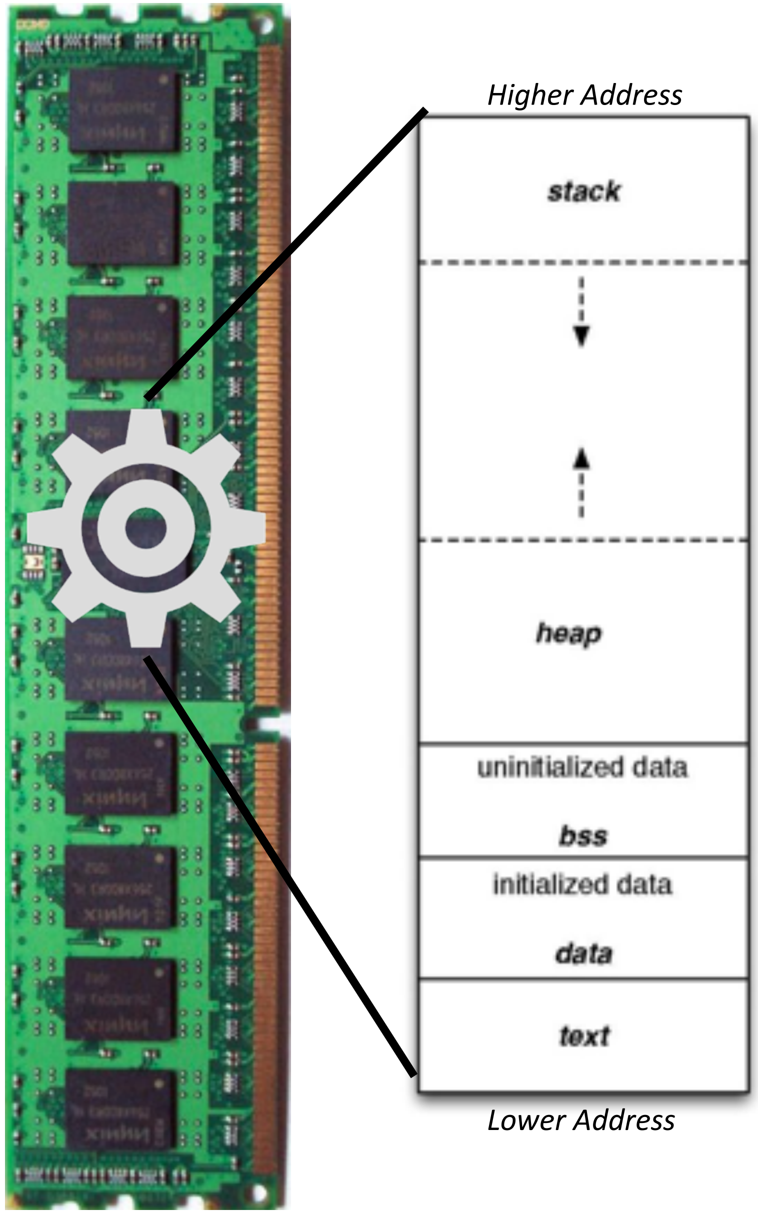
```
gcc –o test test.c
```



Program becomes process when executable file loaded into memory

Process Memory Layout

https://en.wikipedia.org/wiki/Data_segment



Stack Area contains the program stack, a LIFO structure. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The **stack area** contains temporary data: function parameters, return addresses, and local variables.

Heap Area is the memory that is dynamically allocated during process run time. The heap area is managed by `malloc`, `calloc`, `realloc`, and `free`, which may use the `brk` and `sbrk` system calls to adjust its size

BSS Data Segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

Initialized Data Segment contains any global or static variables which have a pre-defined value and can be modified

Text (Code) Segment is one of the sections of a program in an object file or in memory, which contains executable instructions

Process execution *must* progress in ***sequential*** fashion

```c
#include <stdio.h>

int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        248         8          1216       4c0    memory-layout
```

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        248         12         1220       4c4    memory-layout
```

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        248         16         1224       4c8    memory-layout
```

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

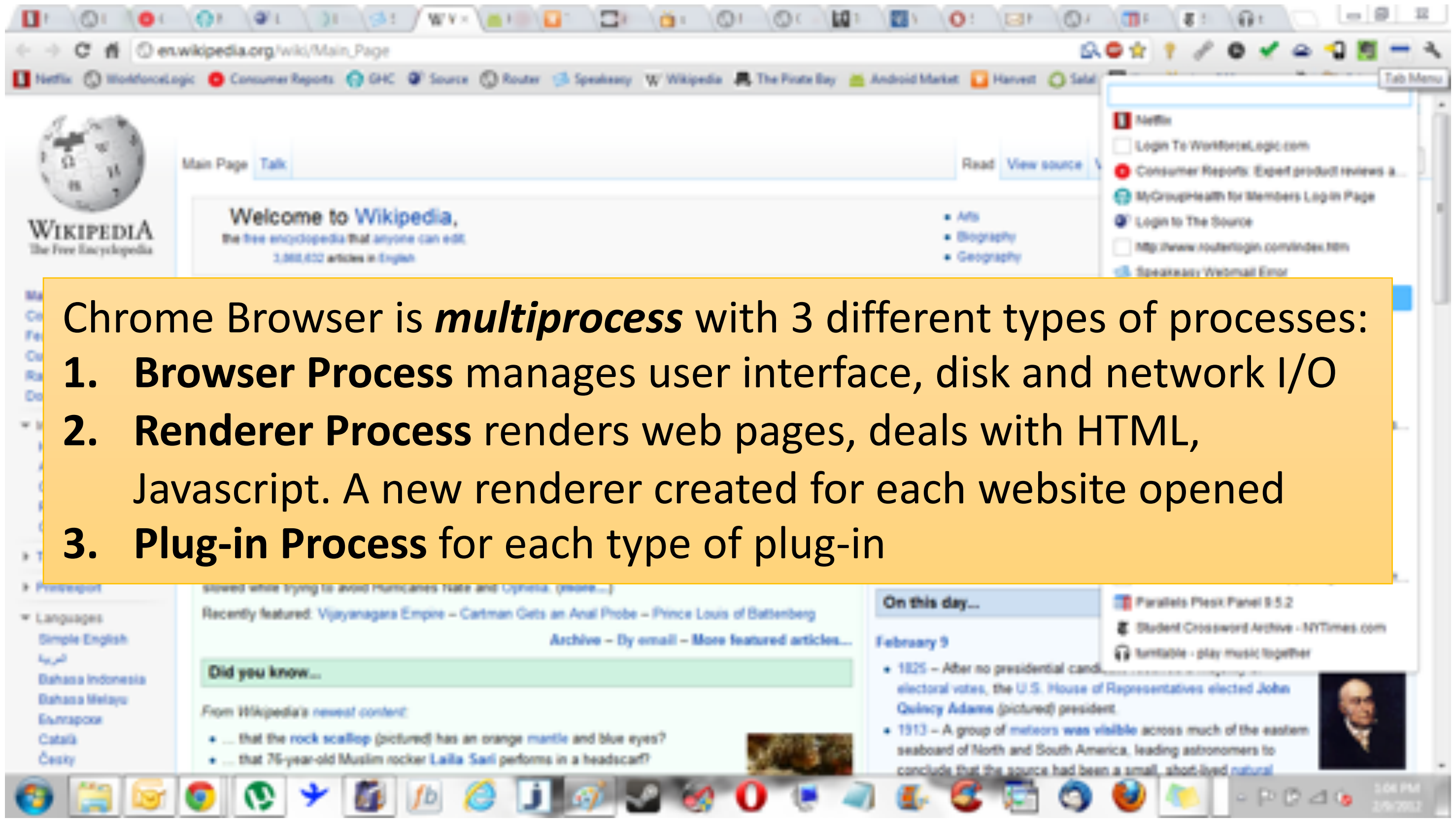
```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        252         12         1224       4c8    memory-layout
```

```c
#include <stdio.h>

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        256         8          1224       4c8    memory-layout
```

