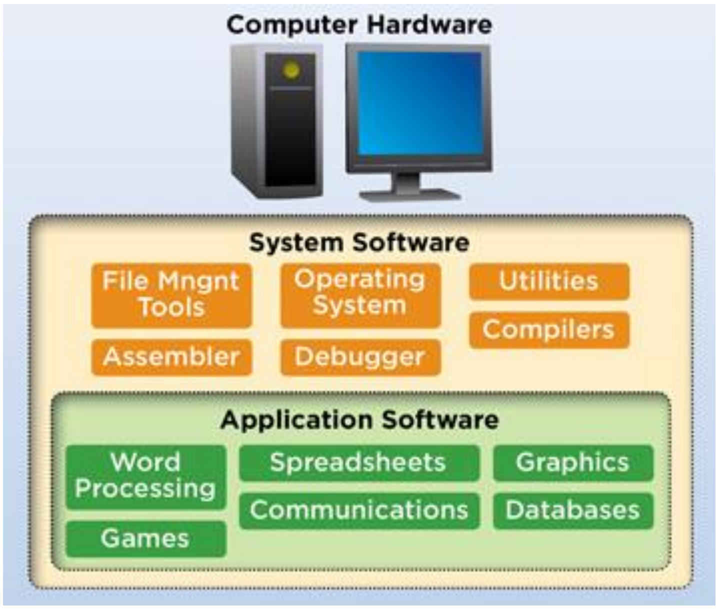
Contents

Knowing how operating systems work is a fundamental and critical to you as a computer science student!

The operating system is the "brain" that manages input, processing, and output, all other disciplines interact with the operating system. An understanding of how the operating system works will provide valuable insight into how the other disciplines work, as your interaction with those disciplines is managed by the operating system.



When a brand new computer comes off the factory assembly line, it can do nothing. The hardware needs software to make it work.

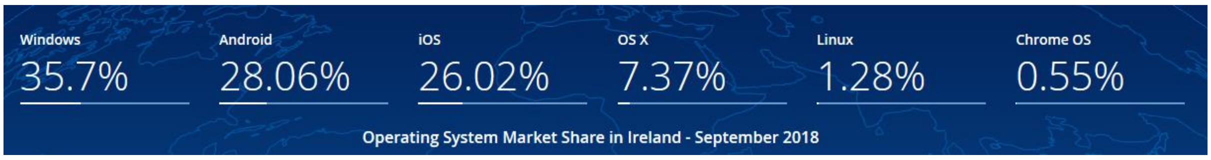
An Operating System (OS) manages and coordinates the function performed by the computer hardware, including the CPU, input/output devices, secondary storage device, and communication and network equipment. The OS is the most important program that runs on a computer with major design goals of efficient use of hardware and ease of use of resources.

It hides hardware complexity, manages computational resources, and provides isolation and protection. Most importantly, it directly has privilege access to the underlying hardware

You probably have used both Desktop (Windows, Mac, Linux) and Embedded (Android, iOS) operating systems before.

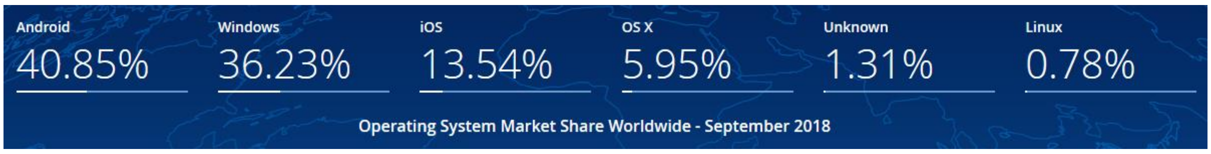
Current Operating Systems Market Share

According to StatCounter (2018), the following shows the OS market share in Ireland in Sept 2018:



Windows	Android	iOS	OS X	Linux	Chrome OS
35.7%	28.06%	26.02%	7.37%	1.28%	0.55%

Operating System Market Share in Ireland - September 2018



Android	Windows	iOS	OS X	Unknown	Linux
40.85%	36.23%	13.54%	5.95%	1.31%	0.78%

Operating System Market Share Worldwide - September 2018

Modes of Operation

1) Single-user

a. Single task single task (palm OS) Only one user using the device accessing one item at a time

b. Single-user multi task, designed with a single user in mind but can deal with many applications running at the same time.

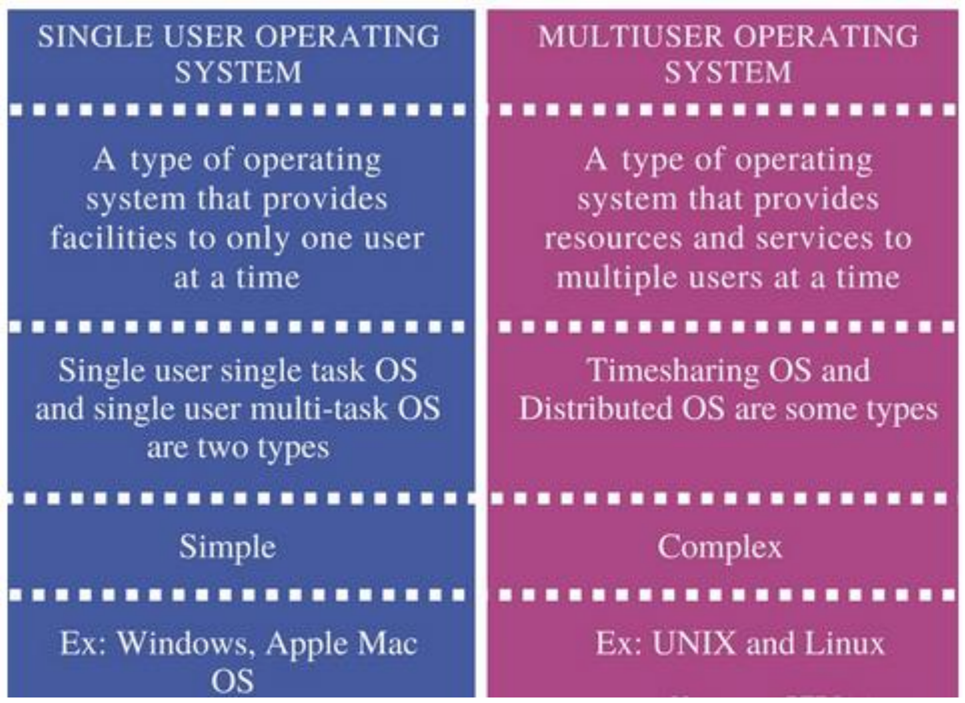
2) Multi-user multi-tasking

The main difference between single user and multiuser operating system is that in a single user operating system, only one user can access the computer system at a time while in a multiuser operating system, multiple users can access the computer system at a time.

