Introduction to Operating System

50.005 Computer System Engineering

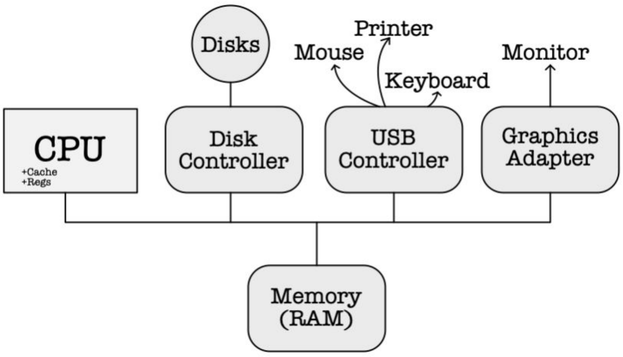
Materials taken from SGG: Ch. 1.1-1.6, 1.8.3, 1.13.3

Natalie Agus

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Introduction

An operating system (OS) is a program that **manages computer hardware.** The figure below shows the hardware components of a common general purpose computer. There are many user programs that are running in a computer, and the OS acts as an intermediary application that enables many user programs to share the same set of hardware, such as the mouse, printer, keyboard, display monitor, etc.



The Operating System

An operating system is a **special program** that acts as an intermediary between users of the computer and the computer hardware.

The goal of an operating system is such that we have a **dedicated program** to fulfil the following essential roles:

1. **Resource allocator and coordinator**: controls hardware and input/output requests, manage conflicting requests, manage interrupts
2. **Controls program execution**:
    a. Storage hierarchy manager
    b. Process manager
3. Limits program execution and ensure **security**: preventing illegal access to the hardware or improper usage of the hardware

Once we have an *operating system,* it makes things easier for users to use a program / code a another program for other purposes within a **computer system.**

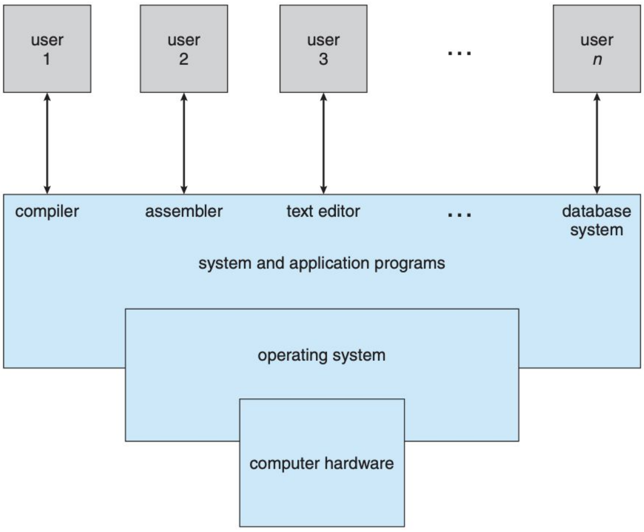
There are a lot of things that make up an operating system, but they are generally divided into three categories:

1. **The Kernel**
2. **System programs**
3. **User programs**

Definition and role of **operating system kernel** can be found in the section below, and it **is the only program** with **full privileges, i.e: absolute access** to control all the hardware in the computer system. Both system programs and user programs run in **user mode,** with **limited privileges**, i.e: any of these programs have to send a *request* to the kernel each time they require access to the I/O or hardware devices.

The Computer System

A computer system can be roughly divided into four components: the hardware, the operating system, the application programs, and the users as shown in the figure below.

