Lecture 1: Computer Systems Overview

Program Execution

- A program consists of a set of instructions to execute, stored in memory
- The execution of a program is done in two steps – fetch and execute
- Both steps are run continuously until all instructions are completed

Fetch	• Processor (CPU) fetches the instructions from main memory • Program Counter (PC) holds the address of the next instruction • PC is incremented after every instruction fetch
Execute	• Instruction Register (IR) will be loaded with the address after fetching • Processor interprets the instruction and executes the instruction • The interpretation is done through opcode

Interrupts

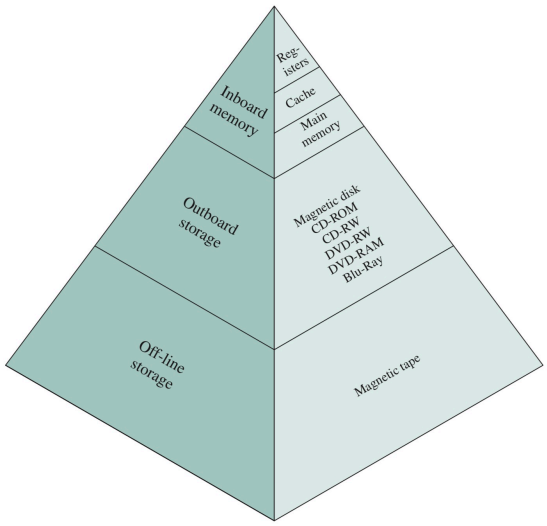
- Main purpose is to "pause" the process of the execution of instructions, which is called "interrupts"
- In other words, the usual sequencing of processor (normal instructions sequence) is interrupted
- When interrupt completes, it continues to execute the program instruction of where it left

Interrupt Type	Description
Program	• Generated by some condition that result from an instruction execution • Usually happens when executing illegal instructions (such as instruction outside from a user's memory)
Timer	• Generated by a timer in the processor • Used to perform tasks on a regular basis
I/O	• Generated by I/O controller • To signal the competition of an operation
Hardware	• Generated by failure • Hardware failure like power shutdown or data corruption

Computer Memory Constraints

- Amount (capacity) – certain program consumes large portion of memory, so memory tends to be limited in some cases
- Speed (access time) – if the memory is almost full, the speed when opening up another program tends to be a little slower
- Expense (cost) – if we want more RAM, the higher the cost that we need to pay

Memory Hierarchy



- Three design key features

1. Faster access time (RAM) = more expensive
2. Higher capacity (storage / ROM) = less expensive = slower access time
3. Top → store RAM (faster time), bottom → for storage (slower time)

Characteristics	Description
Capacity	• The more the storage, the cheaper it is
Access time	• The access time increases when moving from top to bottom • The faster the access time, the more expensive it is
Cost per bit	• The cost decreases when moving from top to bottom • More memory on top requires more cost

Principle of Locality

- Memory references tend to cluster
- Data is organized so that percentage of accesses to each successively lower level is less than above
    - In other words, it is designed so that we can easily access the higher-level ones in short time
- Can be applied across more than two levels memory:
    - Registers (fastest, most expensive and it resides in the peak of memory hierarchy)
    - Main memory (slower, larger, less expensive, and it resides below registers)
    - Secondary memory (slowest, much larger, cheapest and it resides below main memory)

Cache Memory

- Contains a portion of main memory
- Exploits the principle of locality with fast memory
- Processor will first check the cache to access the data and if the data is not found, it will be loaded from main memory into cache

Lecture 2: Operating Systems Overview

- A program that controls the program execution
- Interface between applications and hardware – so that user can easily read what is going on in the hardware, in other words, a bridge between a software and hardware

Evolution of Operating Systems

- Reasons for OS version to evolve over time
  - Hardware upgrades
  - New hardware – new hardware needs better OS performance
  - New services – like a computer embedded with fingerprint unlock requires the OS to read
  - Fixes – to fix bugs in the previous OS version

<table>
<tr><th>Types of OS</th><th>Description</th></tr>
<tr><td>1. Series Processing</td><td>• No operating system
• Programmers interacted with the hardware directly</td></tr>
<tr><td>2. Simple Batch System</td><td>• Users don’t interact with the computer directly
• An operator will receive jobs provided by the users who will them batches (groups and sort) the jobs together and execute them
• Problem with this is that the processor time alternates between the execution of user program (which provided by the operator) and the execution of monitor (where the program sends a signal to the monitor to tell that job has completed), which waste processor time for alternating
• Modes of operations:
<table>
<tr><th>User mode</th><th>Kernel mode</th></tr>
<tr><td>• Executes user program
• Cannot execute certain instructions
• Cannot access protected memory</td><td>• Executes monitor
• Execute privileged instructions
• Access protected memory</td></tr>
</table>
</td></tr>
<tr><td>3. Multi-programmed Batch System</td><td>• Main purpose: to multitask multiple programs
• Uni-programming
  ○ CPU waits for the I/O
  ○ Processor will “do nothing” – only waits for I/O before proceeding to execute next instruction
  ○ Hogs computer resources and utilize lots of time as CPU spends time keep waiting
• Multi-programming
  ○ CPU does not wait for the I/O
  ○ Processor will switch to other jobs to execute while waiting for the I/O
  ○ Saves computer resources and cuts the execution time as CPU “multitask” on multiple programs</td></tr>
</table>

4. Time-Sharing Systems	• Main purpose: to multitask multiple users / jobs • Such OS shares the processor time among multiple users to minimize the waiting time • For instance: an organization has two computer users – user A and user B. When user A is using its computer, user B does not need to wait for user A to finish using the computer because user A's and B's processor time is shared equally

OS Development

Development Type	Description

1. Processes	• An executable program
	• Associated data needed by the program (variables, workspace, buffers, etc.)
	• Execution context (or process state) of the program such as:
	  ○ The internal data that is able to be supervised by OS
	  ○ The contents of the various process register
	  ○ Other information like priority of the processes

2. Memory management	• OS could manage well computer's memory through virtual memory and file system facilities • Virtual memory such as "invisible memory" • File system facilities such as deallocating the memory when a file is closed

3. Information protection and security	• OS started to develop based on security concerns such as
	  ○ Confidentiality
	  ○ Data integrity
	  ○ Authenticity
	  ○ Availability
	• Different levels of organization will manage its security level differently

4. Scheduling and resource management	• OS started to able to manage all computer resources efficiently by considering factors like
	  ○ Fairness – each process executes with fair time
	  ○ Responsiveness – executes process based on its priority level
	  ○ Efficiency – minimize the response time