Multi-tasking isn't just about running more than one application at the same time. Multi-tasking allows multiple tasks to run concurrently, taking turns using the resources of the computer. This can mean running a couple of applications, sending a document to the printer and downloading a web page.

A multi-access (or multi-user) system is one where several users can use the same system together via a LAN. With multi-user systems, many users can take advantage of the computers resources simultaneously (such as time sharing systems and internet servers). Modern personal computers can allow multi-user access.

The CPU deals with users in turn in multi-user systems; clearly the more users, the slower the response time. Generally, however, the processor is so fast the user feels they are being dealt with immediately even with the delay in response time.



SINGLE USER OPERATING SYSTEM	MULTIUSER OPERATING SYSTEM
A type of operating system that provides facilities to only one user at a time	A type of operating system that provides resources and services to multiple users at a time
Single user single task OS and single user multi-task OS are two types	Timesharing OS and Distributed OS are some types
Simple	Complex
Ex: Windows, Apple Mac OS	Ex: UNIX and Linux

Evolution of Operating Systems

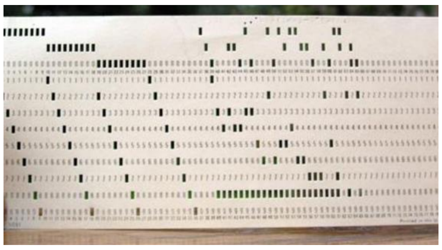
In computer hardware, **generations** have been marked by major advances in componentry from vacuum tubes (first **generation**), to transistors (second **generation**), to integrated circuitry (third **generation**), to large-scale and very large-scale integrated circuitry (fourth **generation**). The successive hardware generations have each been accompanied by dramatic reductions in costs, size, heat emission, and energy consumption, and by dramatic increases in speed and storage capacity.

Originally the main role of an OS was to help various applications interact with the computer hardware. The OS provide a necessary set of functions that allow software packages to control the computer's hardware.

The beginning was when the cost of computer hardware was so high that computer time was allotted very carefully.

First generation computers were quiet slow and did not really need an OS as the machines could not handle multiple concurrent tasks. Human operators were the task managers.

Users had complete access to the machine language. They hand-coded all instructions. Users used punch cards to sign up for time on the machine, running the machine in single-user, interactive mode.



Before loading your program, you first had to load the compiler and continue from there. A great deal of time was lost between the completion of one job and the initiation of the next.

Batch processing

Human operator hand coding was a very wasteful time – for the users and the potential of the computer too! In an effort to make the hardware usable to more people, **batch processing** was introduced.

- Batch processing is the execution of a series of programs ("jobs") on a computer without manual intervention. Jobs were set up so they can be run to completion without human interaction. Batch processing was introduced as a means to keep the CPU busy

2nd generation computers were built with transistors and therefore this resulted in an increase in speed and CPU capacity and this made punched card batch processing increasingly less efficient. Card readers simply could not keep the CPU busy. Magnetic tape offered one way to process desks faster.

.

Early operating systems supported a single process with a single thread at a time (*single tasking*). They ran batch jobs (one user at a time).

At that time computers (c.1950s) were large machines that used:

- Punch cards for input
- Line printers for output
- Magnetic tape for storage

This method of operation lends itself to jobs with similar inputs, processing and outputs where no human intervention is needed. Jobs are stored in a queue until the computer is ready to deal with them. Often batch processed jobs are done overnight. Typically examples might include the processing of payrolls, electricity bills, invoices and daily transactions.

Time-sharing systems/ Multitasking Operating Systems

The next generation of computers marked the introduction of IC's and again, this resulted in an increase of speed. Time-sharing was introduced.

In the late 1960s, **multiprogramming systems** were established and continuing to the present day. This allows several executing programs to be in memory concurrently – a multiprogramming environment that's also interactive with the user (user could interface directly with the computer through typewriter-like terminals). This is achieved by cycling through processes, allowing each one to use the CPU for a specific slice of time. The software that was used initially to handle the jobs were known as **monitors**. Virtual memory and multiprogramming necessitated a more sophisticated monitor which evolved into what we now call an operating system.

In multiprogramming systems several user programs are in main storage at once and the processor is switched rapidly between the jobs. In multiprocessing systems, several processors are used on a single computer system to increase the processing power of the machine.

Although OS relieved programmers and operators of a significant amount of work, users wanted closer interaction with computer and batch jobs were not appealing!

- A timesharing user could locate and correct errors in seconds or minutes, rather than suffering the delays, often hours or days, in batch processing environments.
- Timesharing systems allowed users to submit their own jobs, interactively and get immediate feedback

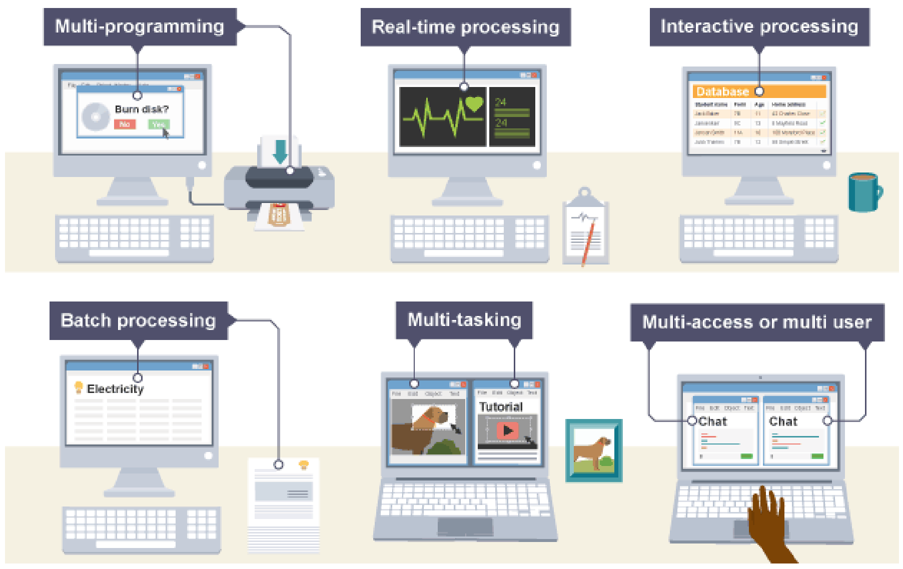
Time Sharing Systems required a more complex operating system.