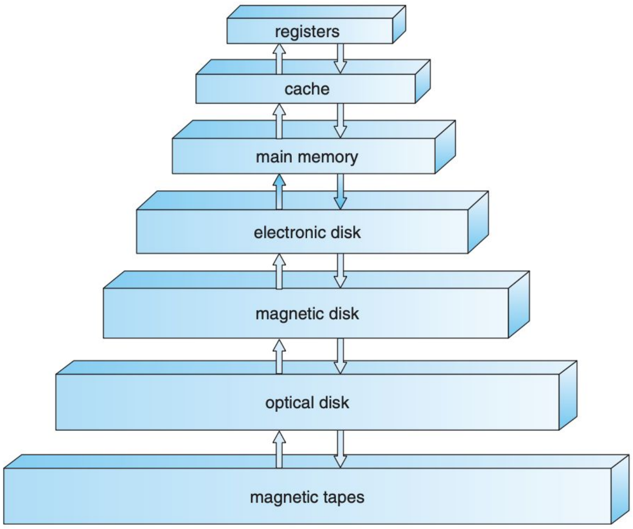


*The operating system is part of the computer system and is analogous to a **government.***

It provides an *environment* such that user programs such as the text editor, web browser, compiler, database system, music player, video editor, etc can do useful work. Since each user program runs in **virtual machine** (i.e: it is written in a manner that the *entire* machine belongs to itself), there has to be some sort of another program that **manages** and **oversees** all programs that lives on the RAM and reside on disk, as well as managing the **memory hierarchy**. This special program is part of the operating system called the **kernel.**

The Memory Hierarchy

The storage structure in a typical computer system is made of **registers**, **caches**, **main memory**, and **non-volatile secondary storage** such as magnetic disk. The wide variety of storage systems can be arranged in terms of hierarchy according to **speed and cost** (increasing speed and increasing cost from bottom up):



The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewritable memory, called main memory (RAM). At each CPU clock cycle, instructions are fetched from the main memory to the CPU.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is **not possible,** for two reasons:

1. Main memory is usually **too small to store all needed programs** and data permanently.
2. Main memory is a **volatile storage device** that loses its contents when power is turned off or otherwise lost.

Recall that the memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data).

Cache devices are typically used to speed up the performance of the computer. They are storing (typically) a few of the most recently used instruction pages. Cache devices are wired directly to the CPU so that the CPU has direct access to it (unlike secondary storages), just like how the CPU can directly access the RAM. You may see it as a more-expensive, smaller-but-faster RAM.

Because caches have limited size, cache management is an important design problem. The **kernel** code contains the actual implementation of caching algorithm that is suitable for the system.

The Kernel

The Kernel is the heart of an operating system. It operates on the physical space — meaning that it has full knowledge of all physical addresses instead of virtual addresses, and has the **ultimate** privilege over all the hardware of the computer system.

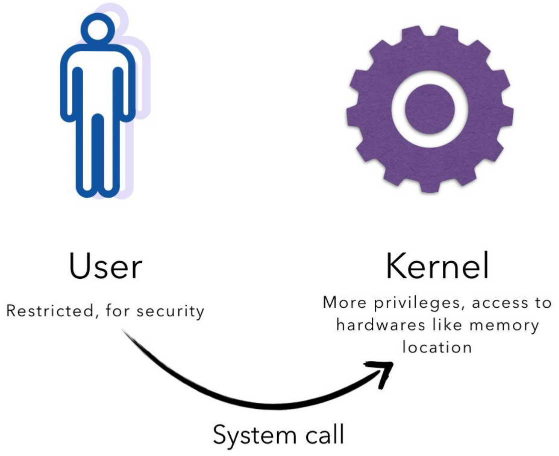
The one program that is running at all times in the computer is the kernel

The kernel runs with special privileges, called the **kernel mode**. It can do what normal user program cannot do:

1. Ultimate access and control to all hardware in the computer system (mouse, keyboard, display, network cards, disk, RAM, CPU, etc)
2. Know (and lives in) the physical address space and manages the memory hierarchy
3. Interrupt other user programs
4. Receive and manage I/O requests
5. Manage other user program locations on the RAM, the MMU, and schedule user program executions

In order for the **kernel** to have more *privileges* than **other user mode programs**, the computer hardware has to support **dual mode operation.**

Dual Mode Operation



The dual mode is possible iff it is supported by the hardware. The kernel is also ***uninterruptible*** *and this interruptible feature is also supported by the hardware.*

In 50.002, we have learned that the control logic unit **prevents** the PC to JMP to memory address with MSB bit of 1 (where kernel program resides) when it was at memory address with MSB bit 0 (user mode). Also, the control logic unit does not trap the PC onto the handler when **interrupt** signal is present if the PC is running the **kernel program** (MSB of the PC is 1)**.**

In Linux system, low memory is dedicated for kernel and high memory is assigned for user processes. It is essentially the same as what we have learned before: the concept of having *dual* mode and *hardware support*. In Linux system, the hardware prevents the PC from jumping *illegally* (not through handlers) to lower memory address (MSB = 0) when it was from higher memory address (MSB = 1).