Lecture 3: I/O Management and Hard Drives

Types of I/O Devices	Description
Human readable	• Devices that can be easily controlled by the user, for instance using printers that plugged into a computer • The interaction happens between the hardware printer (controlled by user - because user can easily unplug it) and the computer user • Other examples such as keyboards and mouse, where the main controller of the devices is the user because the user has the ability to plug and unplug those devices
Machine readable	• Devices that can be easily read by the computer, for instance using USB thumb drives which allows the computer to read and write the data into the USB stick • The interaction happens between the electronic device (because the data is read by the computer) and the computer • Other examples such as CDs, motherboard sensors. In other words, we could say that devices that fall under this category are usually "pure computer" equipment where they are usually used as subsets for computer parts
Communication devices	• Devices that can be easily used to communicate with other devices, for instance computer uses a wireless adapter to connect to a router. Both wireless adapter and router are communicative devices • The interaction happens between the communication devices • This allow data to be passed between computer devices remotely • Other examples such as wireless interface cards, modems, Bluetooth adapters

I/O Performing Techniques	Description
1. Programmed I/O	• Processor issues an I/O command, then the process busy waiting for the I/O operation to be completed • How it works: a) Processor issues I/O command to request for an I/O operation b) I/O device starts performing the operation c) When completed I/O device will set the status bit to “done" d) While performing the operation, the processor will periodically check the status bit. When the status shows “done” the processor continues its task. This is because the I/O device does not inform the processor on its completion but only sets the status bits
2. Interrupt-driven I/O	• How it works: a) Processor issues an I/O command b) The I/O device transfers the data to the I/O module. At the same time, the processor continues its task c) When completed, the I/O module interrupts the processor to notify the completion of the data transfer d) After notifying, the I/O module transfers the data to the processor e) Processor issues I/O operation -> I/O module gets data from I/O device -> I/O module informs processor -> I/O module transfers data to processor • There are two approaches: a) Non-blocking: i. the processor continues its task while the I/O device is transferring the data to I/O module b) Blocking: ii. the processor halts its next instruction of the current process by putting it in a blocked state, then the processor executes the instructions of another process (not the current process because the current process is in the blocked state). iii. Process = a series of tasks to be executed (like a python program is a process). iv. When the I/O module transfer is done, then the current process will be transferred back to ready state to be executed next
3. Direct Memory Access (DMA)	• issue I/O request > DMA transfer > done > DMA informs processor) • Controls the data exchange between main memory and I/O module, for instance, transferring the data from a USB stick (I/O device) to the main memory of the computer. The data transferred to the main memory which will then be transferred to the computer’s hard drive subsequently • This is usually faster because the transfer of data does not have any interference with the CPU task • How it works: a) When an I/O device is needed (like a software prompting user to plug in an external hard drive for external data transfer), the software will first interact with the processor b) The processor then requests the DMA to transfer a block of data to the processor when an I/O device is detected by the DMA

	c) While DMA is transferring the data from external hard drive, the processor continues with its own task on hand d) When the DMA has completed the data transfer, it will then interrupt the processor to pass the data transferred back to the processor e) DMA acts like a middleman that does the job of interacting with external clients f) Processor -> request DMA to receive data from I/O -> DMA transfer data from I/O to main memory → DMA informs the processor and interrupt the processor → DMA transfer the data received back to processor • Processor is only involved during the process of requesting data from I/O and at the end of the transfer process

Disk Reading and Writing

- In order to read / write, the head must be positioned:
  - At the desired track
  - At the beginning of the desired sector on that track
- 4 delay elements:
  - Track selection – moving the head in a movable head system
  - Seek time – time it takes to position the head at the track
  - Rotational delay – time it takes to reach the head
  - Access time – seek time + rotational delay

Disk Cache

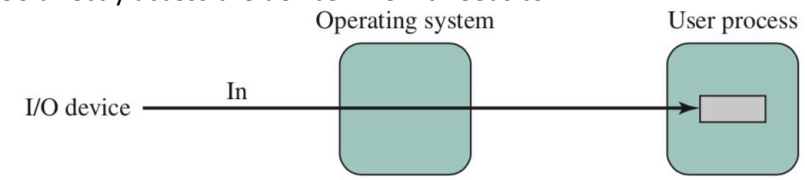
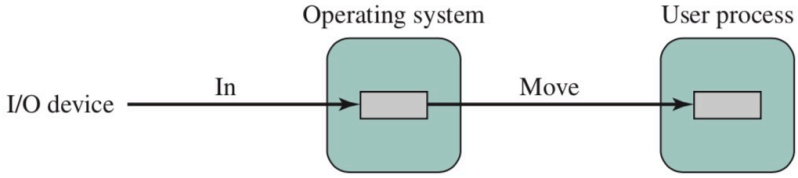
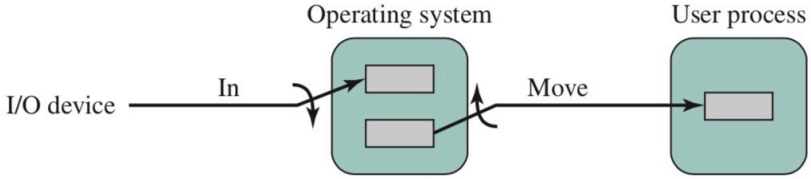
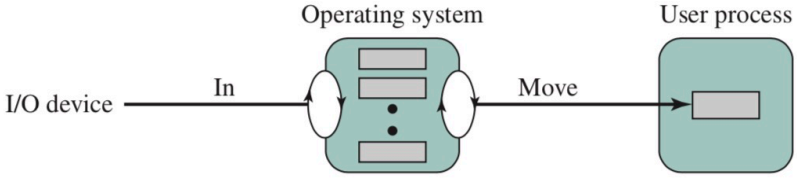
- Main purpose is to reduce the access time by exploiting the principle of locality
- Used to apply to a memory that is smaller and faster than main memory
- When an I/O request is made, it first checks if the cache contains such request and read from there

I/O Buffering

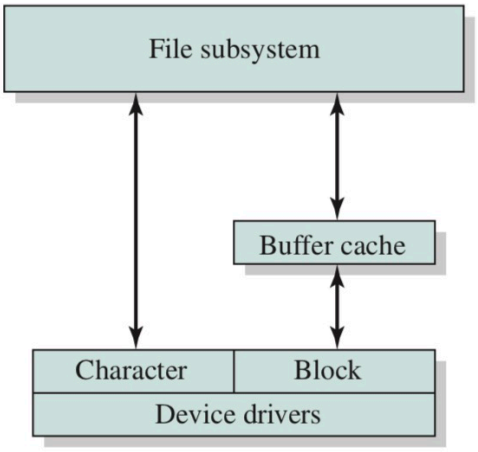
- Main purpose is to smooth out the I/O request (multiprogramming)
- Performs two things:
  - Input transfer before request is made
  - Output transfer after request is made