They had to do scheduling which is allocating resources to different programs and deciding which program should use which resource and when. During this era, a new term was coined which is a **process**.

A ***job***: is a program to be run.

A ***process***: is a program that is in memory and waiting for resources. Many of these can co-exist in memory with one being executed at one time. Each process gets assigned a unique Process ID (PID)

Terminals were connected to systems that allowed access by multiple concurrent users. Interactive programming facilitated time-staring (time slicing) where the CPU switches between user sessions very quickly, giving each user a small slice of processor time.



Real-Time OS

Real-time systems emerged in which computers were used to control industrial processes such as gasoline refining. Military real-time systems were developed to monitor thousands of points at once for possible enemy air attacks. An OS that needs to be **time** deterministic, predictable, has to react within a guaranteed time to an input is a Real Time Operating System (RTOS).

A RTOS does not necessarily have to be fast. It simply has to be quick enough to respond to inputs in a predictable way. For example, the engine management system within a car uses a real-time operating system in order to react to feedback from sensors placed throughout the engine, traffic control, patient monitoring and military control systems. Embedded computers often contain an RTOS as many are used to control something.

A RTOS is a software component that rapidly switches between tasks, giving the impression that multiple programs are being executed at the same time on a single processing core.

In actual fact the processing core can only execute one program at any one time, and what the RTOS is actually doing is rapidly switching between individual programming threads (or *tasks*) to give the impression that multiple programs are executing simultaneously.

When switching between Tasks the RTOS has to choose the most appropriate task to load next. There are several scheduling algorithms available, including Round Robin, Co-operative and Hybrid scheduling. However, to provide a responsive system most RTOS's use a pre-emptive scheduling algorithm (Algorithms discussed in more detail later)

Minimisation of circuitry also saved space on chip size allowing more room for circuits. For example when address spaces increased from 16 to 32 bits, kernel design was no longer constrained by the hardware architecture, and kernels began to grow larger.

4th generation technology, VLSI, allowed the PC market to flourish and network OS and distributed OS are a result of this technology.

Distributed systems

Networking and internetworking have created a new dimension in operating systems. A job that was previously done on one computer can now be shared between computers that may be thousands of miles apart. A program can be run partially on one computer and partially on another if they are connected through a network. Resources can be distributed too.

Distributed Systems are loosely coupled systems that communicate via message passing. Advantages include resource sharing, speed up, reliability, communication. Distributed systems combine features of the previous generation with new duties such as controlling security.

Operating Systems Architecture

A key principle of a distributed system is openness and with this in mind, the 3 types of Operating Systems commonly used nowadays are:

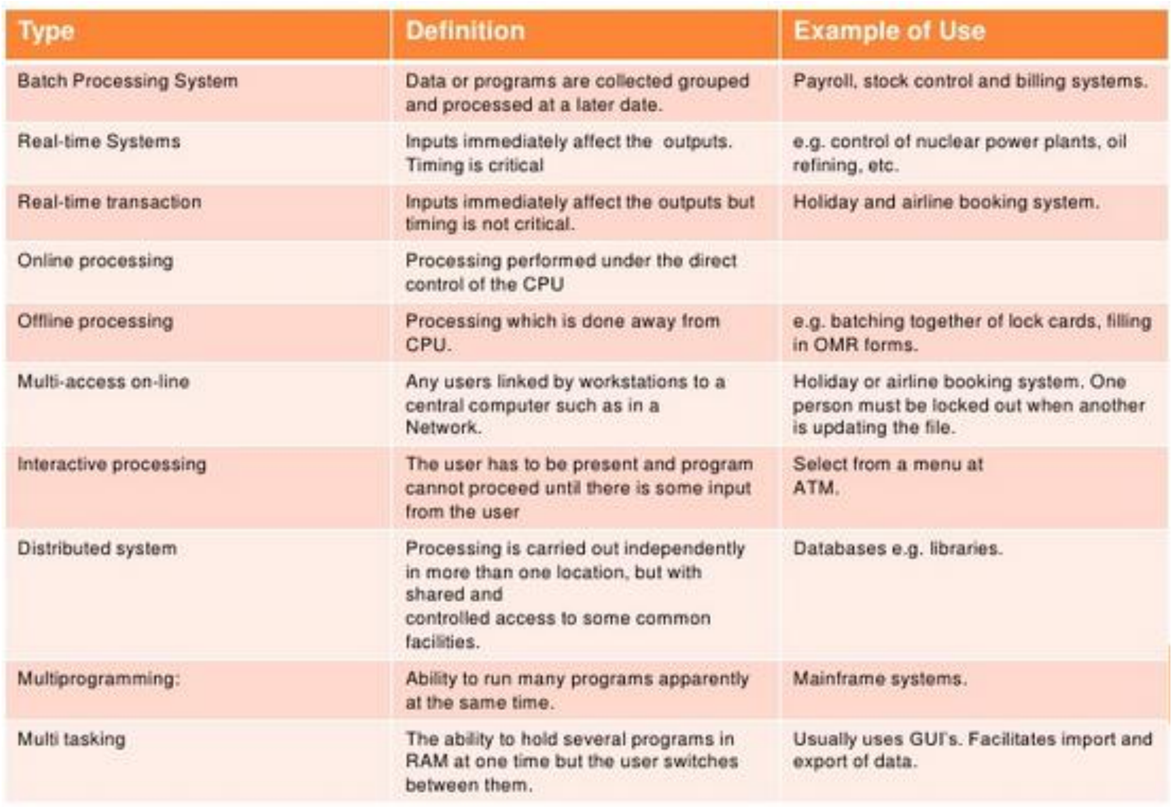
1. **Monolithic OS**, where the entire OS is working in kernel space and is alone in supervisor mode; A monolithic structure means the kernel (all OS's have them) are set up so that the kernel does everything. This was the way it was first done in Linux - back when the machines own kernel had to be compiled if there was support required for sound (for example). As kernels got more sophisticated, they became too large to go with the "one size fits all" things, so they evolved to use modules, that could be added in whenever needed, and to accommodate more complex and sophisticated hardware. Linux and Android are pure monoliths.
2. **Layered/Modular OS**, in which some part of the system core will be located in independent files called modules that can be added to the system at run time; and
3. **Microkernel OS**, where the kernel is broken down into separate processes, known as servers. Some of the servers run in kernel space and some run in user-space. Microkernel is the near-minimum amount of software that can provide the mechanisms needed to implement an OS

The first two may be put under the same category as monolithic.