A general purpose CPU has at least dual mode operation that is supported by its hardware:

1. **The Kernel mode** (privileged)
2. **The User mode** (unprivileged) : all user programs such as a web browser, word editor, etc and also system programs such as compiler, assembler, file explorer, etc. Runs on *virtual machine*

User programs have to perform **system calls** (supervisor call — i.e: programs interrupt themselves) when they require services from the kernel, such as access to the hardware or I/O devices.

Booting

'Booting' is a process of starting up a computer. **It is usually hardware initiated** — (by the start button that users press) — meaning that users physically initiate simple hardwired procedures to kickstart the chain of events that loads the firmware (BIOS) and eventually the entire OS to the main memory to be executed by the CPU. This process of loading basic software to help kickstart operation of a computer system after a hard reset or power on is called **bootstrapping**.

Recall that softwares or programs (including the operating system kernel) must **load** into the main memory before it can be executed. However, at the instance when the start button is pressed, there's no program that resides in the RAM and therefore nothing can be executed in the CPU.

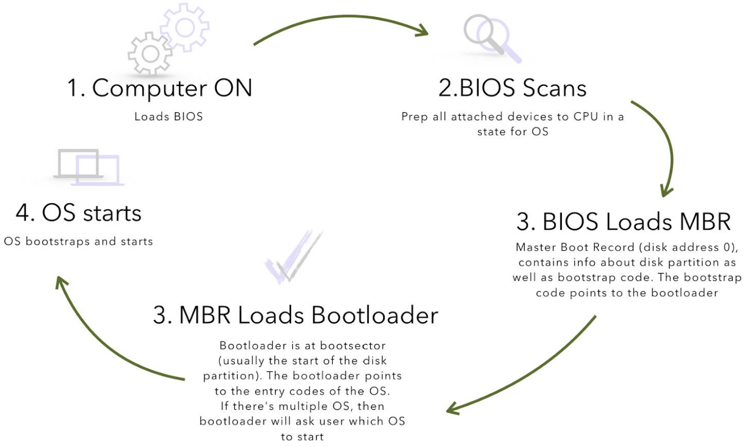
We require a software to load another software into the RAM — this results in a paradox.

To solve this paradox, the bare minimum that should be done in the hardware level upon pressing of the start button is to load a *special* program onto the main memory from a dedicated input unit — **a read-only-memory (ROM) that comes with a computer when it is produced and that cannot be erased.** *This special program is generally known as* ***firmware or BIOS.***

After the firmware is loaded onto the main memory through hardwired procedures, the CPU may execute it and **initialise all aspects of the system, such as:**

1. Prepare all attached devices in a state that is ready to be used by the OS
2. Loads other programs — which in turn loads more and more complex programs finally,
3. Loads the Kernel from disk