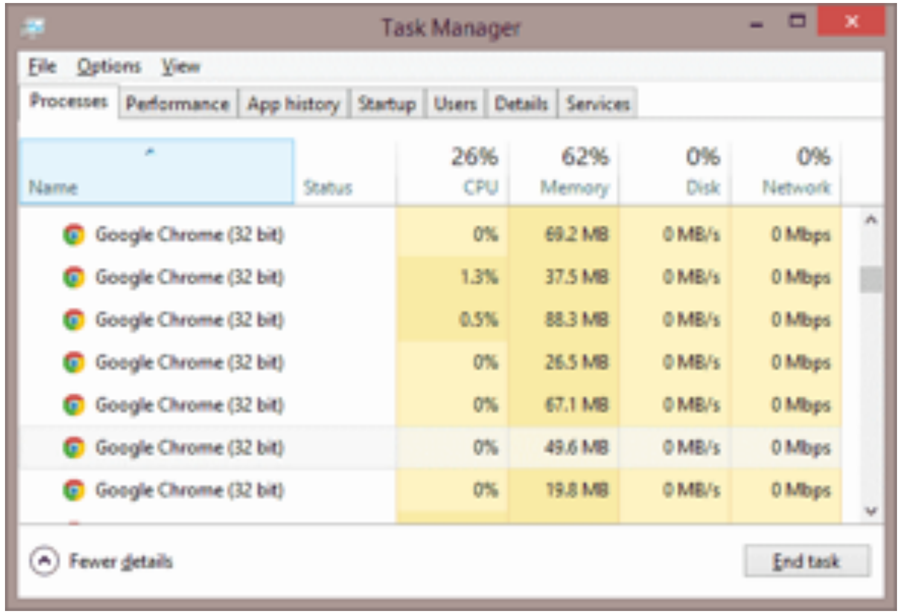
https://www.geeksforgeeks.org/memory-layout-of-c-program/

One program can be several processes

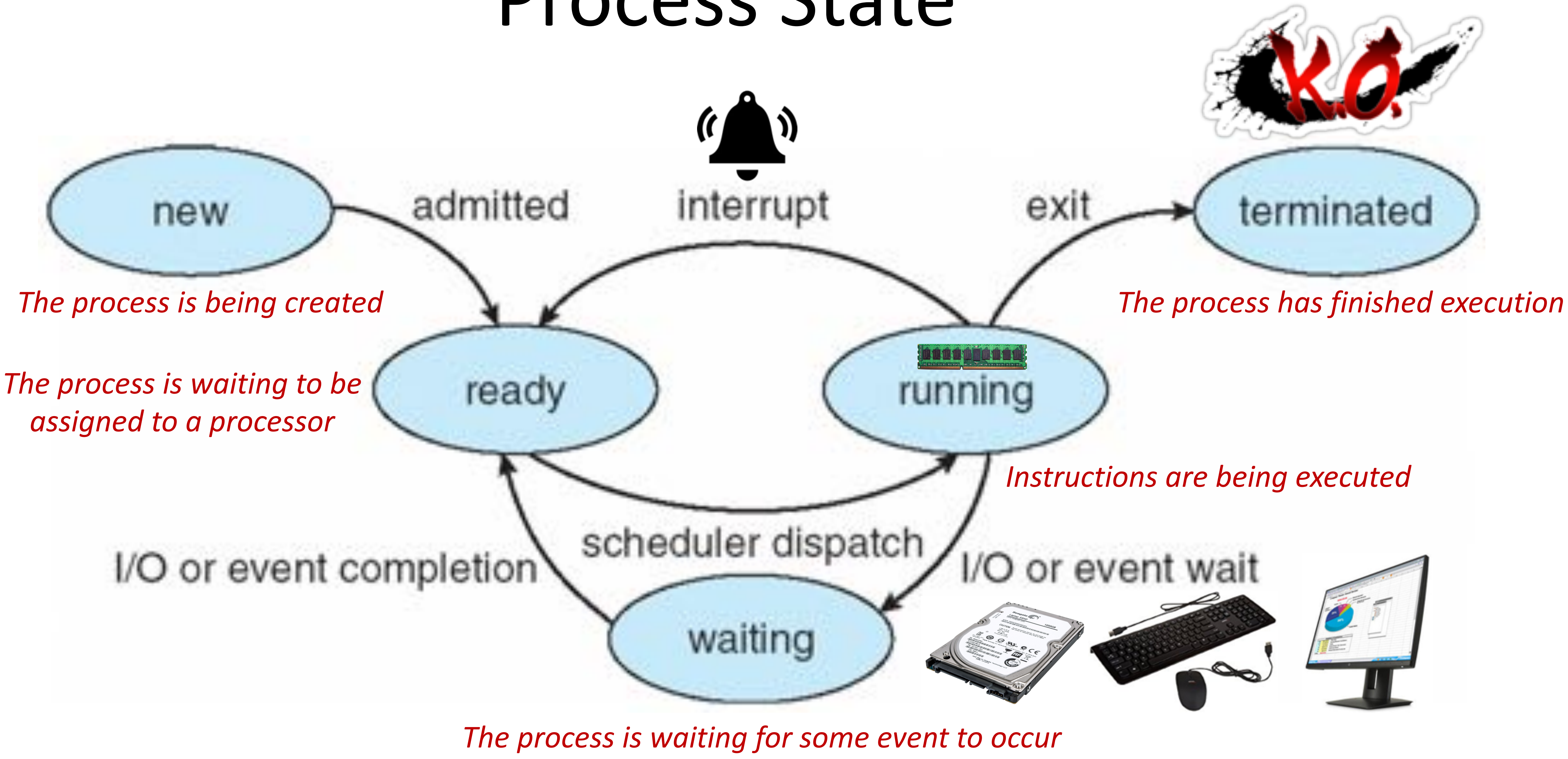


Chrome Browser is ***multiprocess*** with 3 different types of processes:

1. **Browser Process** manages user interface, disk and network I/O
2. **Renderer Process** renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
3. **Plug-in Process** for each type of plug-in



Process State



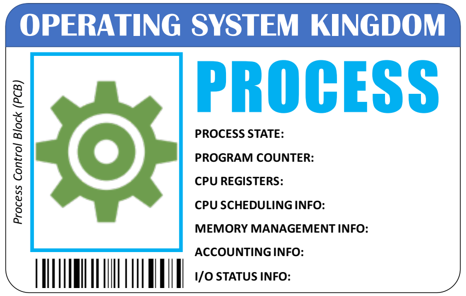
Context Switching

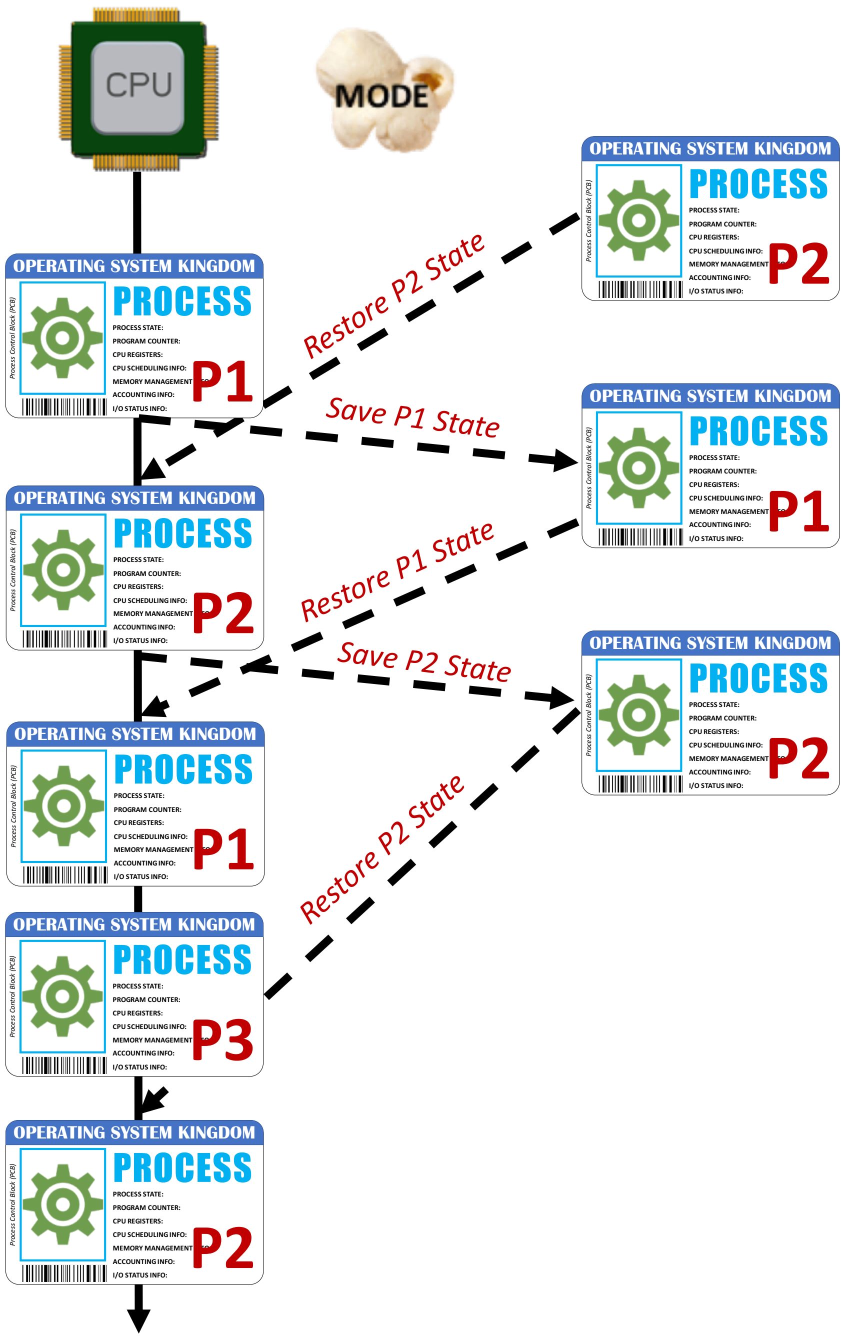
enables multiple processes to share a single CPU

The mechanism to store and restore **the state or context** of a CPU in **Process Control Block** so that a process execution can be resumed from the same point at a later time

*When the scheduler switches the CPU switches from executing one process to another process, the system must **save the state "Context"** of the old process and **load the saved state "Context"** for the new process*