Device Types	Description	Examples
Block-oriented Device	• Stored information in blocks of fixed size • transfer information per block	• Disk drive • Tape drive • USB keys
Stream-oriented Device	• Store information in stream of bytes • No block structures • Transfer information as a stream of bytes	• Terminal • Printers • Communication ports

I/O Buffering Types

Buffering Type	Description
No buffer	OS directly access the device when it needs to 
Single buffer	OS uses a single buffer by anticipating an I/O request If the I/O device is block-oriented: • When the transfer into buffer is done, the process moves the block into user space and immediately request for another block • This means that single buffer is able to predict that "it will always need another block to read in" • Such approach is faster because the block can be processed in user process while another block of data is being read in (aka simultaneous) If the I/O device is stream-oriented: • Line at a time (data is in the form of lines) where user input is one line • Byte at a time (data is in the form of keystrokes) to produce interaction between OS and user process 
Double buffer	OS uses two system buffers: • Transfer data to the first buffer • If first buffer full, it transfers data to second buffer • While transferring data to second buffer, the OS empties the first buffer by moving it to the user process • Buffer will continuously swap until all data is transferred 
Circular buffer	OS uses two or more buffers: • Improved version of double buffer in speed because circular buffer can input more data at single time • Each individual buffer is one unit in a circular buffer • Used when I/O operation must keep up with process (aka performance is emphasized / smooth flow of data) 

I/O Management in Unix



Buffered I/O	Buffer Cache	• Mainly used by block-oriented devices
		• A disk cache that performs I/O operations
		• Data transfer occurs via DMA
		• Because buffer cache lives in the main memory and help to transfer the data from I/O device to buffer cache, it does not use up any processor cycles, but it consumes bus cycle
		• Requirements:
		  ○ Free list – list of slots in the cache that are open for allocation (aka data transfer)
		  ○ Device list – list of all buffers associated with each disk
		  ○ Driver I/O queue – list of buffers that are waiting for I/O on a device

	Character Queue	• Mainly used by character-oriented devices • Data is written by I/O device and read by the process • Once the character is read, it is destroyed
Unbuffered I/O		• DMA between device and process space • Considered as the fastest way to perform I/O • The process is locked in main memory and cannot be swapped out • Thus, it causes the I/O device to tie up with the process for the duration of the transfer, making unavailable for other processes, resulting in a faster approach

Disk Scheduling

Example: Disk requests: 98, 183, 41, 122, 14, 124, 65, 67. Head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass

Algorithms	Description	Advantages	Example
First in First Out (FIFO)	• Processes the requests in sequential order (aka first come first served)	• Fair to all processes	98, 183, 122, 124, 65, 67
Priority (PRI)**	• Control scheduling outside of control of disk management software • Goal is not optimizing the disk utilization but to meet other OS objectives • May have starvation because longer jobs may have to wait for long time • Shorter jobs are provided with higher priority	• Provide good interactive response time	
Shortest Service Time First (SSTF)	• Selects the request that has least number of head movements from current position • Always choose with minimum seek time (shortest distance first)	• High utilization with small queues	65, 67, 41, 14, 98, 122, 124, 183 **get minimal difference**
SCAN (Elevator)	• The head moves in one direction • When it reaches the end of disk, it reverses the direction and services all the requests along that direction	• Better service distribution	65, 67, 98, 122, 124, 183, 41, 14
C-SCAN (Circular SCAN)	• Improved version of SCAN • SCAN moves in one direction, then reverse its direction from the end • C-SCAN moves in one direction, but when it reverses, it doesn't service the request along • C-SCAN restricts the request to only one direction	• Lower service variability	65, 67, 98, 122, 124, 183, 14, 41

Lecture 4: File Management

File

- A sequential list of characters of number of bytes
- Two types of files:

File Type	Description
Text files	• Made up of characters such that each character is also known as a byte • The terminal displays the actual characters in the
Binary files	• Store in a sequence of 4 bytes to represent 32-bit integers • Also known as executable files • Used to be executed directly by processor • It shows nonsense character when viewed the file as text file

File Allocation Table (FAT) [Windows OS]

- Directory in FAT stores the address of the starting block of each file
- Files are allocated to the first empty block then chain together with other blocks to form a cluster for a larger file (similar to linked list)
- For instance, hello.txt file is stored in a folder and stored in the memory block of 0002 while world.txt is stored in the block of 0005. Then the folder will keep track of these memory blocks (like an array of linked lists)
- When user reads the hello.txt file, it looks at the 0002 address with a chain to the subsequent linked blocks until end of file is read
- The main purpose of this chain is to remove external fragmentation (or to reduce wasted space)