Simplicity, flexibility and high performance are crucial for an OS.

Parallel systems

This involves multiple CPUs on the same machine. Each CPU can be used to one program or a part of a program, which means that many tasks can be accomplished in parallel instead of serially. The OS required for this are more complex than those that support single CPUs.

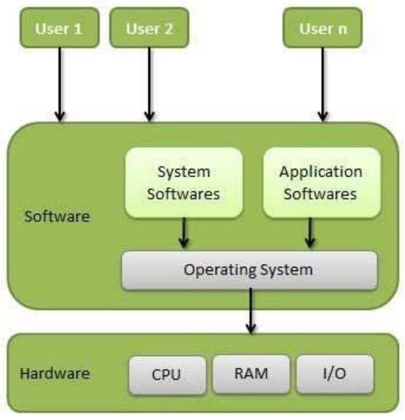


Type	Definition	Example of Use
Batch Processing System	Data or programs are collected grouped and processed at a later date.	Payroll, stock control and billing systems.
Real-time Systems	Inputs immediately affect the outputs. Timing is critical	e.g. control of nuclear power plants, oil refining, etc.
Real-time transaction	Inputs immediately affect the outputs but timing is not critical.	Holiday and airline booking system.
Online processing	Processing performed under the direct control of the CPU	
Offline processing	Processing which is done away from CPU.	e.g. batching together of lock cards, filling in OMR forms.
Multi-access on-line	Any users linked by workstations to a central computer such as in a Network.	Holiday or airline booking system. One person must be locked out when another is updating the file.
Interactive processing	The user has to be present and program cannot proceed until there is some input from the user	Select from a menu at ATM.
Distributed system	Processing is carried out independently in more than one location, but with shared and controlled access to some common facilities.	Databases e.g. libraries.
Multiprogramming:	Ability to run many programs apparently at the same time.	Mainframe systems.
Multi tasking	The ability to hold several programs in RAM at one time but the user switches between them.	Usually uses GUI's. Facilitates import and export of data.

Figure 1: Evolution of OSs

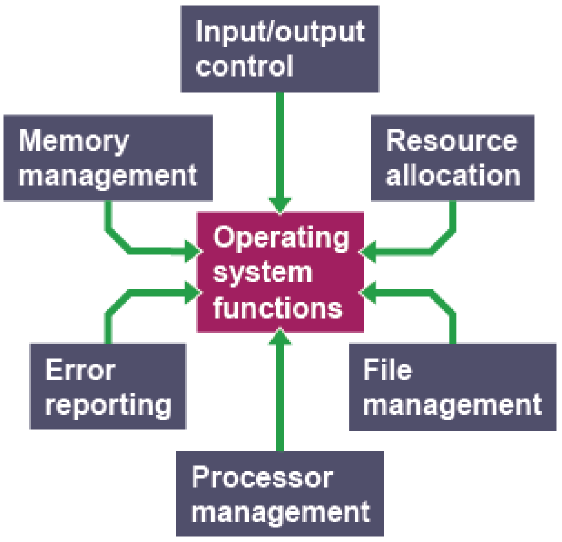
Users' expectations of OSs have changed considerably through the years. Users assume an OS will make it easy for them to manage the system and its resources and this expectation has resulted in "drag and drop" file management, as well as "plug and play" device management.

From the programmer's perspective, the OS hides the details of the system's lower architectural levels. As well as providing an interface to the programmer, it also acts as a layer between the application software and the actual hardware of the machine (the interface in UNIX is called a **shell** and, as you know at this stage from your Shell Scripting Assessment!), can be accessed via Command Line Interface (CLI), and in others is called a **window** as it is menu driven and has a GUI component.



The OS is, in essence, a virtual machine that provides an interface from hardware to software. It deals with real devices and real hardware so the application programs and users don't have to.

Some other responsibilities of an OS include:



- Allows convenient usage for the user by providing an interface between the user and computer;
- Allows efficient usage of the computer;
- Allows parallel activity, avoids wasted cycles
- Provides information protection, security and access rights of users
- Gives each user a slice of the resources by organising processor time
- Deals with the transfer of data and/or information in/out of memory and initially (on "power-on") loading other programs from memory for execution
- Acts as a control program for the secondary memory and I/O devices

The OS itself is little more than an ordinary piece of software. It differs from most other software in that it is loaded by booting the computer and then is executed directly by the processor.

The Hardware

A microprocessor, sometimes called a logic chip, is a computer processor on a microchip. The microprocessor contains all, or most of, the central processing unit (CPU) functions and is the "engine" that goes into motion when you turn your computer on.

A microprocessor is designed to perform arithmetic and logic operations that make use of small number-holding areas called registers.

Typical microprocessor operations include adding, subtracting, comparing two numbers, and fetching numbers from one area to another. These operations are the result of a set of instructions that are part of the microprocessor design

Microprocessor architectures

"Instruction set architecture (ISA) is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine" **IBM introducing 360 (1964)**

There are certain instructions that the CPU knows and when we give them those instructions, different transistors inside it switch ON and OFF to perform those instructions.
The instructions that we input are in the form of 1's and 0's, or ***opcode***. Since it is hard for us to remember combinations of 1's and 0's, we tend to use shorthand's for those instructions, called assembly language, and a compiler converts it into opcode. The number of instructions that a particular CPU can have is limited and the collection of all those instructions is called the Instruction Set.

A proper design of hardware and instruction set can determine how fast the CPU is.

CPU Performance

The performance of a CPU is the number of programs it can run in a given time.

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

The performance is determined by:

- the number of instructions that a program has:
  - more instructions = more time to perform them.
- the number of cycles (clock cycles) per instructions.

Optimising CPU performance in done via the instruction set and the hardware of the CPU.