Context





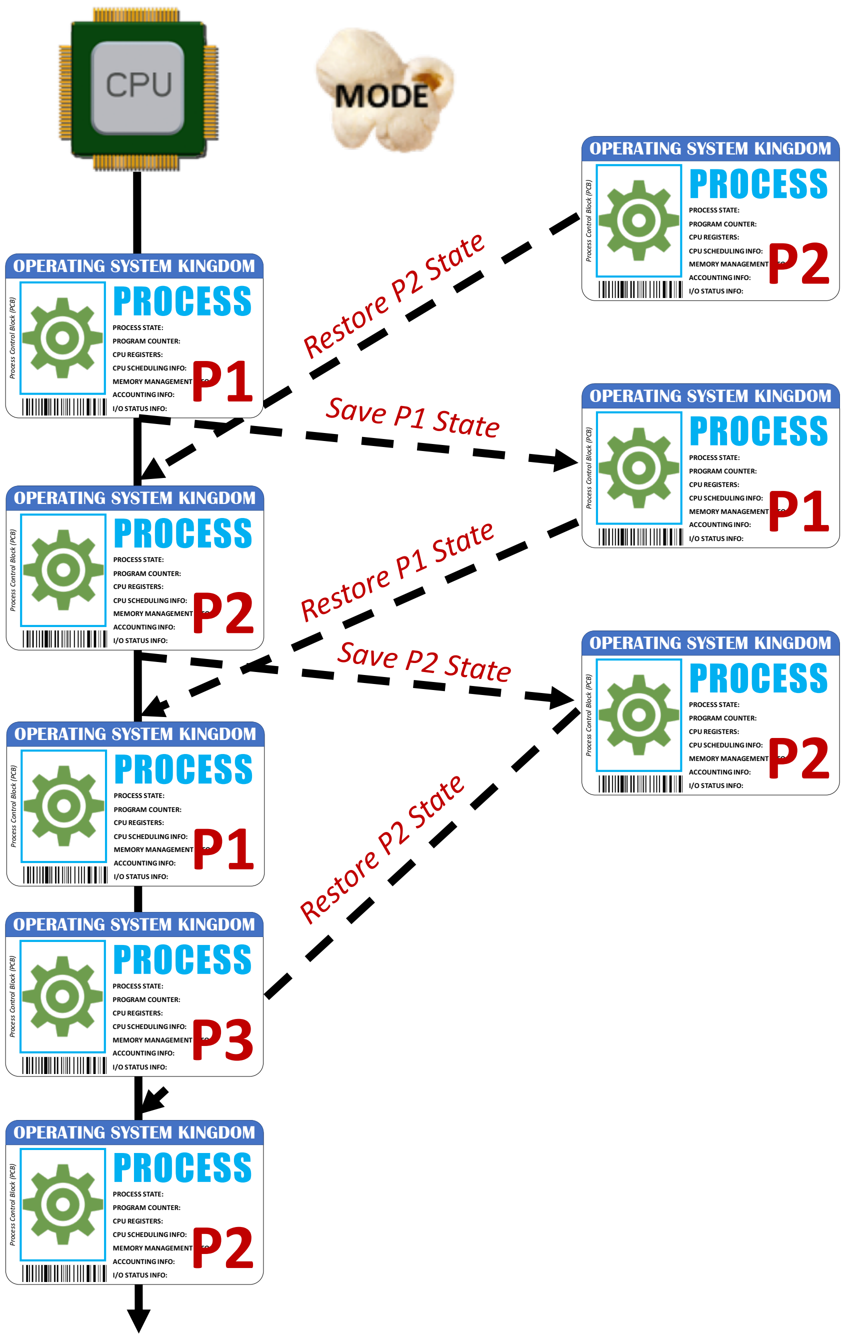
Context Switching

enables multiple processes to share a single CPU

Context switches are **computationally intensive** since register and memory state must be saved and restored

The more complex the OS and the PCB; the longer the context switching

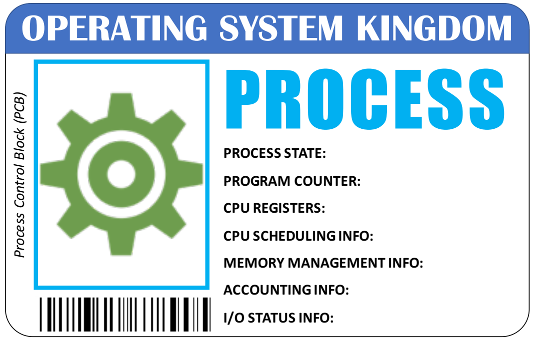
To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers so that multiple contexts loaded at once.

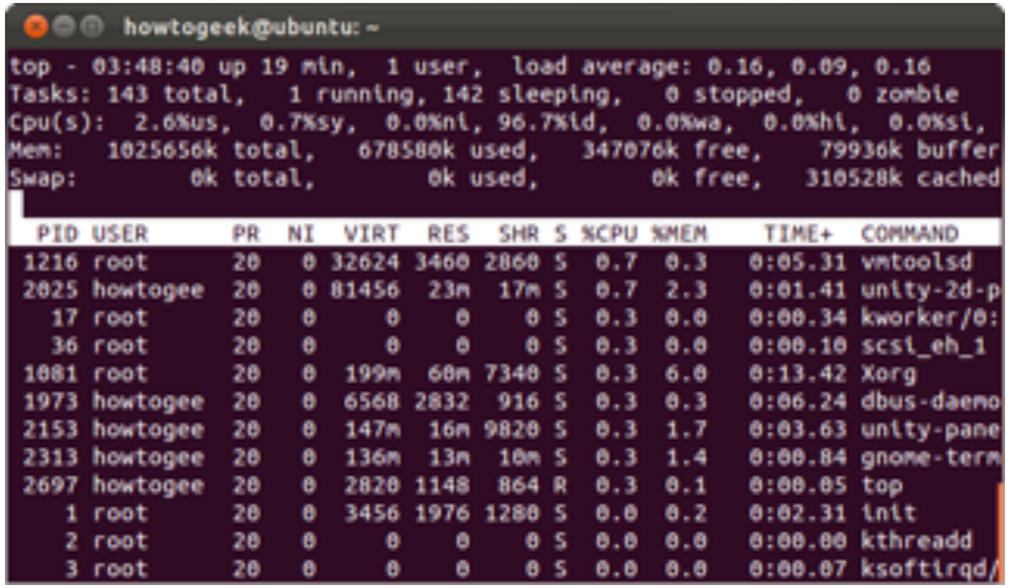


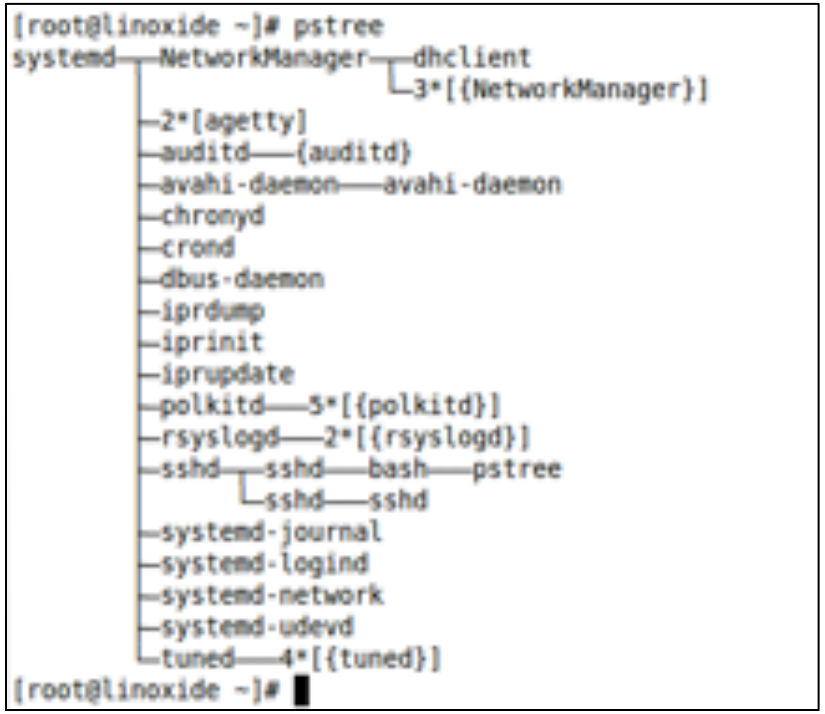
Process Creation

Parent *process creates* ***children*** *processes, which, in turn create other processes, forming a* ***tree of processes***