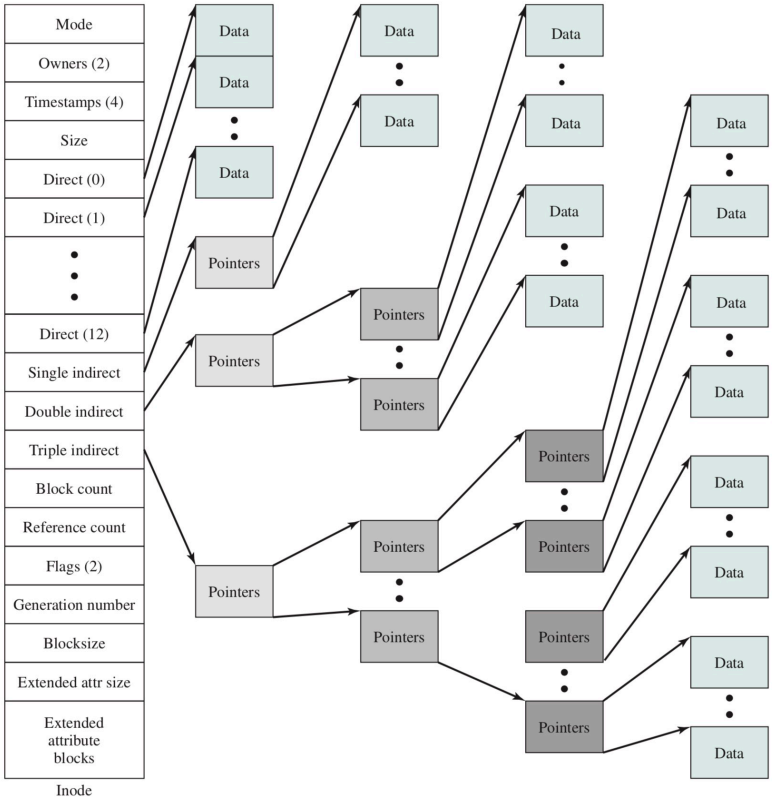
Fragmentation Types

Fragmentation Types	Description
External Fragmentation	• Talks about "outside view" of the blocks, like how file is allocated into each block and how those blocks are wasted when a small file is removed from the memory blocks • Free blocks exist between files, but those free blocks are not large enough to fit some files • These spaces may only be usable by smaller files • Such fragmentation creates multiple memory blocks are empty (or wasted) because new files are tough to fit in the slots
Internal Fragmentation	• Talks about "inside view" of the blocks, like when a small file is allocated into a block that is larger than the size of the block, then the remaining space becomes wasted space • This wasted space cannot be allocated or shared with other files • Like a file needs 1GB but block size is 2GB, the remaining 1GB free space is result of internal fragmentation

File System Aging

- Chaining method in FAT system allows it to reduce wasted space by linking memory blocks together but the file is spread over multiple tracks
- This spread over of multiple blocks increases the seek time (time to access the data) because the head of the hard drive needs to move to different locations every time to access the entire file
- For instance, a file is located at 0002, 0003, 0005 then 0099, the hard drive consumes some time to move the head to 0099 location, which is time consuming
- Over time, the disk will perform periodic defragmentation to allow the memory blocks be arranged continuously rather in chaining manner

INodes [Unix OS]



- Instead of fragmenting the files into multiple blocks like what FAT does, INodes makes each directory to have a list of files and each file is mapped to an INode (a data structure)
- Files are located by looking up the appropriate inode in the inode table
- If the file requires less than 12 blocks on a typical disk, it will be listed directly in the inode itself through direct(1), direct(2)…. direct(12), where each “direct” stores a memory block of data (similarly to a block in FAT)
- If the file requires more than 12 block of memory, last direct(12) is mapped to a pointer such that the pointer contains the remaining blocks
- Depending on the size of the file, if the size is extremely large, multiple levels of pointer blocks are used, meaning there can be up to triple indirect pointer

Booting OS

- OS stays in hard drive; we need a way to link the hard drive to the computer system to bootup the OS
- Method of linking is called bootstrapping – small primitive system (aka low-level software) to load OS
- Two types of bootstrapping:

Bootstrap Approach	Description
BIOS	• A small tiny chip is located at the very end of the main memory • When the CPU first boots up, it looks up the data in this tiny chip • This tiny chip contains the instructions code to recognize the connected hard drive. In other words, this tiny chip helps to connect the CPU to the hard drive that contains the OS so that the CPU is able to boot up with the OS • It only knows how to load the raw bytes (aka information) from the hard drive to the memory, meaning the tiny chip only knows how to boot up the OS
Master Boot Record (MBR)	1. The first 512 bytes on a bootable hard drive contains instructions code to load the 2. BIOS first loads the instructions into the main memory 3. The main memory will then contain the instructions of loading the OS 4. The CPU is then pointed to this location in the main memory to boot up the OS

Lecture 5: Process

- A program that is being executed (or running instance of a program)
- Requirements for a program to be executed:
    - Program code (a series of logical instructions)
    - Data associated with program code
- For instance, playing a game requires the data of the game (accessed from game database) and program code (logic of the game)

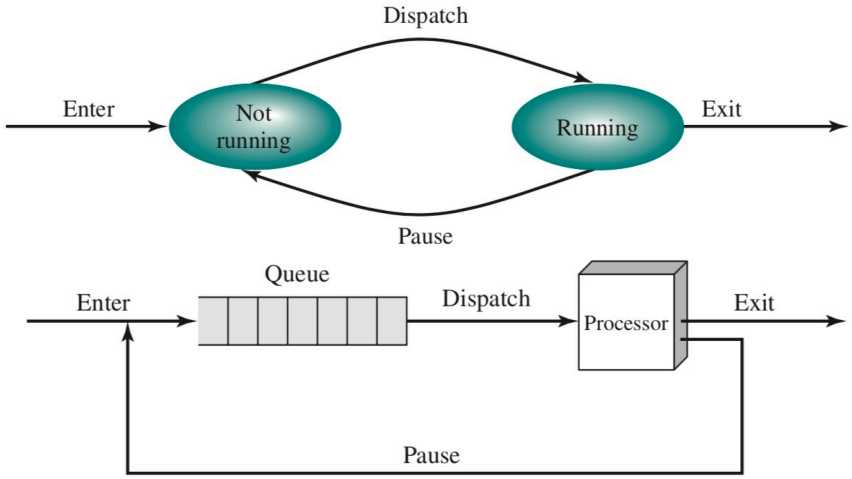
Process Control Block (PCB)

- Process = Program Code + Data + PCB
- PCB is a data structure containing the elements of the process
- When the program code is executed, PCB provides a control flow over the program code on its execution (aka interrupting the code execution) temporarily
- When the process is ready to continue, it will return back to the instruction of where it left
- PCB is useful for multiprocessing (handling many processes at the same time through interrupts)

Program Execution

- A process is created for every program execution
- Running a python file will create a process for the python file
- While running, the processor will change the PC register to point to next instruction of the file
- Process is typically created by another process
- For instance, PyCharm opening a python file
    - The OS (a process) creates a process for PyCharm application
    - The PyCharm (a process) creates another process for opening a python file
    - The Python file (a process) contains a series of instructions to be executed
    - Here, PyCharm is the parent process while python file execution is the child process

