What is an Operating System?

- An OS is just a program – like any other programs that you write.
- But it's a special program.
  - Acts as an intermediary between a user of a computer and the computer hardware
  - Used a lot by all the users.
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

OS is part of computer system

- Computer system can be divided into four components:
  - Hardware – provides basic computing resources
    - CPU, memory, I/O devices
  - Operating system
    - Controls and coordinates use of hardware among various applications and users
  - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - Users
    - People, machines, other computers

Roles of Operating System

- OS is a resource allocator
  - Manages all resources *(resources are RAMs/Registers)*
  - Decides between conflicting requests for efficient and fair resource use
- OS is a control program
  - Controls execution of programs to prevent errors and improper use of the computer *(disable programs that try to access illegal operations or improper use of the hardware)*
- Like *government* of a country?

Resource allocator: hardware & I/O control
handle interrupt
manage storage hierarchy
process management
Control program: process management (scheduling & context switch)
security

What's the OS? What's the "kernel"?

- No universally accepted definition of what constitutes OS.
- Some people think of it as "Everything a vendor ships when you order an operating system" – could be useful first approximation
  - But varies wildly *(Kernel is running at all times as long as your computer is on)*
- More precise definition: "The one program running at all times on the computer" is the kernel. Everything else (even shipped by OS vendor) is either a system program or an application program.
- The kernel is *unique*. *|(uninterruptible because you have the right hardware to do it)*
  - Although it's just a program that someone wrote, it *runs* with special privileges – it can do what normal user code cannot do (e.g., access to special instructions & special memory regions).
  - Hardware support required – CPU has (at least) dual mode operation: user (unprivileged) mode vs. kernel (privileged) mode
  - *System calls* let (unprivileged, untrusted) user programs access (privileged, trusted) kernel services

How OS starts to run?

- Bootstrap program is loaded at power-up or reboot of computer
  - Typically stored in ROM or EPROM, generally known as firmware
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution

- BIOS loads the boot loader.
- BIOS is part of the hardware. When you power on your computer, BIOS is loaded first. It scans all the attached devices (eg. network card, RAM, etc) & prepares them to be in a form that the Operating System can use.
- Once BIOS is done with its work, it will load the boot loader. Bootloader is part of the OS and each OS typically has its own bootloader (does the job of loading the OS)

Computer-System Operation

- CPU is not the only component capable of running code or starting activities. IO devices/controllers can also act autonomously.
- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer.
- CPU moves data from/to main memory to/from local buffers.
- I/O is from the device to local buffer of controller.
- CPU/controller needs coordination. E.g., device controller informs CPU that it has finished its operation by causing an *interrupt*.

More about Interrupts

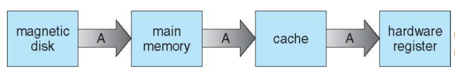
- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines. *interrupts can be interrupted (if high priority) but cannot interrupt a kernel*
- Interrupt architecture must save the address of the interrupted instruction.
- Incoming interrupts are *disabled* while another interrupt (of same or higher priority) is being processed to prevent a *lost interrupt* or *reentrancy* problems.
- A *trap* is a software-generated interrupt caused either by an error or a request by user code. Latter allows a user program to invoke an OS function (system call) and run it in kernel mode. Hence, entry points into kernel are carefully controlled – Why? And why is this important?
- Modern operating system is interrupt driven.

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter.
- Determines which type of interrupt has occurred:
  - polling *polling -> go through all usb controllers and as they called :)*
  - vectored interrupt system *polling takes more time if you alr have the veci*
- Separate segments of code determine what action should be taken for each type of interrupt.
- Interrupt handling is done in *kernel* mode, by the service routine (aka interrupt handler) for the device type.

Migration of Integer A from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy.



- Multiprocessor environment must provide cache *coherency* in hardware such that all CPUs have the most recent value in their cache.

Clustered Systems

- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a storage-area network (SAN)
  - Provides a high-availability service which survives failures
    - Asymmetric clustering has one machine in hot-standby mode
    - Symmetric clustering has multiple nodes running applications, monitoring each other
  - Some clusters are for high-performance computing (HPC)
    - Applications must be written to use parallelization

Operating System Structure

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

Operating System Structure (Cont.)

- Timesharing (multitasking) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing.
  - Response time should be < 1 second
  - Each user has at least one program executing in memory ⇨ process
    - A process is a *running program* – how are the two different?
  - If several jobs ready to run at the same time ⇨ CPU scheduling
  - If processes don't fit in memory, swapping moves them in and out to run
  - Virtual memory allows execution of processes not completely in memory

More on dual-mode operation

- Software error or request creates exception or trap (*software* interrupt)
  - Division by zero, request for operating system service
- Other (buggy) user process problems include infinite loop, processes modifying each other or OS
- Dual-mode operation allows OS to protect itself and other system components from (possibly buggy) user processes
  - User mode and (privileged) kernel mode
  - Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions only executable in kernel mode (examples?)
    - Some parts of memory inaccessible from user mode (examples?)
    - System call changes mode to kernel, return from call (via return-from-trap or RETT instruction) resets it to user
    - Hardware interrupts (e.g., by IO devices) of user process also change processing to kernel mode

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources.
- Single-threaded process has one program counter specifying location of next instruction to execute.
  - Process executes instructions sequentially, one at a time, until completion.
- Multi-threaded process has one program counter per thread.
- Typically system has many processes (some user, some operating system) running concurrently on one or more CPUs.
  - Concurrency by multiplexing the CPUs among the processes / threads.