The Microprocessor without Interlocked Pipeline Stages (MIPS) microprocessor paradigm was created in the early 1980's by researchers Stanford University. Since that time, the MIPS paradigm has been so influential that nearly every modern-day processor family makes some use of the concepts derived from that original research.

MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA). MIPS is a register based architecture, meaning the CPU uses ***registers*** to perform operations on.
Registers are memory just like RAM, except registers are much smaller than RAM, and are much faster. In MIPS the CPU can only do operations on registers, and special immediate values.
The early MIPS architectures were 32-bit, with 64-bit versions added later. With some registers reserved, a fair number of registers however are available for use. For example, one of these registers, the *program counter*, contains the memory address of the next instruction to be executed. As the processor executes the instruction, the program counter is incremented, and the next memory address is fetched, executed, and so on <<recall fetch-decode-execute cycle>>.

RISC

The MIPS processor is a **RISC** (Reduced Instruction Set Computer) processor (i.e. MIPS is an example of an early RISC architecture). RISC is a type of microprocessor architecture that utilises a small, highly-optimised set of instructions.
As a RISC architecture, MIPS doesn't assign individual instructions to complex, logically intensive tasks. This is in contrast to its predecessor, the CISC architectures.

CISC

Compared with their **CISC** (Complex Instruction Set Computer) counterparts (such as the Intel Pentium processors), RISC processors typically support fewer and much simpler instructions.
The CISC architecture tries to reduce the number of Instructions that a program has, thus optimising the Instructions per Program part of the above equation. This is done by combining many simple instructions into a single complex one.

In a CISC style instruction, the CPU has to do more work in a single instruction, so clock speeds are slightly slower. Moreover, the number of general purpose registers are less as more transistors need to be used to decode the instructions.
E.g. **MUL 1200, 1201**

> This instruction takes two inputs: the memory location of the two numbers to multiply, it then performs the multiplication and stores the result in the first memory location.
> Where MUL takes the value from either two memory locations (say 1200 and 1201) or two registers, finds their product and stores the result in location 1200.

MIPS and other RISC architectures were based on the philosophy that, among other things, by only implementing a small core of the most common instructions, architects could simplify the design and speed up the majority of common instructions so much that the cost of implementing complex programs as multiple instructions would be hidden.

Generally, a single instruction in a RISC machine will take only one CPU cycle.

Multiplication in a RISC architecture cannot be done with a single MUL like instruction as above.

Instead, we have to first load the data from the memory using the LOAD instruction, then multiply the numbers, and the store the result in the memory.

Load A, 1200
Load B, 1201
Mul A, B
Store 1200, A

> Here the Load instruction stores the data from a memory location like 1200 into a register A or B. Mul multiplies values in the two registers stores it in A. Then finally we store the value of A in 1200 (or any other memory location). Note that in RISC architectures, we can only perform operations on Registers and not directly on the memory. These are called addressing modes.

This might seem like a lot of work, but in reality, since each of these instructions only take up one clock cycle, the whole multiplication operation is completed in fewer clock cycles

Since RISC has simpler instruction sets, complex High-Level Instructions needs to be broken down into many instructions by the compiler. While the instructions are simple and don't need complex architectures to decode, it is the job of the compiler to break down complex high level programs into many simple instructions.

The premise is, however, that a RISC processor can be made much faster than a CISC processor because of its simpler design. While CISC tries to complete an action in as few lines of assembly code as possible, RISC tries to reduce the time taken for each instruction to execute. These days, it is generally accepted that RISC processors are more efficient than CISC processors.

Even the popular CISC processor that is still around (Intel Pentium) internally translates the CISC instructions into RISC instructions before they are executed
RISC takes less cycles to execute but its primary drawback is its code density
CISC is most often used in automation devices whereas RISC is used in video and image processing applications (RISC devices as they are faster and less resource and power hungry).

While many Intel CPU's are CISC architecture based, all Apple CPUs and ARM (Advanced RISC Machine) devices have RISC architectures. Because RISC OS is written specifically for ARM, it can run on all kinds of small-board computers, including every model of Raspberry Pi.

ARM is the processor architecture of many phones – are you carrying a RISC in your pocket?

Threads and Processes

- ***Thread***: a sequential execution stream

  Executes a series of instructions in order (only one thing happens at a time).

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

- ***Execution state***: everything that can affect, or be affected by, a thread:

  Code, data, registers, call stack, open files, network connections, time of day, etc.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

- ***Process***: one or more threads, along with their execution state.

  Part is shared among all threads in the process

  Part of the process state is private to a thread

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

Advantages of Thread:

- Threads minimise the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilisation of multiprocessor architectures to a greater scale and efficiency.

Threads are implemented in the following 2 ways:

1. **User Level Threads:** User managed threads.
2. **Kernel Level Threads:** Operating System managed threads acting on kernel, an operating system core.

Context Switch

The procedure of switching between processes is called **context switching** and the OS performs this quickly, in essence giving the user a personal virtual machine - bearing in mind that there will be context switch overtime with **dispatcher** latency when it needs to save details on process in the **Process Control BlockError! Reference source not found.** and load details of other processes

During a context switch, all pertinent information about the currently executing process must be saved so that when the process is scheduled to use the CPU again, it can be restored to the exact state in which it was interrupted. This required the OS to know all the details of the hardware.

Page tables and other information associated with virtual memory must be saved during a context switch. CPU registers must also be saved during this context switch because they contain the current state of the executing process.

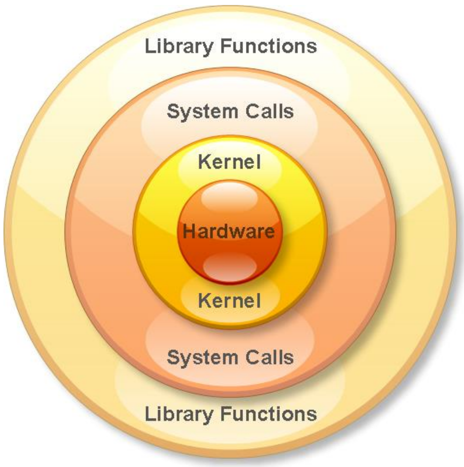
Context switches take resources and time and therefore the OS must deal with them quickly and efficiently.

What about when the OS wants to execute an instruction by a process?

The OS needs to be able to respond to the event. The operating system kernel is the part of the operating system that responds to

1) System calls,
2) Interrupts and
3) Exceptions

1) System Calls



System calls are the interface between processes and the kernel. They are the instructions that allow access to privileged or sensitive resources on the CPU.