The figure below summarises the startup process

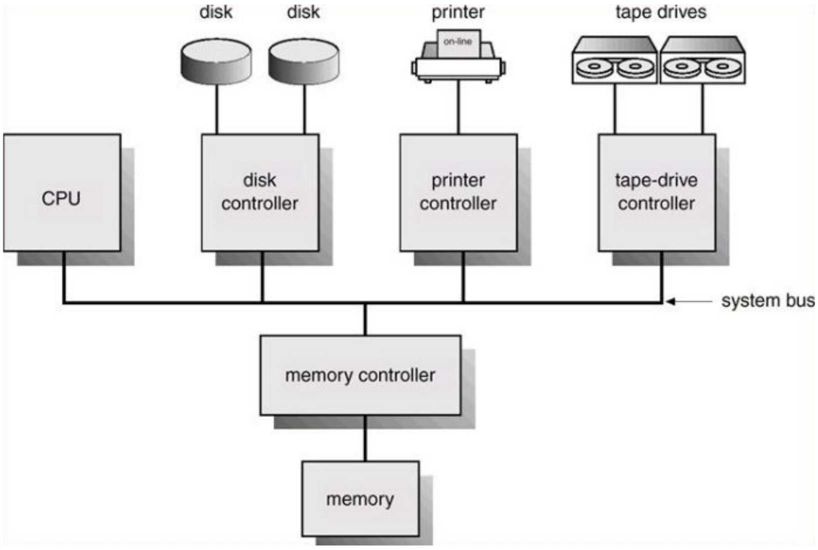

1. Computer ON
Loads BIOS
2.BIOS Scans
Prep all attached devices to CPU in a state for OS
3. BIOS Loads MBR
Master Boot Record (disk address 0), contains info about disk partition as well as bootstrap code. The bootstrap code points to the bootloader
3. MBR Loads Bootloader
Bootloader is at bootsector (usually the start of the disk partition). The bootloader points to the entry codes of the OS. If there's multiple OS, then bootloader will ask user which OS to start
4. OS starts
OS bootstraps and starts

Computer System I/O Operation

There are two types of hardware in the computer system that are **capable of running code or machine instructions:**

1. The CPU
2. I/O device controllers

Each I/O device is managed by an autonomous hardware entity called the **device controllers** as shown in the figure below:



In other words, I/O devices and the CPU can execute concurrently.

An OS have a **device driver** for each device controller — where this driver is a specific program to **understand** the specific behavior of each device controller (we typically install *device driver* when we plug in new I/O unit to our computers through the USB port).

Components that make up the device controllers:

1. Registers — contains instructions that can be read by an appropriate **device driver program at the CPU**
2. Local memory buffer — contains instructions and data that will be fetched by the CPU when executing device driver program, and ultimately loaded onto the RAM.
3. A simple program to *communicate* with the device driver

I/O operation happens when there's transfer of data between the local memory buffer of the device controller and the device itself.

We always need to move data from the RAM to the device controller's buffer, or from the device controller's buffer to the RAM. Therefore, we need the **kernel** to **coordinate** between executing I/O process and executing other user programs.

Roles of an Operating System Kernel

As mentioned in the introduction, there are several purposes of an operating system: as a resource allocator, controls program execution, and guarantees security in the computer system. Before we can understand these roles, we need to first understand how computer system I/O operates.

Resource Allocator and Coordinator

The kernel controls and coordinates the use of hardware and I/O devices (as seen in the section above) among various user applications. Examples of I/O devices include mouse, keyboard, graphics card, disk, network cards, among many others.

I/O Interrupts — How I/O operations work in a nutshell when there's external request:

1. Upon the presence of new **input** : such as keyboard press, mouse click, mouse movement, incoming fax, the device controllers will throw an **interrupt** request.

2. This **transfer** control to the interrupt handler in **the Kernel mode**. The interrupt handler will **save the register states** first (the interrupted program instruction) into the process table before transferring control to the appropriate interrupt service routine — depending on the device controller that made the request.

3. There are two ways to know which controller made the request:
    a. **Vectored system:** an interrupt signal that **includes** the identity of the device sending the interrupt signal