Process identified and managed via a process identifier (PID) – **Unique ID**



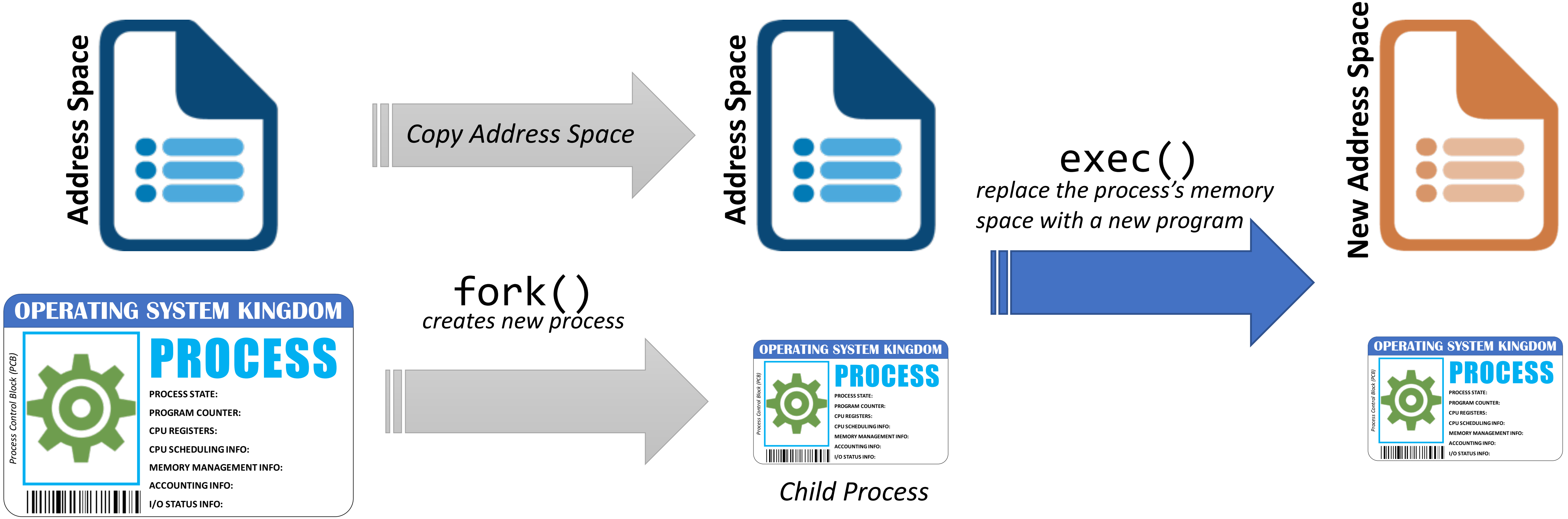




First process to run is the "systemd" process that is started at **system boot**. This is the grand parent of all processes in the whole system

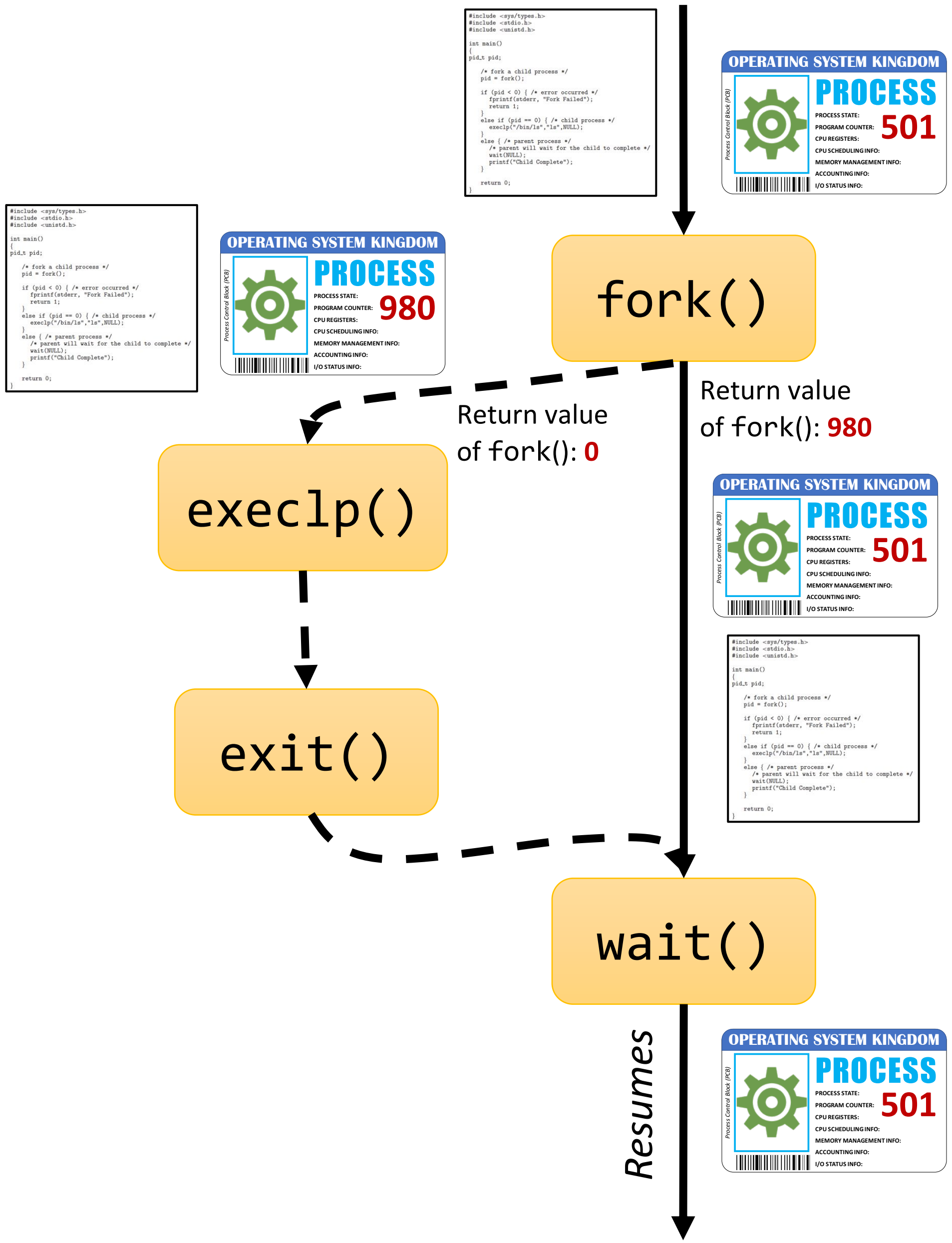
If a process dies, then its orphan children are re-parented to the "systemd" process

Process Creation



On most systems, the new child process **inherits the permissions of its parent**, unless the parent deliberately forks a new child process with lower permissions than itself.