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Other Important OS Subsystems

- Memory management
- Storage management (including file systems)
- Mass storage management
- IO management
- Plus …

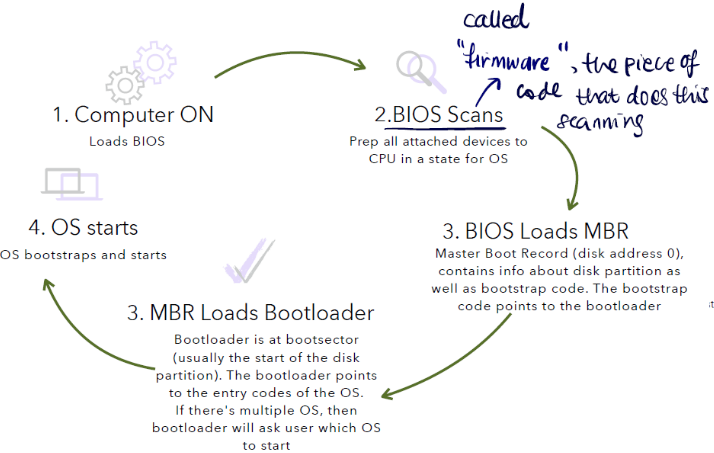
Purposes of an OS

1. Hardware & I/O Control
Controls & coordinate use of hardware among various applications (programs) and users
2. Handles interrupts
Vectored: upon interrupt, will look at interrupt vectors to see which device made the interrupt
Polling: go through each device & check who made the request
If not in kernel mode, you can interrupt an interrupt if u hav higher PRIORITY. However, this comes at a cost of context switch
3. Managing the Storage Hierarchy
so that data integrity & consistency is guaranteed.
Registers>Cache>Main Memory>Secondary storage>Disk controllers>Disks
Main mem: resource allocation, garbage collection
4. Multiprogramming: process management (scheduling & context switch)
Clustered system: High service availability & must write programs that utilises parallel programming
Why Multiprogramming?
gives the illusion that your machine can multitask = being efficient
(1. Be efficient in organising/scheduling jobs, since CPU can only execute 1 inst/clk cycle
2. Allows timesharing: context switch so rapidly that users still see it as interactive computing)
NOTE: but not too rapid cus context switch is an OVERHEAD (CPU cannot do actual work when doing context switch)
Kernel owns process table -> which might be on cache/RAM/disk ("swap space")
Context switch (1. save all states (reg,cache,stack,etc) of Pi 2. Load all states of Pj 3. Resume Pj)
When do you switch from Pi to Pj?
> if Pi is waiting for I/O
> the turn for Pi is up
Rules for managing multiple processing:
1. A single program (user) cannot keep CPU busy at all times
2. CPU always have something to do
HOW multiprogramming:
Response time is fast enough, always have at least 1 program active at any time, if RAM is full, swap w disk

System has many processes, multiplexed (takes turn to run)
Concurrency: Multiplex process executions so it has the illusion that they run at the same time
Parallelism: processes executed at the same time at different CPUs (multicore system)

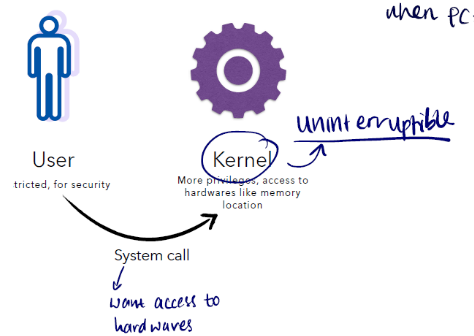
1. When a device controller finishes an IO operation, it interrupts the CPU with a number identifying itself. This method of identifying the interrupting device is called **_vectored_** interrupt, which is more efficient than the alternative paradigm of **_polling_** interrupt.
2. The CPU handles an interrupting device by switching execution to an interrupt handler. This handler runs in **_kernel_** mode.
3. Kernel mode execution is privileged. What are the two major kinds of privileges? Special instructions that can only be executed in kernel mode. Protected memory that is only accessible in kernel mode.
4. What is the difference between a program and a process? How are the two concepts related? Program is static; process is dynamic. A process is a program in execution.
OS is made of codes; Multiple CPUs share the same RAM -> OS needs to manage them

HOW DOES OS RUN?