Two-State Process Model

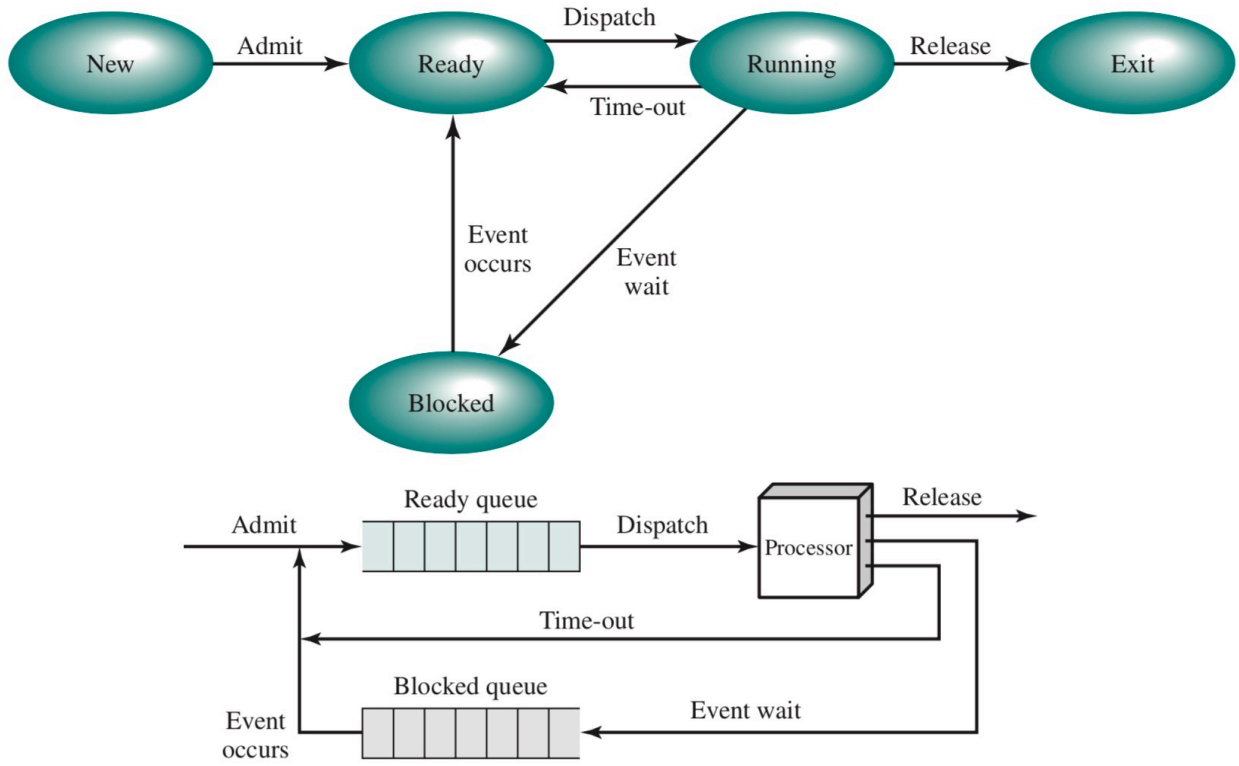


States	Description
Not Running	• Process will first enter the not running state and behaves like a queue • Each process in the queue is a pointer to PCB
Running	• Process will be dispatched by the processor to enter running state • Each process that enters this state is given a fixed time to execute • If the process has not been executed completely, it will pause and return to the not running state so that it will continue to execute in next turn • This uses round robin strategy with fixed execution time

Disadvantage of Two State

- If a process is making I/O request, it should not enter not running state yet
- This is because the process is not ready to be executed yet
- Hence, we need additional approach that could deal with interrupts or I/O requests, which is the Five State Process Model

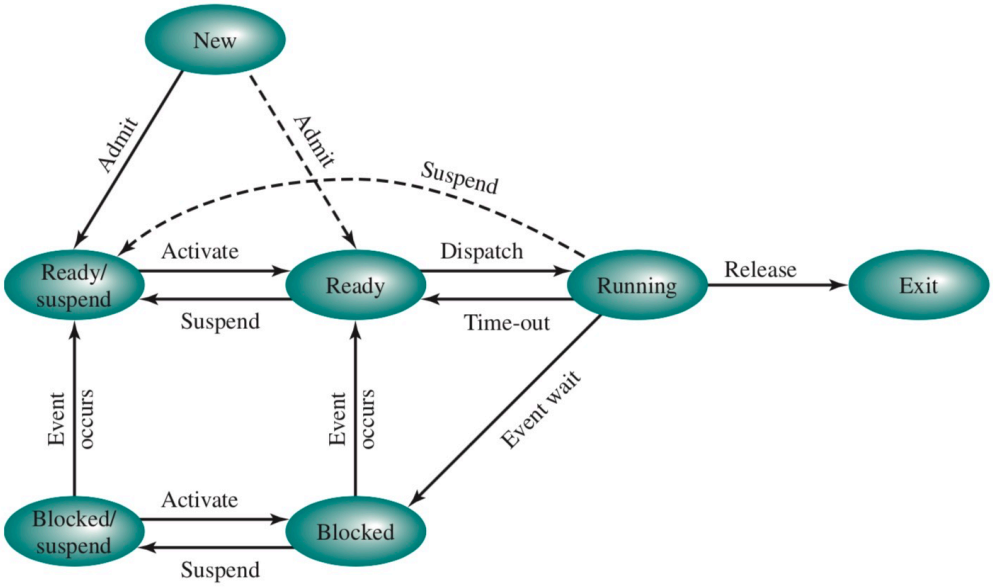
Five-State Process Model



States	Description
New	• A new process is created here • The PCB of that process will then be admitted into the ready state to prepare to be executed
Ready	• Process in this state is waiting to be executed • Processor will dispatch these processes in order to execute them • The dispatch of processes depends on the scheduling algorithm used
Running	• Process in this state is executed • If the process has not been completed, it will signal timeout and return back to ready state to continue execute in its next turn • If the process request for I/O, it will be transferred to blocked state
Blocked	• Process in this state is waiting for some I/O events to occur • When the event occurs, it returns back to ready state to be executed
Exit	• When a process completes, it will be released by the OS into exit state • In some cases, when parent process terminates, all its child process will be terminated

Process Suspension

- Main purpose is to move part or all process from main memory to disk (to save main memory space)
- Can be said as a "temporary state" for the process to stay
- When ready state is empty, then OS moves one of the processes from the blocked state to suspend state
- OS then brings in another process from suspend state to honor a new process request (activating the process from suspend to ready state)
- In suspend state, process is not immediately available for execution, which needs to be transferred to ready in order to be executed
- The process may be removed from the suspend state if:
    - The parent process request for removal
    - The OS is preventing the process from further executing



State	Description
Blocked/Suspend	• OS moves one of the processes from blocked state to blocked/suspend state • Main purpose is to make room for new processes to reside in main memory • Processes in blocked/suspend will stay in secondary disk (hard drive)
Ready/Suspend	• When the I/O events happen for process in blocked/suspend state, it will be transferred into ready/suspend • Processes in ready/suspend is readily available for execution once it is loaded into main memory • When it is ready to be executed, it will be moved to ready state, based on the priority assigned to each process • After dispatched to running state, if the OS is preempting the process for higher priority, then the lower priority ones will be moved to ready/suspend state to free up some main memory spaces