Process Creation

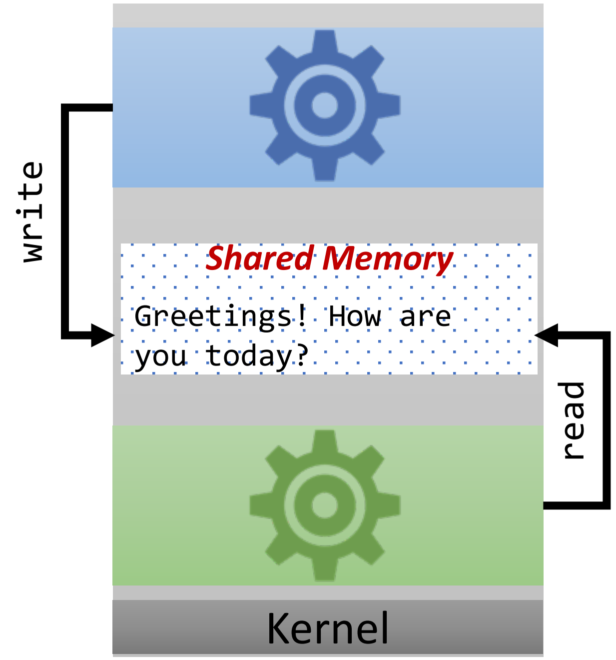



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

OS **prevents** one process from accessing another process's memory

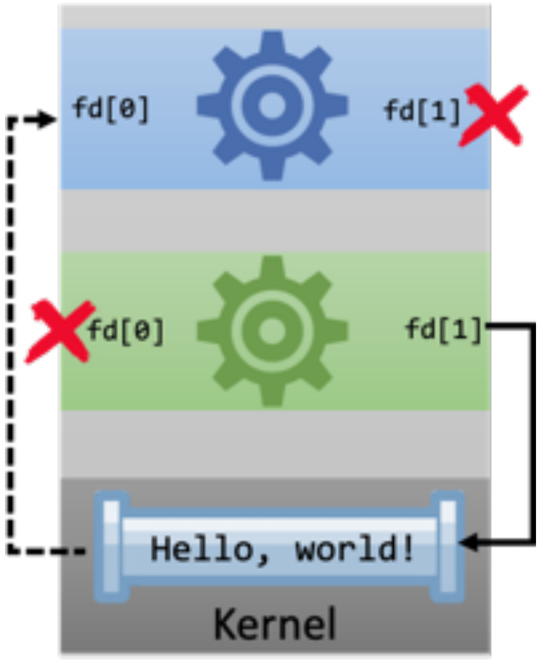
Inter-process Communication (IPC)

In order to manage shared resources, it is often necessary for processes to communicate with each other. Thus, operating systems usually include mechanisms to facilitate inter-process communication (IPC).



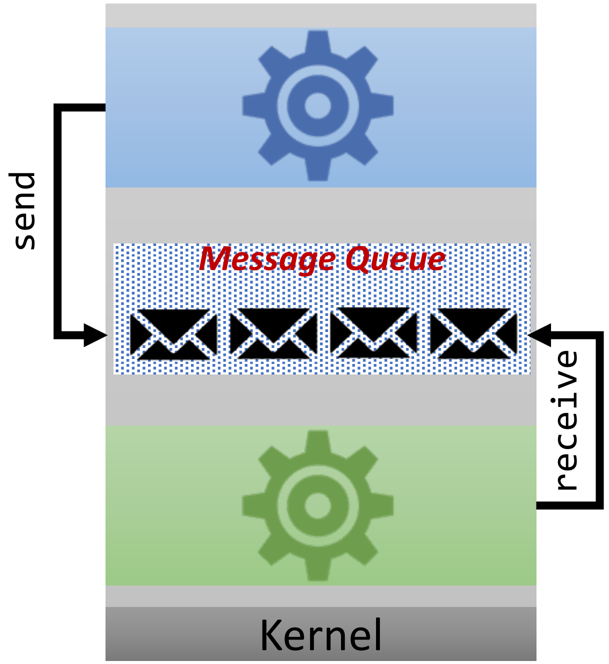
Shared Memory

A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region



Pipes

A conduit allowing related processes to communicate



Message Passing

Communication takes place by means of messages exchanged between the cooperating processes

Signals

- Sometimes, rather than communicating via shared memory or a shared communication channel, it is more convenient to have a means by which processes can send direct messages to each other **asynchronously**.
- Unix based systems incorporate signals, which are essentially notifications sent from one process to another.
- When a process receives a signal from another process, the operating system interrupts the current flow of execution of that process, and checks whether that process has an appropriate signal handler (a routine designed to trigger when a particular signal is received).
- If a signal handler exists, then that routine is executed; if the process does not handle this particular signal, then it takes a default action.

Signals

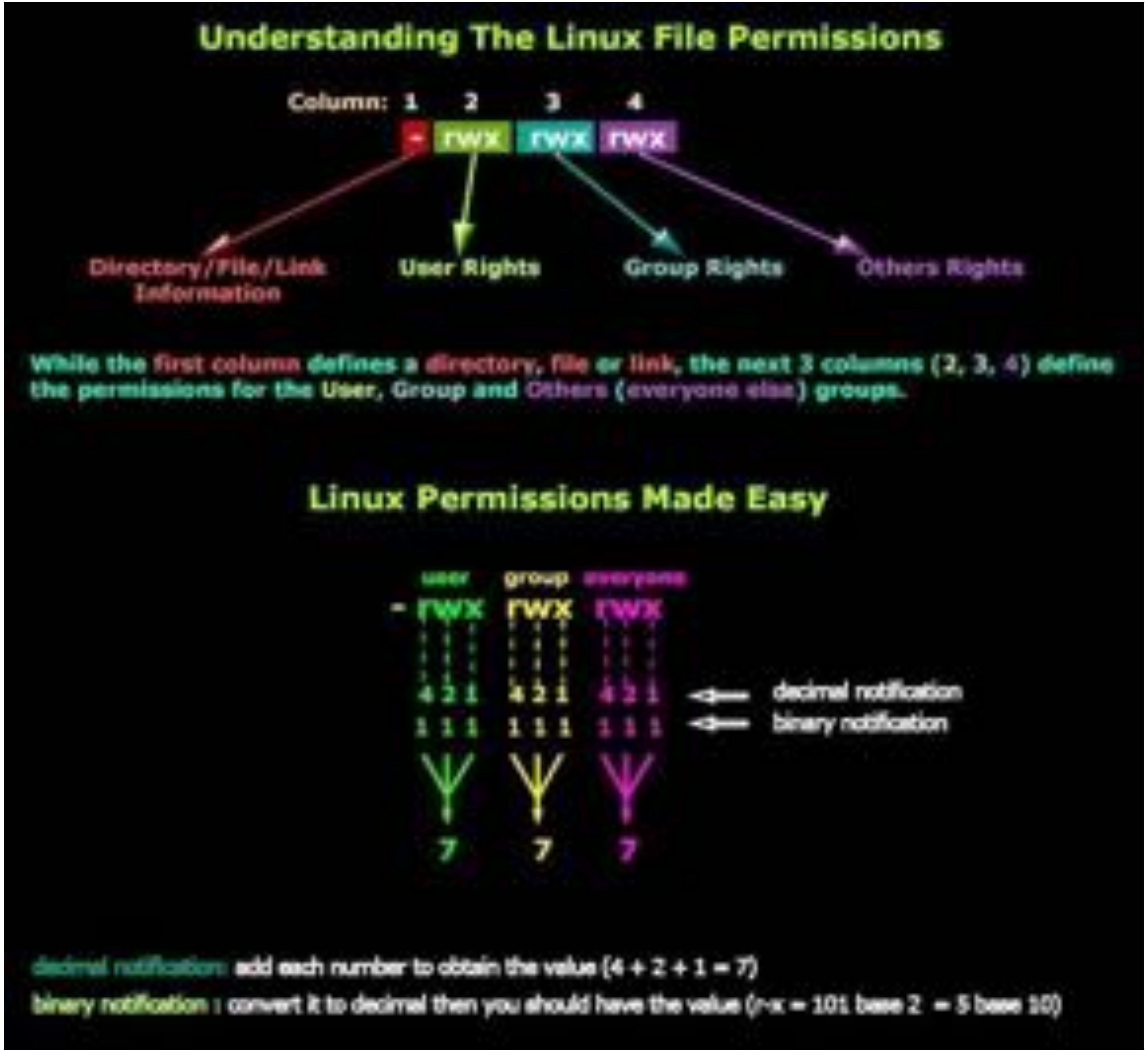
- Terminating a nonresponsive process on a Unix system is typically performed via signals.
- Typing Ctrl-C in a command-line window sends the INT signal to the process, which by default results in termination.