A process uses system calls to request operating system services.

From point of view of the process, these services are used to manipulate the abstractions that are part of its execution environment. For example, a process might use a system call to

- open a file
- send a message over a pipe
- create another process
- increase the size of its address space

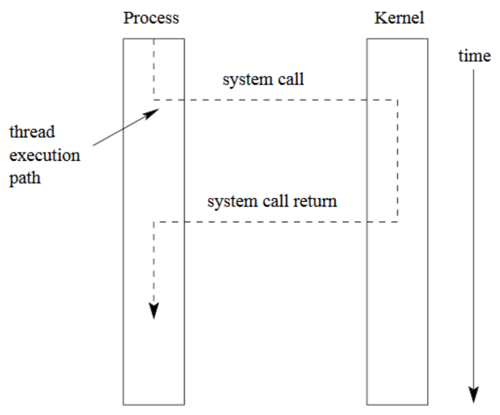
How the System Calls Work

The hardware provides a mechanism that a running program can use to cause a system call. Often, it is a special instruction, e.g., the **MIPS syscall** instruction.

What happens on a system call?

- the processor is switched to system (privileged) execution mode
- key parts of the current thread context, like the program counter and the stack pointer, are saved
- the thread context is changed so that:
  - the program counter is set to a fixed (determined by the hardware) memory address, which is within the kernel's address space
  - the stack pointer is pointed at a stack in the kernel's address space
- Once a system call occurs, the calling thread will be executing a system call handler, which is part of the kernel, in system mode.
- The kernel's handler determines which service the calling process wanted, and performs that service.
- When the kernel is finished, it returns from the system call. This means:
  - restore the key parts of the thread context that were saved when the system call was made
  - switch the processor back to unprivileged (user) execution mode
- Now the thread is executing the calling process' program again, picking up where it left off when it made the system call.

System Call Diagram



2) Exceptions

Exceptions are a way that control is transferred from a process to the kernel.

Exceptions are conditions that occur during the execution of an instruction by a process. For example:

- arithmetic error, e.g. overflow (recall this from the half-adder)
- illegal instruction
- memory protection violation
- page fault

Exceptions are detected by the hardware

When an exception occurs, control is transferred (by the hardware) to a fixed address in the kernel (e.g. 0x8000 0080 on MIPS & OS/161) Transfer of control happens in much the same way as it does for a ***system call***.

In fact, a system call can be thought of as a type of exception, and they are sometimes implemented that way (e.g., on the MIPS).

In the kernel, an exception handler determines which exception has occurred and what to do about it. For example, it may choose to destroy a process that attempts to execute an illegal instruction.

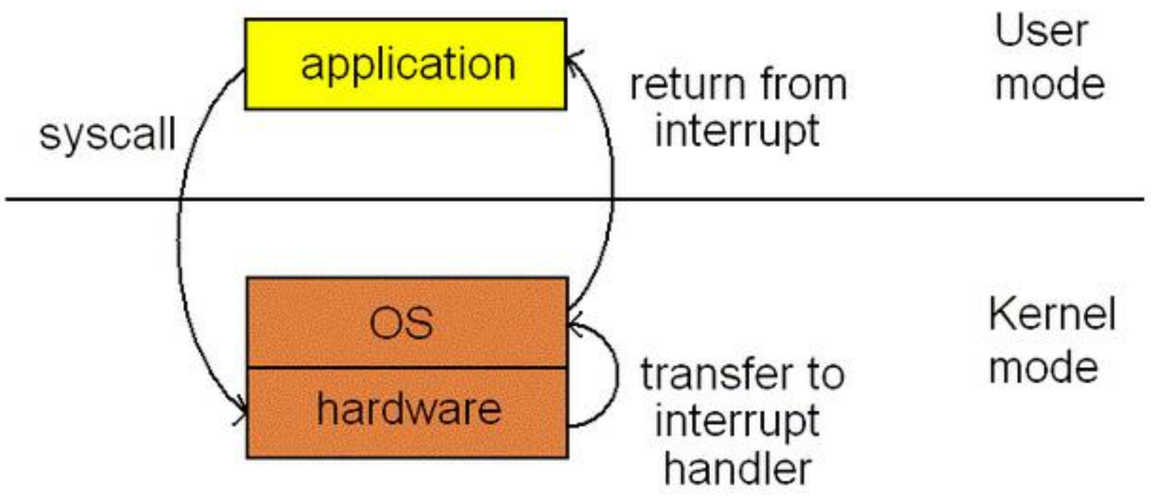
3) Interrupts

Interrupts are a third mechanism by which control may be transferred to the kernel

Interrupts are similar to exceptions. However, they are caused by hardware devices, not by the execution of a program. For example:

- a network interface may generate an interrupt when a network packet arrives
- a disk controller may generate an interrupt to indicate that it has finished writing data to the disk
- a timer may generate an interrupt to indicate that time has passed

Interrupt handling is similar to exception handling - current execution context is saved, and control is transferred to a kernel interrupt handler at a fixed address



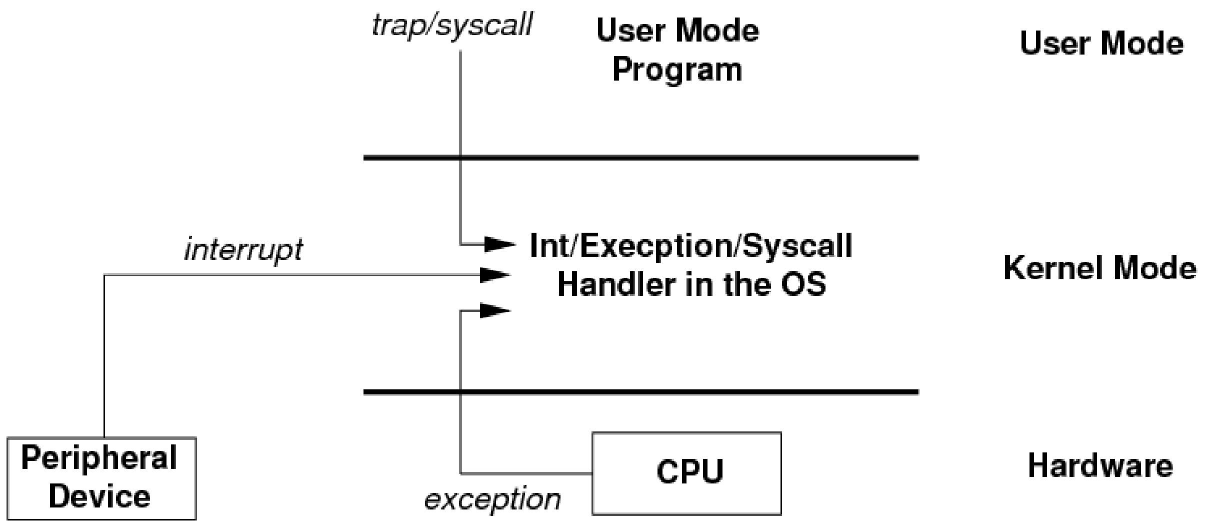
Summary of Hardware Features Used by the Kernel

Interrupts and Exceptions, such as timer interrupts, give the kernel the opportunity to regain control from user programs.