OS Management

- Uses OS control tables to manage process
- Each entry of the record will contain at least one pointer to the process image (meaning in order for process table to be created, it first needs to have at least one process to be executed)
- The process images must be:
    - Referenced to other system resources directly or indirectly (like memory, I/O operations)
    - Accessible by OS (subjected to memory management)

OS's view of process

- OS manages the process by knowing only two things:
    - The location of the process
    - The attributes of the process (that are needed for management)

Process Location	Process Attributes

• It must include a program to be executed
• It must have sufficient memory to hold the data of the process
• It uses stack to keep track of which process to execute first
• Partial process image will reside in main memory (OS must know which portion) and large portion will reside in secondary memory
• When executing a process, the entire process image will be loaded into main memory
	• Uses PCB to keep track of the attributes
	• Program code + data + stack + PCB = process image
	• Process image location depends on memory management
	• PCB information is broken into 3 parts:
	  ○ Process identification
	  ○ Process state information
	  ○ Process control information

Process Identification

- Each process is assigned with a unique identifier which contains:
    - Index of the primary process table
    - Mapping of the OS to locate the appropriate process table
- Uses multiple tables to manage tons of processes (through concept of cross referencing)
- When processes communicate with another process, the process identifier informs the OS of the destination of a particular communication
- When processes are allowed to create other processes, identifiers indicate the parent and descendants of each process

Processor State Information

- Contains the contents of processor registers:
    - User-visible registers
    - Control and status register
    - Stack pointers
- Program Status Word (PSW):
    - A set of registers that contain condition codes and other status information
    - When a process is interrupted, the register information is saved so that it can resume operations

Process Control Information

- Additional information that is needed by OS to control and coordinate active processes such as:
    - State of the process to be scheduled (in ready/running /blocked state)
    - Priority of the processes

Process Switching

Mechanism	Cause	Use
Interrupts [I/O, time interrupts]	External events request causes the current instruction execution to be halted	To react to an external event
Trap	Associated with current instruction execution	OS determines if the error/exception and handle them accordingly
Supervisor call	An explicit request by the process	Process calls to an OS function like opening a file, hence switching the process to opening file state

Process States in Unix

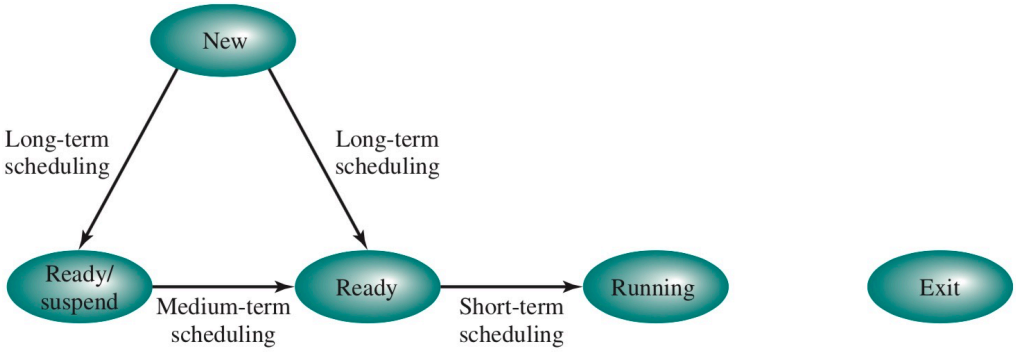
Process Type	System Processes	User Processes	
Running Mode	Run in kernel mode	Run in user mode	Run in kernel mode
Programs Execution	Execute OS code to manage resources (like memory allocation, process swapping)	Execute programs and utilities	Execute instructions that belong to kernel – like C program issuing system call

Process Creation in Unix

- Process is created through system call of fork()
    - It allocates a slot in process table for the new process
    - It assigns a unique process ID for child process
    - It makes a copy of the process image of the parent, with the exception of shared memory
    - It increments counters for any files owned by parent, to reflect that additional process also own those files (aka to share those files between parent and child)
    - It assigns the child process to the ready to run state
    - It returns the ID number of the child to the parent process and a value of 0 to the child process
- After executing the process, OS can do the following:
    - Stay in parent process – parent process continues to execute
    - Transfer control to child process – child process begins to execute
    - Transfer control to another process – both parent and child are turned into ready to run state

Lecture 6: Uniprocessor Scheduling

- Main objective is to assign processes in a way that maximizes its efficiency in aspects such as response time and processor efficiency
- Scheduling is done in three parts – long term, medium term and short term



Long-term Scheduling

- It determines if a program that will be added into pool of execution
- This means that scheduling in this way will control the level of multiprogramming – how efficient do the programs run when processes are executed concurrently
- If more processes are admitted for processing (aka execution), then there will be lots of process to be executed, and thus causing each process execution time to be reduced
- For instance, 10 processes and total time is 50 minutes, so each process gets 5 minutes. But if there are 100 processes, then each process only gets 0.5 minutes
- To avoid each process to have shorter execution time, long term scheduling usually is done in a strict way so that there are lesser programs admitted and thus causing the current set of processes have a higher efficiency (aka longer execution time)
- For instance, limiting each execution to have at most 10 processes, so that each process will have more time to execute, instead of allowing an infinite number of processes to be executed
- The scheduling on which job / program to admit can based on first-come first served algorithm

Medium-term Scheduling

- It determines if a program to be admitted into the execution pool partially or fully in main memory
- This swapping of programs out from main memory to disk or disk to main memory depends on the need of multiprogramming
- When a process is swapped in, it will consider the memory requirements of those processes
- For instance, if a process holds 1GB memory is swapped out into secondary disk, when trying to achieve multiprogramming with this process, it will first check if the ready state has sufficient memory to swap in this process
- If the ready state does not have sufficient 1GB space, then the process can't be swapped in
- Thus, medium term scheduling has memory management issues

Short-term Scheduling

- It determines if a program is to be dispatched into running state
- This scheduling is done most frequently because processes need to be dispatched every time
- It makes the final decision on which process to execute next