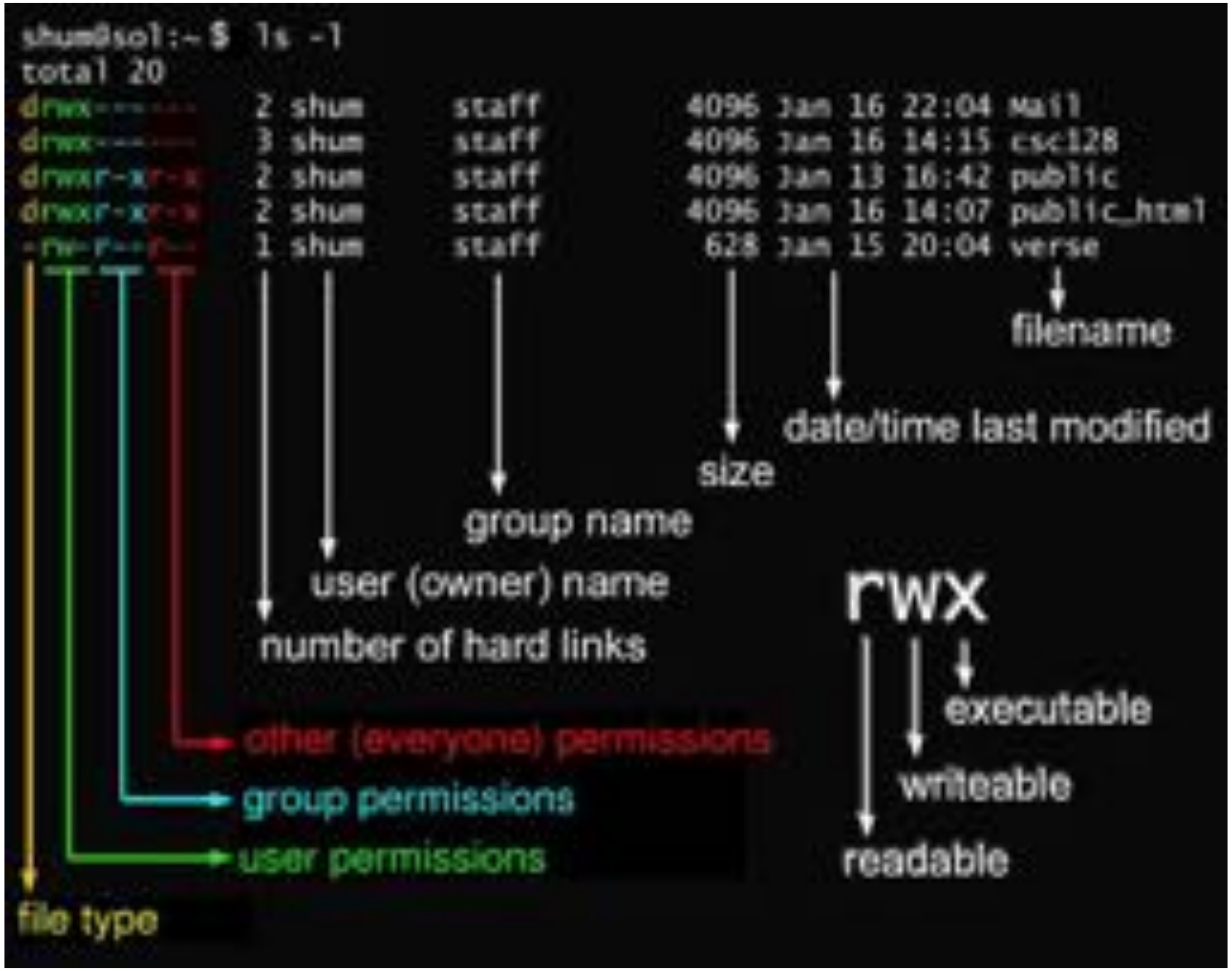
The Filesystem

- Another key component of an operating system is the filesystem, which is an abstraction of how the external, nonvolatile memory of the computer is organized.
  - Operating systems typically organize files hierarchically into folders, also called directories.

File Access Control

- One of the main concerns of operating system security is how to delineate which users can access which resources, that is, who can read files, write data, and execute programs.
- In most cases, this concept is encapsulated in the notion of file permissions, whose specific implementation depends on the operating system.
  - Namely, each resource on disk, including both data files and programs, has a set of permissions associated with it.


Understanding The Linux File Permissions
Column: 1 2 3 4
- rwx rwx rwx
Directory/File/Link Information
User Rights
Group Rights
Others Rights
While the first column defines a directory, file or link, the next 3 columns (2, 3, 4) define the permissions for the User, Group and Others (everyone else) groups.
Linux Permissions Made Easy
user group everyone
- rwx rwx rwx
4 2 1 4 2 1 4 2 1
1 1 1 1 1 1 1 1 1
decimal notification
binary notification
7 7 7
decimal notification: add each number to obtain the value (4 + 2 + 1 = 7)
binary notification : convert it to decimal then you should have the value (r-x = 101 base 2 = 5 base 10)