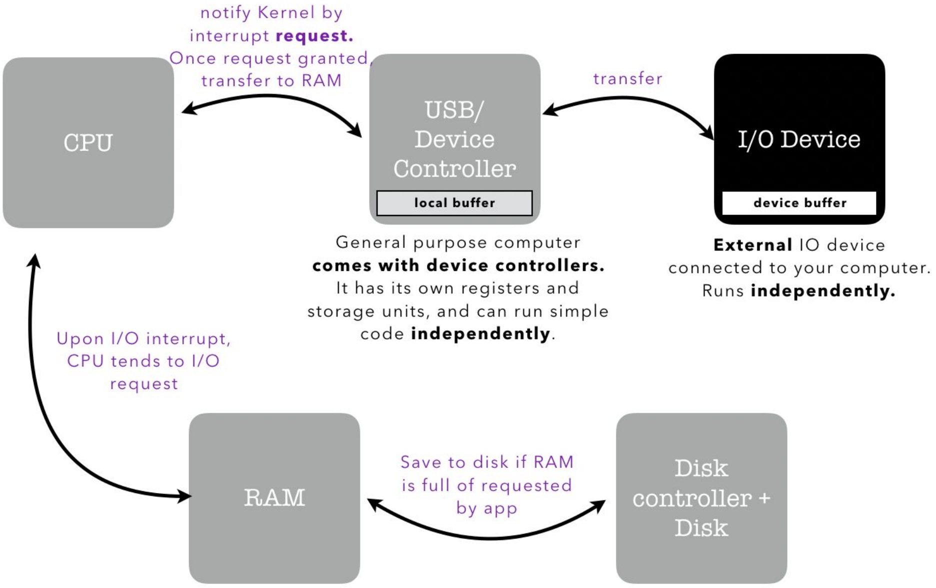
    b. **Polled system:** an interrupt signal that **does not include** the identity of the device sending the interrupt signal. The interrupt controller must **send a signal out** to each device to determine which one made the request.

4. If there's more than one I/O interrupt requests from multiple devices, the Kernel may decide which interrupt requests to service.

5. When the service is done, the Kernel scheduler may choose to resume the user program that was interrupted.

Modern computer systems are **interrupt-driven**. This way, the computer may efficiently fetch I/O data only when the devices are ready, and will not waste CPU cycles to "check" whether I/O has arrived or not.

The figure below summarises the **interrupt-driven** procedure of **asynchronous I/O handling:**


notify Kernel by interrupt **request.** Once request granted, transfer to RAM
CPU
USB/ Device Controller
local buffer
transfer
I/O Device
device buffer
General purpose computer **comes with device controllers.** It has its own registers and storage units, and can run simple code **independently**.
External IO device connected to your computer. Runs **independently.**
Upon I/O interrupt, CPU tends to I/O request
RAM
Save to disk if RAM is full of requested by app
Disk controller + Disk

We can simply plot a timeline for these interrupt requests to understand the interrupt process execution better. In the figure below, the CPU **multiplexes** between executing user process as well as I/O processes (device driver).

The 'Interrupt' timeline

1. At first, the CPU is busy doing tasks for programs in user mode. At the same time, the device is idling.

2. The **device controller** (not shown, independent of the CPU as well) throws the first I/O request — assume that these I/O requests were made by some system calls by programs in user mode, etc.

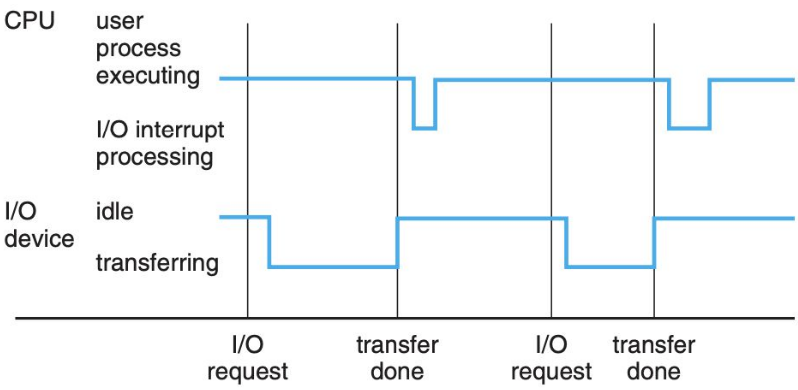
3. The I/O device then proceeds on responding to the request and **transfer** the data from the **device buffer** to the **local device controller buffer.**

4. When I/O transfer is complete, the device controller throws a *transfer done* **interrupt**. The simplest way for the device controller to throw an interrupt is by **hardware control directly to the system's CPU** (a direct IRQ signal)

5. User program is interrupted, states saved by the handler, and the CPU executes I/O processing to transfer **the data from device controller buffer** to the **RAM**

6. User mode is resumed — typically scheduler will decide which user program to resume

7. Steps 2-5 are repeated upon another I/O request by the device controller



System Call — How I/O operations work in a nutshell when there's internal request