Short-term Scheduling Policies

Policies	Description
First Come First Served [Non-preemptive]	• Performs better for long processes due to non-preemptive (where it hogs the CPU resources without switching – wasting time) • Favors processor-bound processes over I/O bound processes • Processor-bound processes → process that spent most time in CPU (like performing calculations, executing instruction) • I/O bound processes → process that spent most time in performing I/O operations • FCFS is better at CPU-bound because CPU-bound does not need to be interrupted and does not do lots of process switching (which wastes time)
Round Robin [Preemptive]	• Each process is given a fixed slice of time before preempted • Useful in general-purpose time-sharing system • It has huge overhead because providing each process fixed share of CPU needs to keep switching the resources between processes (known as context switching)
Shortest Job First / Shortest Process Next [Non-preemptive]	• Selects the process with shortest execution time • Can cause starvation for longer process (because if a longer process came early, those processes with shorter time will be executed first, so the longer process may not have opportunity to execute) • Such policy requires it to estimate the process's execution time
Shortest Remaining Time [Preemptive]	• Preemptive version of SJF above (because it gives a fixed execution time, then CPU takes back the resources) • Selects the process with the shortest remaining processing time • For instance, while executing current process, the remaining time is 5, then a new job with execution time of 3 comes in, the old job will be halted, and the new job will be executed immediately • It provides a good turnaround time performance because for all new shorter jobs that comes in, they will be given preference first without waiting
Feedback Scheduling [Preemptive]	• Scheduling is done on a time quantum basis using priority • For instance, there are three processing queues, when the process first enters the first queue, the CPU estimates its processing time to determine its priority • If the priority is lower, the process will be shifted to subsequent queues • Lower priority process may suffer from starvation

Selector Function

- Usually used as part of short-term scheduling to select the process to execute next
- Two decision modes in selector function:

Non-preemptive	Preemptive

• "Non-preemptive" means allocated computer resources cannot be taken back
• When a process is in running state, it will continue to execute until:
  ○ the process terminates
  ○ the process request for I/O which enters into blocked state
• When it reaches one of the states above, the resources are returned back to CPU
	• "Preemptive" means allocated computer resources is only for limited time
	• Currently running process can be interrupted and move to ready state
	• When this happens, the resources are taken by back the CPU
	• This usually happens when new process arrives (it needs to utilize the resources, so CPU takes back from the old process)

Lecture 7: Threads

- Unit of dispatching: thread / lightweight process
- Unit of resource ownership: process / task

Process and Threads

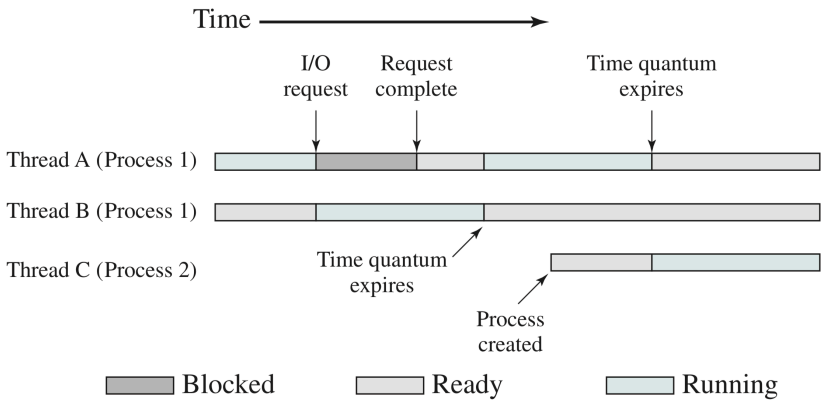
Process	Threads
• Heavyweight process • Consumes more resources • Takes more time for context switching • Different process has different address / memory space	• Lightweight process • Consumes less resources • Takes lesser time for context switching • All threads within a process share same address space • Multi-threads (that run concurrently) improves response time • If these threads interact different parts of the program, concurrency control (mutual exclusion) need to be applied • Takes less time to be created • Enhance efficiency when communicating between programs (using threads)

Threads Attributes

- Execution state
- Saved thread context when not running
- Execution stack (contain information required by the thread to execute)
- Static storage for local variables
- Access to memory and resources to its process (threads share same address space)

Thread States

- Threads have the same states as process: READY, RUNNING, BLOCKED
- If a process is swapped out into secondary memory, all threads will be swapped out too because they share the same address space
- A process change in state is managed by the operating system, but a thread change in state is managed by its own operations



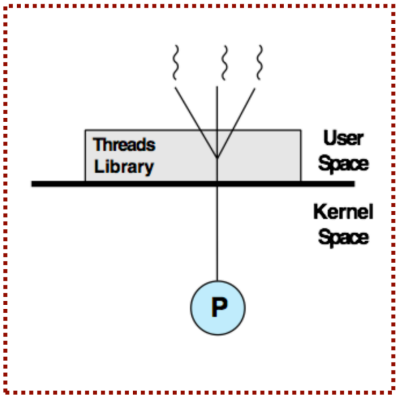
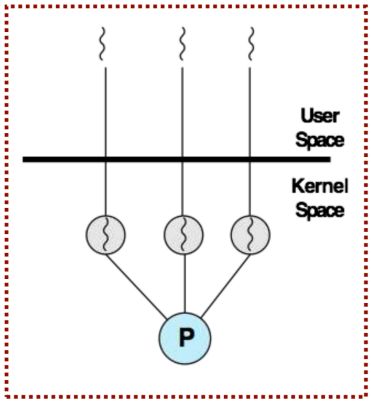
- Thread operations:

Operation	Functionality
Spawn	• Spawns a new thread when a new process is created • Spawns a new thread by another thread within a process
Block	• Threads wait for an event while blocking registers, PC, and stack pointers • Executes another thread (of same / another process) while waiting
Unblock	• Moves the blocked thread into Ready state when the event happens
Finish	• Deallocates stack pointers for threads to remove memory

Thread Synchronization

- Threads share the same address space with other resources
- Modification done by one thread could affect the performance of another thread
- Need to synchronize the activities of the threads to avoid interference of its own performance

User level threads & Kernel level threads

Thread	User-level	Kernel-level
Description	• All threads are done by application (which the thread is executed from) • The OS is not aware of existence of threads • Any program can utilize multi-threading concept by just using the threads library (without the OS knowing / supporting)	• Threads are done by kernel • The OS knows and aware of these threads (as it is the one that creates)
Example		
Pros	• As the OS doesn't know about these application/user-created threads, so switching between threads doesn't require kernel privileges, making it faster • Can run on any OS (by just utilizing the threads library, making it more convenient)	• Because these threads are control by the kernel, so the kernel can simultaneously schedule multiple threads (of the same process) on multiple processors – perform multiprogramming easily • If one thread is blocked, the kernel can schedule another thread easily
Cons	• A multi-thread application cannot take full advantage of multiprocessing because threads will be blocked by certain system calls	• Requires mode switching when transferring one thread to another thread (of the same process) – causes some overhead