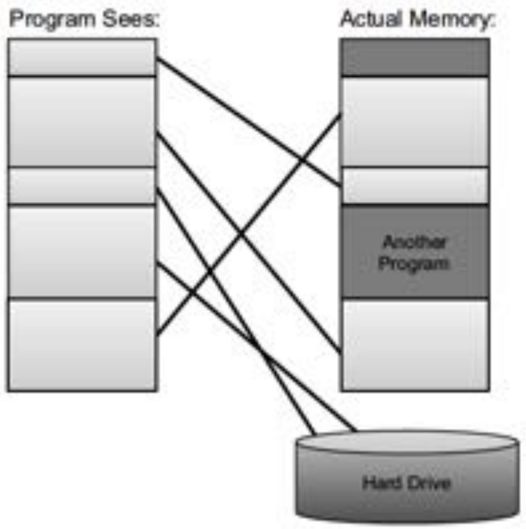

shum@sol:~$ ls -l
total 20
drwx------ 2 shum staff 4096 Jan 16 22:04 Mail
drwx------ 3 shum staff 4096 Jan 16 14:15 csc128
drwxr-xr-x 2 shum staff 4096 Jan 13 16:42 public
drwxr-xr-x 2 shum staff 4096 Jan 16 14:07 public_html
-rw-r--r-- 1 shum staff 628 Jan 15 20:04 verse
filename
date/time last modified
size
group name
user (owner) name
number of hard links
other (everyone) permissions
group permissions
user permissions
file type
rwx
executable
writeable
readable

Virtual Memory

- Even if all the processes had address spaces that could fit in memory, there would still be problems.
  - Idle processes in such a scenario would still retain **their respective chunks of memory**, so if **enough processes were running**, memory would be needlessly scarce.
- To solve these problems, most computer architectures incorporate a **system of virtual memory**, where each process receives a **virtual address space**, and each virtual address is **mapped to an address in real memory by the virtual memory system**.

Virtual Memory

- When a virtual address is accessed, a hardware component known as the memory management unit looks up the real address that it is mapped to and facilitates access.
  - Essentially, processes are allowed to act as if their memory is contiguous, when in reality it may be fragmented and spread across RAM



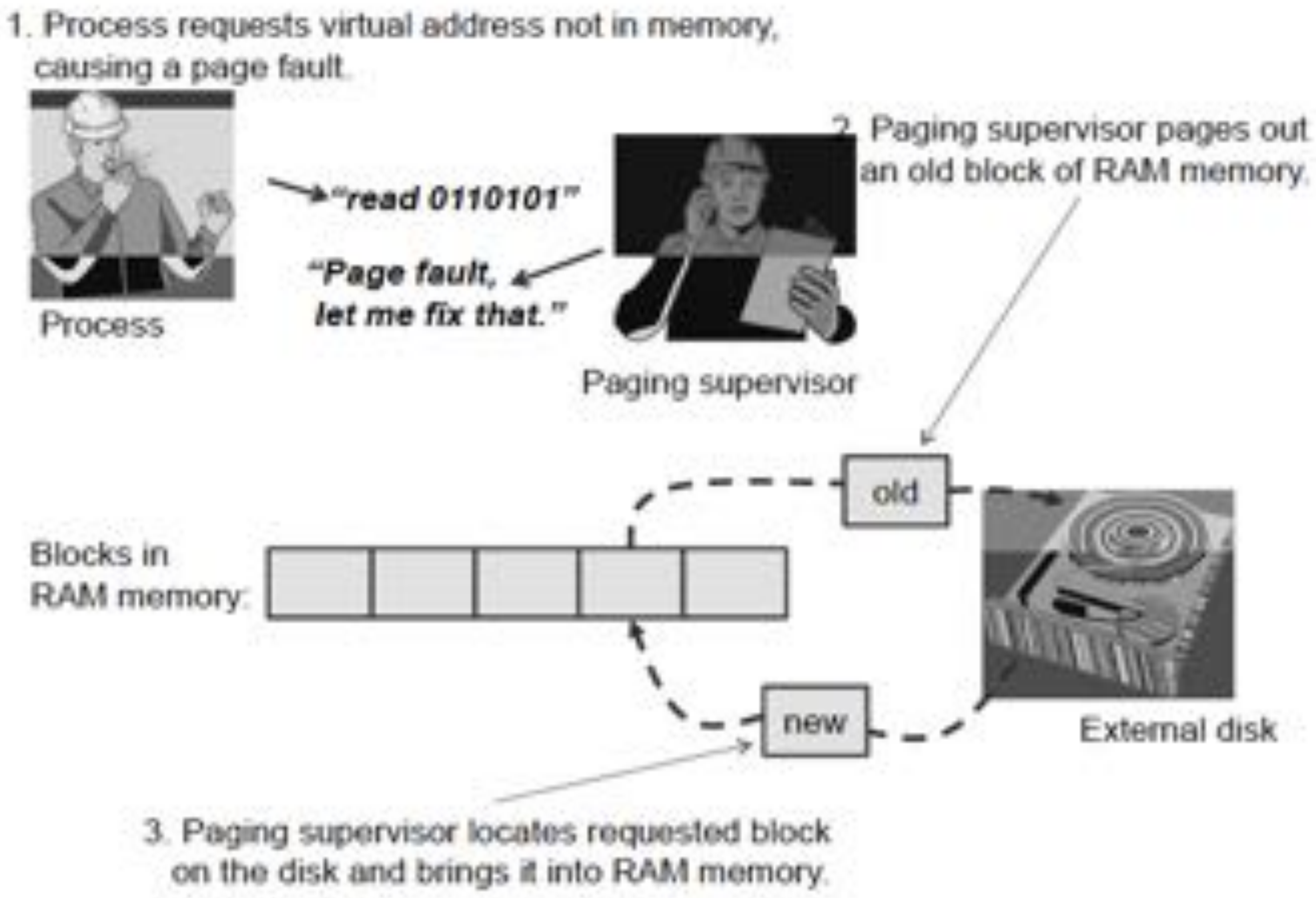
Virtual Memory

- An additional benefit of virtual memory systems is that they allow for the *total size of the address spaces of executing processes to be larger than the actual main memory of the computer*.
- This extension of memory is allowed because the virtual memory system can use a portion of the external drive to “park” blocks of memory when they are not being used by executing processes.
- This is a great benefit, since it allows for a computer to execute a set of processes that could not be multitasked if they all had to keep their entire address spaces in main memory all the time.

Page Faults

- There is a slight time trade-off for benefit we get from virtual memory, however, since accessing the hard drive is much slower than RAM. Indeed, accessing a hard drive can be 10,000 times slower than accessing main memory.
- So operating systems use the hard drive to **store blocks of memory that are not currently neede**d, in order to have most memory accesses being in main memory, not the hard drive.
- If a block of the address space is not accessed for an extended period of time, it may be paged out and written to disk. When a process attempts to access a virtual address that resides in a paged out block, it triggers a page fault.

Page Faults



1. Process requests virtual address not in memory, causing a page fault.
"read 0110101"
"Page fault, let me fix that."
Process
Paging supervisor
2. Paging supervisor pages out an old block of RAM memory.
old
Blocks in RAM memory:
new
External disk
3. Paging supervisor locates requested block on the disk and brings it into RAM memory.

Any program to run **must** be loaded in memory