Memory management features, such as memory protection, allow the kernel to protect its address space from user programs.

Privileged execution mode allows the kernel to reserve critical machine functions (e.g, halt) for its own use.

Independent I/O devices allow the kernel to schedule other work while I/O operations are on-going.



Modern kernels (Linux, FreeBSD) have a modular design, which offers run-time adding and removal of services.

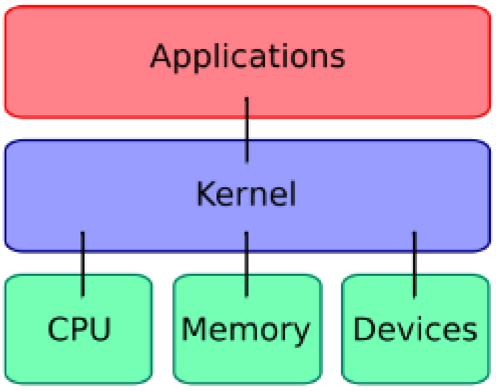
For now, think of the kernel as a program that resides in its own address space, separate from the address spaces of processes that are running on the system.

What constitutes an Operating System?

- A kernel
- System programs
- Application programs

Kernel:

The kernel is a program. It has code and data like any other program.



An OS runs in one of two modes

1) **User mode** where normal programs are executed.
2) **Kernel mode**, or **supervisor mode**, or protected mode. Usually kernel code runs in kernel mode, while the rest of the operating system does not.

If every program had unrestricted access to the CPU, main memory and the peripheral devices, all concepts of separation of programs and the data in memory, on disk etc. would not exist.

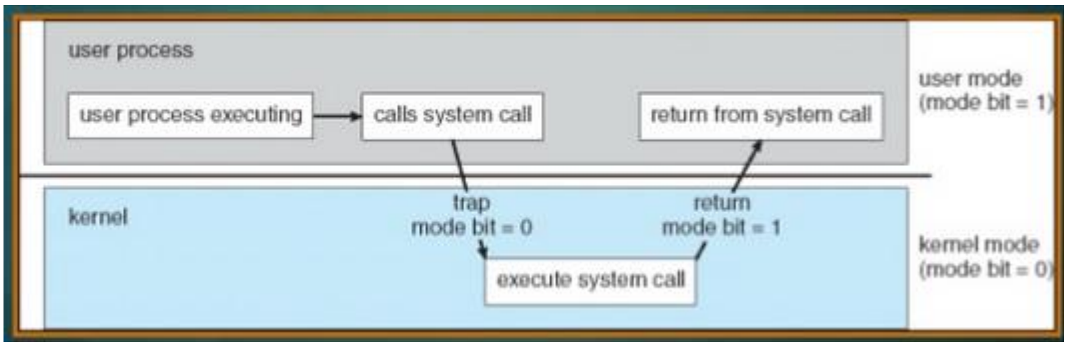
- A program could look at all memory locations, including that of other programs, as well as read all the data on all of the attached disks, and read all the data being sent across the network.

In **kernel mode**, the CPU has instructions to manage memory and how it can be accessed, plus the ability to access peripheral devices like disks and network cards. The CPU can also switch itself from one running program to another.

In **user mode**, access to memory is limited to only some memory locations, and access to peripheral devices is denied. The ability to keep or relinquish the CPU is removed, and the CPU can be taken away from a program at any time.

The whole kernel runs in "kernel mode", a processor mode in which the software has full control over the machine.

The processes running on top of the kernel run in "user mode", in which programs have only access to the kernel services



In brief, the operating system creates a process by:

- Creating and initialising the process control block.
- Load code and data into memory.
- Create first thread with call stack.
- Provide initial values for "saved state" for the thread
- Make thread known to dispatcher; dispatcher "resumes" to start of new program.

What does it mean to run in privileged mode?

Kernel uses privilege to

- control hardware
- protect and isolate itself from processes

Privileges vary from platform to platform, but may include:

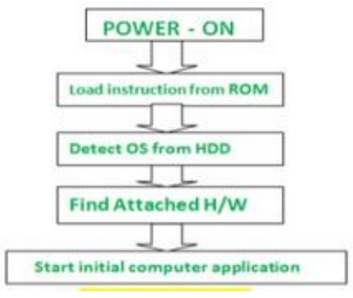
- ability to execute special instructions (like halt)
- ability to manipulate processor state (like execution mode)
- ability to access virtual addresses that can't be accessed otherwise

Kernel ensures that it is isolated from processes. No process can execute or change kernel code, or read or write kernel data, except through controlled mechanisms like **system calls**.

The OS is responsible for loading other programs into memory for execution. However, the OS itself is a program that needs to be loaded into memory and be run. How is this dilemma solved?

Bootstrap process: Firmware

Early computers would have hardware that would enable the operator to press a button to load a sequence of bytes from punched cards, punched paper tape, or a tape drive



During the boot process, the operating system kernel is loaded into RAM –The kernel provides essential OS services.

• The CPU boots in kernel mode, with full access to the system hardware. It then proceeds to load and start the operating system running.

• At some point, the operating system prepares a program to run, e.g. by loading the instructions from the disk, setting up the memory for the program etc.