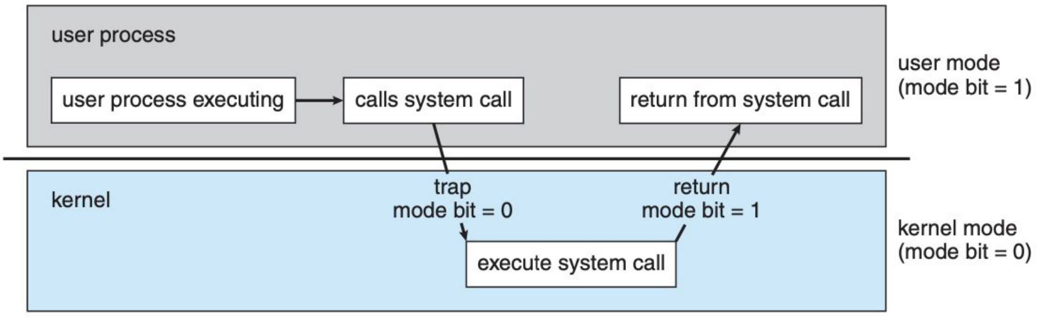
Sometimes user programs will require inputs from external devices and may **pause** itself until the presence of the required input arrives as shown in the figure below.

The steps of invoking **system calls** are summarised as follows:

1. User programs may **trap** themselves by purposely throwing an **illegal operation as a system call** shown in the figure below.

2. This will invoke the illegal operation handler in Kernel mode

3. Similarly, all user states will be saved before the handler proceeds to invoke the appropriate system call handler to service the user request



Thus entry points to the Kernel code can only be done by making the appropriate system calls.

Examples of system calls are: fork(), chdir(), and more Linux system calls can be found at http://man7.org/linux/man-pages/man2/syscalls.2.html

Reentrancy in Interrupts

Incoming interrupts are typically ***disabled*** while another interrupt (of same or higher priority) is being processed to **prevent** a ***lost interrupt* —** i.e: when user states are currently being saved before the actual interrupt service routine began or various ***reentrancy***[1] **problems**.

Timed Interrupts

We must ensure that the kernel — as a resource allocator — **maintains control** over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system.

To ensure that no user program can occupy a CPU for indefinitely, a computer system comes with a (hardware) timer. A timer can be set to interrupt the computer by the kernel after a specified period.

How kernel utilises the timer to allocate resources:

1. The period of the timer itself may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second).
2. A **variable** timer is generally implemented by a fixed-rate clock and a counter.
3. The operating system kernel **sets the counter:**
    a. Every time the clock ticks, the counter is decremented.
    b. When the counter reaches 0, an **interrupt** occurs, currently running user program is *paused*
    c. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Memory Management

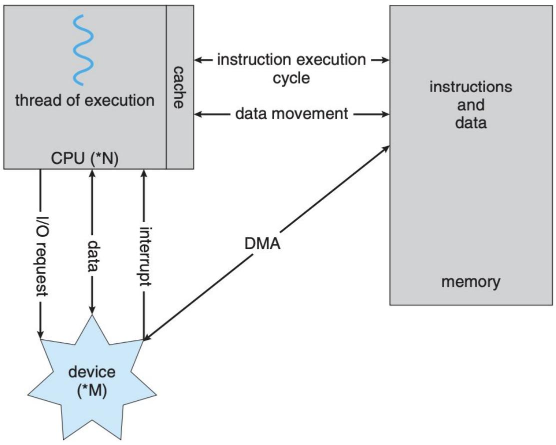
The kernel also has to manage the memory hierarchy:

1. **Handles the caching algorithm:**
    a. Keeping track of which parts of memory are currently being used and by whom
    b. Deciding which processes and data to move into and out of memory
    c. Allocating and deallocating memory space as needed
    d. If RAM is full, migrate some contents (e.g: least recently used) onto the **swap space** of the disk
2. **Ensures cache coherency for multiprocessor system**

[1] A reentrant procedure can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution. There are several problems that must be carefully considered before declaring a procedure to be *reentrant*. All user procedures are reentrant, but it comes at a costly procedure to prevent lost data during interrupted execution.