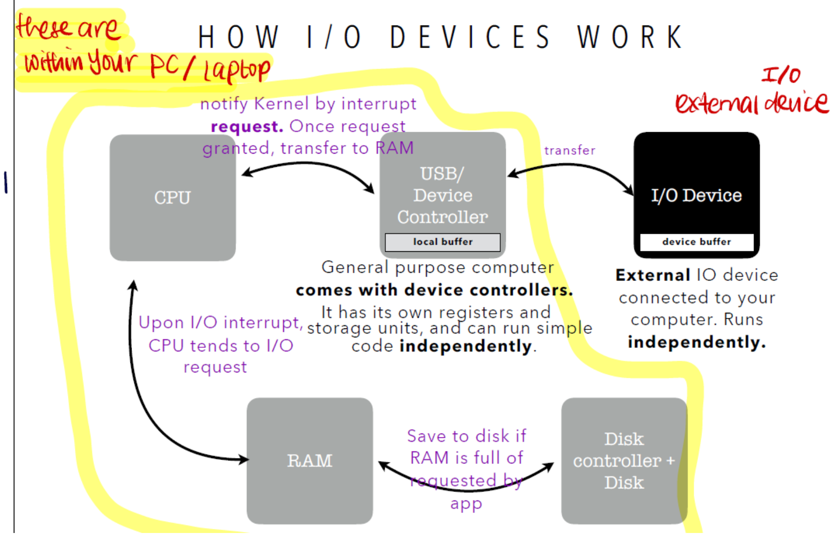


DUAL MODE

dual mode is possible if it is supported by hardware
↳ control unit prevents jmp when PC31 is 1

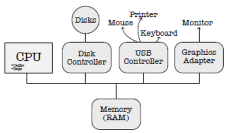


HOW I/O DEVICES WORK

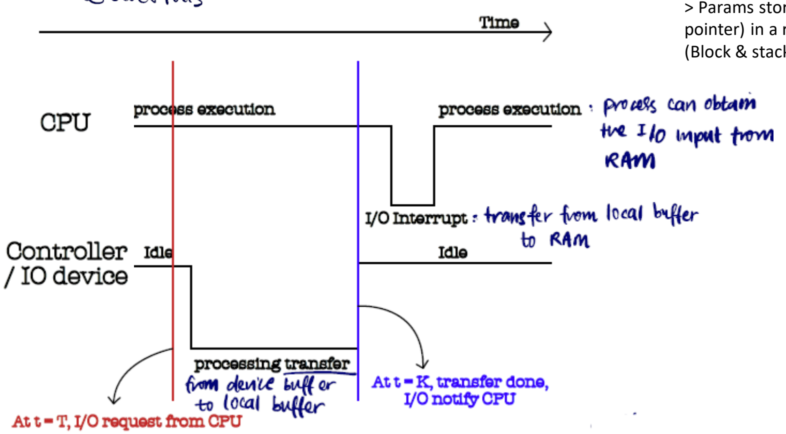


HOW I/O DEVICES WORK

- CPU, I/O device controllers, and I/O devices act **independently** of one another
- Each I/O device has a **controller** (whats why you install *drivers*) that comes with a ***local buffer***
- I/O devices and controllers can run code an starts activity on its own too



(A) CPU wants to move data to/from device controller buffer from/to memory (RAM)
(B) I/O happens when data is moved from/to device controller buffer to/from device
We need **coordination** to do step (A) and (B) above: with *interrupts*
OS does this



OS services (help user use computer)
>User interface >Program execution (load/run/end, errorhandling)
>I/O operations >Communications of processes via shared mem
>File System manipulation (create/del/rename/ r/w /search, manage permission)
>Security&error detectn - handles debugging facilities (isolate/recover from error)
>Resource allocation (how to allocate among multiple jobs concurrently?)
>Accounting (keep track of processes)
System calls (programming interface provided by OS kernel for user to access services) (kernel has access to hardwares)
<u>To make system calls through OS interface:</u>
1. program calls kernel want to request a certain service
2. a system call is associated with a number
3. each number refers to a specific service (I/O, time, date, file op. etc)
Types of system calls: (1 set corr to each OS subsys)
> Process control, file managemt, device managem, info maintenance, comm, protection
System call is like fn calls, need to pass parameters to it
General methods of Parameter Passing:
> Pass in registers (simplest, but may hav more param than reg)
> Param pushed onto stack by program and popped off stack by OS
> Params stored in a block/table, in mem, & addr of block passed as a param (eg C pointer) in a reg
(Block & stack mtd do not limit #params being passed

I/O Devices:
Input: camera, barcode reader, mouse, kb
Output: monitor, printer, speaker
Both: modems, network cards, touch screen
Summary:
OS is software that manages hardware (acts as intermediary b/w hardware(reg,cache) & user programs (browser))
Dual mode of OS (user/kernel)
Functions of OS:
> Resource allocator (allocate memory/cpu time, manage conflicting requests) Hardware & I/O control, handle interrupts, manage storage hier, process managemt
> Control program executions
(prevents illegal access/improper use of hardware/process management(sch,contx swtc)
Heart of OS is kernel (running at all times)

System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
  - *Recall*: it's like interrupt, but it's a *software* interrupt (*trap* instruction)
- System call interface (e.g., documented as Unix/Linux manpages) invokes intended system call in OS kernel and returns status of the system call and any return values
- Caller needs know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call (usage is just like a function or library call)
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler), e.g., libc (C) or JCL (Java)