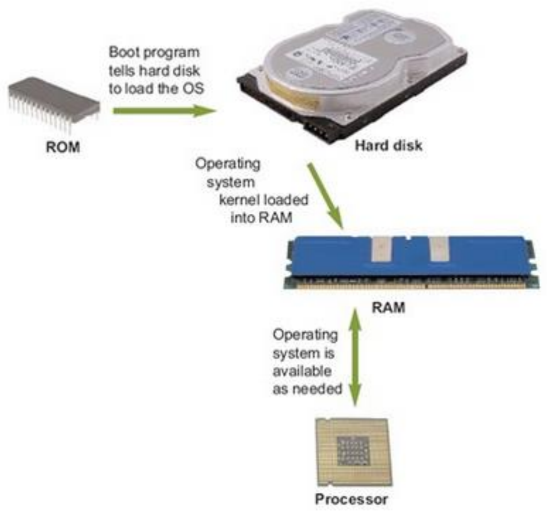
• Just before jumping to the first instruction, the CPU lowers its privilege level by marking the mode flag as user mode.

• Thus, when the program starts execution, it is already in user mode, and is limited in what hardware operations and accesses it can perform.

A very small piece of memory is made of ROM and holds a small program built into the special ROM circuitry called the **bootstrap program** (**Firmware**).

When the computer is turned on, the CPU counter is set to the first instruction of this bootstrap program and executes the instructions in this program. When loading is done, the program counter is set to the first instruction of the operating system RAM.

Both basic input/output system (BIOS) and Unified Extensible Firmware Interface (UEFI) are types of firmware for computers.


Boot program tells hard disk to load the OS
ROM
Hard disk
Operating system kernel loaded into RAM
RAM
Operating system is available as needed
Processor

BIOS-style firmware is (mostly) only ever found on IBM PC compatible computers.

The BIOS takes care of the basic initialisation processes in a PC; it boots (starts) the hardware and makes the first checks on it to make sure the hardware is in good health, the keyboard is connected, the RAM is ready to go and then BIOS lets the OS takes over

On your BIOS PC, you have one or more disks which have a Master Boot Record (MBR). The MBR is another de facto standard; basically, the very start of the disk describes the partitions on the disk in a particular format, and contains a ‘boot loader’, a very small piece of code that a BIOS firmware knows how to execute, whose job it is to boot the operating system(s)

All a BIOS firmware knows, in the context of booting the system, is what disks the system contains. You, the owner of this BIOS-based computer, can tell the BIOS firmware which disk you want it to boot the system from. The firmware has no knowledge of anything beyond that. It executes the bootloader it finds in the MBR of the specified disk, and that’s it. The firmware is no longer involved in booting i.e. they can look for an MBR on a disk, and execute the boot loader from that MBR, and leave everything subsequently up to that bootloader.

Advancements in technology empowered the OS much more than expected and pushed the BIOS to the initial system boot, with the OS handling most of the operations

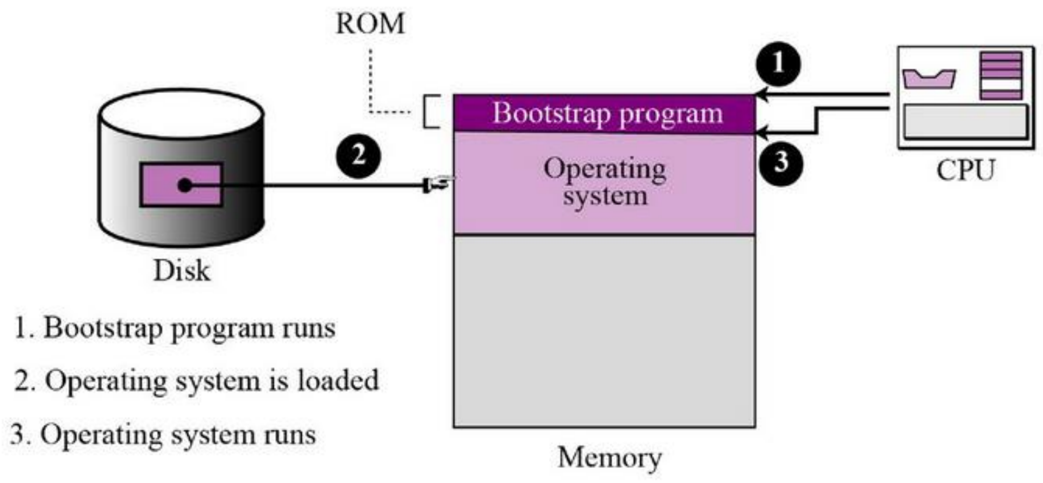
Chipmaker Intel revealed that by 2020 it will phase out the last remaining relics of the PC BIOS. This will mark a permanent shift to Unified Extensible Firmware Interface (UEFI) firmware.

UEFI is *completely* different to BIOS. UEFI is managed by the UEFI Forum. Microsoft is a member of the UEFI forum. So is Red Hat, and so is Apple, and so is just about every major PC manufacturer, Intel, AMD, and so on

The UEFI also connects a computer's firmware to its OS like BIOS does. It is also installed when the computer is made and is the first program that goes live when you switch on a computer. And it has many advantages over BIOS

- *UEFI* can run in 32-bit or 64-bit mode and has more addressable address space than BIOS, which means your boot process is faster
- It can handle large hard disk partitions (it is a micro OS. It is "programmable", and this facility enables PC makers add applications and drivers to it)

The bootstrap program is only responsible for loading the OS itself, or that part of it required to start up the computer, into RAM memory.



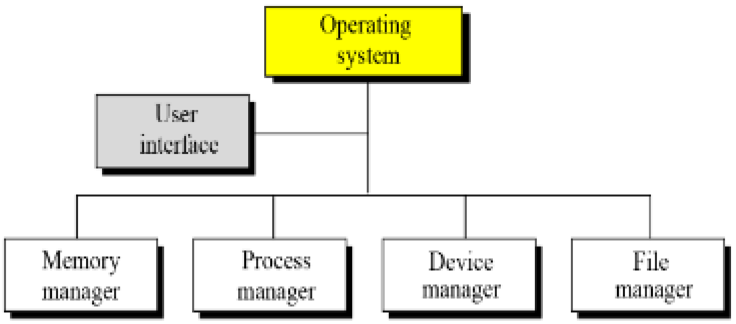
PC hardware made the switch to UEFI with Intel's Sandy Bridge processors, which were introduced in 2011. Now, most systems have UEFI.

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

OS major Duties/Responsibilities

A modern operating system has at least four duties:

1. process manager,
2. memory manager
3. device manager and
4. file manager.



We will investigate these in the next lecture.