As we have seen in the section above, the hierarchical storage structure requires some form of memory management since the same data may appear in **different levels** of storage system. **Caching** is an important principle of computer systems. We perform the **caching algorithm** each time the CPU needs to execute a new piece of instruction or fetch new data **from the main memory**. In fact, cache is part of the CPU itself in modern computers as shown in the figure below:

Note that "DMA" means direct memory access from the device controller onto the RAM[2]*.*



In the kernel, we will find **cache management algorithms** — where the kernel will swap pages, updates the all of the relevant MMU pagetables, and perform necessary writes during swaps. Careful selection of the cache size and of a replacement policy can result in greatly increased performance.

The kernel has to also ensure **cache coherency** in **multiprocessor architecture,** such that all CPUs have access to the most recently updated data. Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy.

Given a *cache hit ratio* α, cache miss access time ε, and cache hit access time τ, we can compute the **effective access time as** $\alpha\tau + (1-\alpha) * \varepsilon$

[2] Interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, direct memory access (DMA) is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU.

Process Management to Support Multiprogramming and Timesharing feature

Multiprogramming is a concept that is needed for efficiency. **Multiprogramming: increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.** The kernel is responsible to schedule processes in a computer system.

The reason for the need of multiprogramming is as follows:

- **Single user cannot keep CPU and I/O devices busy at all times.**
    - Since the clock cycles of a general purpose CPU is very fast (in Ghz), we dont actually need 100% CPU power in most case
    - It is often too fast to be dedicated for just one program for the entire 100% of the time
    - Hence, if multiprogramming is not supported, the CPU will spend most of its time *idling*
- The kernel organizes jobs (code and data) **efficiently** so CPU always has one to execute
- A subset of total jobs in system is kept in memory + swap space of the disk. **Virtual memory** allows execution of processes not completely in memory.
- One job is selected per CPU and run by the scheduler
- When a particular job has to wait (for I/O for example), OS switches to another job (context switching)

Multiprogramming allows for timesharing — context switch is performed so rapidly so that users still see the system as *interactive* and seemingly capable to run *multiple*.

- Multiprogramming alone does not necessarily *provide* for user interaction with the computer system
- Time sharing -- the logical extension of multiprogramming -- requires an interactive computer system, which provides direct communication between the user and the system.
- Accordingly, the response time should be short—typically less than one second.

Process vs Program

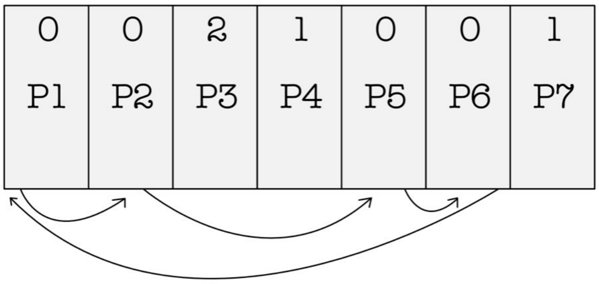
Formally, we can define a program in execution as a process (code that resides in RAM). It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.

A program resides on *disk* and isn't currently used. It does not take up **resources: CPU cycles, RAM, I/O, etc.** When a process **terminates**, the resources are *freed* by the kernel — such as RAM space is freed, so that other processes may use the resources. A typical general-purpose computers run multiple processes at a time.

If you are interested to find the list of processes in your Linux system, you can type `top` in your terminal to see all of them in real time. A single CPU achieves **concurrency** by multiplexing the executions of many processes.

Process Manager

The kernel **process manager** manages and keeps track of the system-wide **process table**, for example as illustrated in the figure below. Seven programs are currently running, and each process typically has some kind of **indicator**. In this example, assume that non-zero values means that the process is waiting for I/O on certain devices. As a result, the kernel will only **schedule** a round-robin execution between P1, P2, P5, and P6. Other processes are **skipped** since they need to wait for some incoming I/O processes. **This greatly improves efficiency since we do not waste CPU cycles "waiting" for incoming (asynchronous) I/O.**



0	0	2	1	0	0	1
P1	P2	P3	P4	P5	P6	P7

The process manager in the Kernel is responsible for the following:

1. Creation and deletion of both user and system processes
2. Pausing and resuming processes in the event of **interrupts** or **system calls**
3. Synchronising processes and provides communications between virtual spaces
4. Provide mechanisms to handle **deadlocks**

Providing Security and Protection

The operating system kernel provides **protection** for the computer system — a mechanism for **controlling** access of processes or users to resources defined by the OS.

How operating system identifies users:

1. Using user identities (user IDs, security IDs) include name and associated number
2. User ID then associated with all files, processes of that user to determine access control
3. For shared computer: group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file

Privilege escalation happens when a user can change its ID to another effective ID with more rights. You may read on how this can happen in Linux systems here.

It also provides **security** for the computer system — a **defense** mechanism against internal and external attacks (firewalls, encryption, etc). There's a huge range of security issues when a computer is connected in a network, some of them include denial-of-service, worms, viruses, identity theft, theft of service.

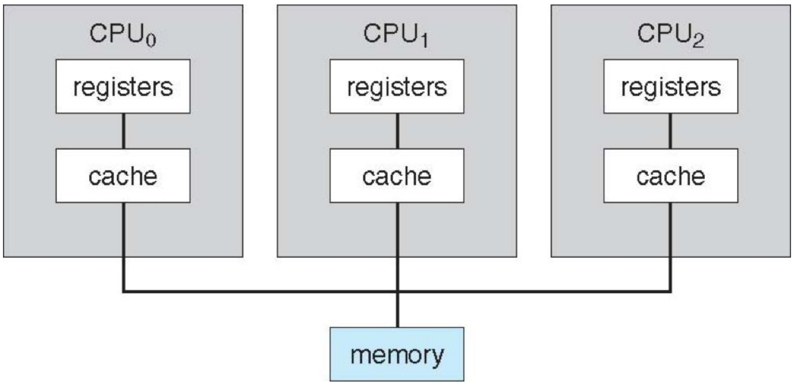
Multiprocessor System

Unlike single-processor system that we have learned before, multiprocessor systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Multiprocessor systems have three main **advantages**:

1. Increased **throughput** — we can execute many processes in parallel
2. **Economy** of scale — multiprocessor system is cheaper than multiple single processor system
3. Increased **reliability** — if one core fails, we still have the other cores to do the work

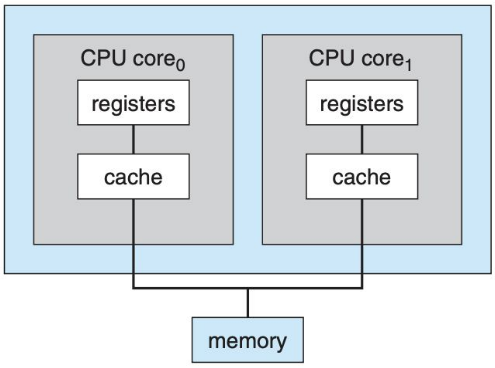
There are different architectures for multiprocessor system, such as a **symmetric architecture** — we have multiple CPU chips in a computer system:



Notice that each processor has its own set of registers, as well as a private — or local — cache; however, all processors share physical memory. This brings about design issues that we need to note in symmetric architectures:

1. Need to **carefully control I/O** to ensure that the data reach the appropriate processor
2. **Ensure load balancing:** avoid the scenario where one processor may be sitting idle while another is overloaded, resulting in inefficiencies
3. Ensure **cache coherency**

Another example of a symmetric architecture is to have **multiple cores on the same chip** as shown in the figure below (a dual core chip example). Nevertheless, they are also multiprocessor systems that are built on a single chip — this carries an advantage that **on-chip communication is faster than across chip communication**. However it requires a more delicate hardware design to place multiple cores on the same chip.



Some other systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task with the presence of a super processor that controls the system. This scheme defines a master–slave relationship. The master processor schedules and allocates work to the slave processors. The other processors either look to the master for instruction or have predefined tasks.