Lecture 8: Concurrency

Factors When Managing Process and Threads

Concern	Description
Multiprogramming	• executes multiple processes on a uniprocessor system
Multiprocessing	• executes multiple processes on a multiprocessor system
Distributed computing	• executes multiple processes on a distributed computing system

*uniprocessor: computers with only one CPU core as processor
*multiprocessor: computers with multiple CPU cores as processors (dual-core, quad-core)

Concurrency

Principles	Speed of execution of processes is unpredictable because it relies on: • the activities of other processes (if more processes are running at the same time, speed may be slower) • the way OS handles interrupts • scheduling policies of OS
Difficulties	• Problems occur because certain resources are shared among processes/threads • Difficult to manage allocation of resources optimally • Difficult to locate programming errors – results are not deterministic and reproducible (results vary every time)

Control Problems When Solving Concurrency Issues

Problem	Description	Requirements
Mutual Exclusion	• Main purpose: to maintain the integrity of the shared data (can only be modified one at a time, prevent info leak) • Multiple processes running the same code • This code has a critical section where it allows the process to modify the shared data • To enforce mutual exclusion, when one process has entered the critical section, no other processes are allowed to enter that critical section	• When a process exits critical section, it must not interfere other processes (running the same code) • A process can only be in the critical section for finite time (cannot forever) • Deadlock and starvation cannot happen
Deadlock	• Situation where multiple programs sharing the same resource prevents each other from accessing the resource	-
Starvation	• Condition where high priority programs continuously requests and receives resources, preventing low priority programs to obtain resources to execute • Considered as indefinite blocking of a process	-

Concurrency Problems

- Happens when multiple processes / threads access and modify shared data (race condition)

```
Process P1                    Process P2
•                             •
chin = getchar();             •
•                             chin = getchar();
chout = chin;                 chout = chin;
putchar(chout);               •
•                             putchar(chout);
•                             •
```

	Process P1	Process P2
Multiprogramming Approach	In Uniprocessor → P1 and P2 execute on same processor core In Multiprocessor → P1 and P2 execute on different processor cores	
Shared variables	chin; chout;	
Modifications	[1] chin = 10 [2] chin = 20 [3] chout = 20 [4] output = 20 [5] output = 20	
Steps taken	[1] Gets a new character, chin = 10 [3] Sets chout same as chin [4] Prints chout	[2] Gets a new character, chin = 20 [3] Set chout same as chin [5] Prints chout
Output	20	20
Expected Output	10 – because it reads 10	20 – because it reads 20
Conclusion	• Data read is not consistent / no integrity • Results in race condition where each process race to modify data	
Solution	• Synchronize the processes to avoid reading and writing at the same time and to ensure data consistency by letting processes to cooperate • Set controls over who can access (like when P1 is executing, prevent P2 from reading another input value)	

OS responsibilities

Concern	Description
1. Keep track of various processes	• Using PCB (process control block) structures to keep track of list of processes
2. Allocate and de-allocate resources for active process	• If the process is active (running), OS must provide resources • When the process is no longer active, OS must take back its resources (preempting it whenever possible) • If there are multiple process executing, OS must allocate resources fairly **Example of resources:** 1. Processor time 2. Memory 3. Files 4. I/O device
3. Protect the data and physical resources of each process	• Each process's data must be independent so that data cannot be modified / interfered by other processes • This means that the OS must set controls over what the process can access, such as memory and files
4. Ensures the process's output are the same regardless of the processing time	• Each process's task should not be influenced by processing time • This means that if process A and process B are doing the same thing, but A takes a longer time than B, the output of both processes must be the same • Reason is because the functioning (the objective) of both processes are the same • The functioning of a process and the output it produces must be independent of the speed at which its execution is carried out relative to the speed of other concurrent process.

Control Mechanisms of Critical Section

- Processes will compete for the same system resources (I/O devices, memory, processor time)
- These competitions create control problems because sometimes they need to run concurrently
- When they run parallelly (at the same time), it creates concurrency – race condition
- 3 solutions to serialize the critical section (making the processes to run in sequential manner):

<table>
<tr><th>Approach</th><th>Description</th></tr>
<tr><td>Semaphores</td><td>

Using integer variables as signals among processes
3 operations in semaphores (counting semaphore type):

<table>
<tr><td>1. Initialization</td><td>Initialize a locker for semaphore to a non-negative integer value (usually the initial value is set to 0):
<pre>/* initialise locker for sem_t */
sem_init(&mutex, 0, 1);</pre></td></tr>
<tr><td>2. Decrement</td><td>Using semWait operation to decrease the value. If the value is -ve, then a process executing critical section</td></tr>
<tr><td>3. Increment</td><td>Using semSignal operation to increase the value. If the value becomes +ve, then a process has exited.</td></tr>
</table>
<pre>/* The first thread - it increments the global variable theValue */
void *pthread1(void *param)
{
    int x;
    printf("\nthread 1 has started\n");

    /*** The critical section of thread 1 */
    sem_wait(&mutex);
    sleep(rand() & 1);   /* encourage race condition */
    x = theValue;
    sleep(rand() & 1);   /* encourage race condition */
    x++;   /* increment the value of theValue by 1 */
    sleep(rand() & 1);   /* encourage race condition */
    theValue = x;
    /*** The end of the critical section of thread 1 */
    sem_post(&mutex);

    printf("\nthread 1 now terminating\n");
}</pre>
</td></tr>
<tr><td>Mutex</td><td>

A variable flag set to:

0 when a critical section is locked, and
1 when a critical section is unlocked

While a process is in the critical section, no process are allowed to enter (because only one process is allowed to hold the lock one at a time)
When a process is holding the lock, then this process must be the one that unlocks the lock when it exits the critical section

<u>Example (similar to binary semaphore type):</u>
<pre>/* The first thread - it increments the global variable theValue */
void *pthread1(void *param)
{
    int x;
    printf("\nthread 1 has started\n");

    /*** The critical section of thread 1 */
    pthread_mutex_lock(&locker);
    sleep(rand() & 1);   /* encourage race condition */
    x = theValue;
    sleep(rand() & 1);   /* encourage race condition */
    x++;   /* increment the value of theValue by 1 */
    sleep(rand() & 1);   /* encourage race condition */
    theValue = x;
    /*** The end of the critical section of thread 1 */
    pthread_mutex_unlock(&locker);

    printf("\nthread 1 now terminating\n");
}</pre>
</td></tr>
<tr><td>Conditional variables</td><td>

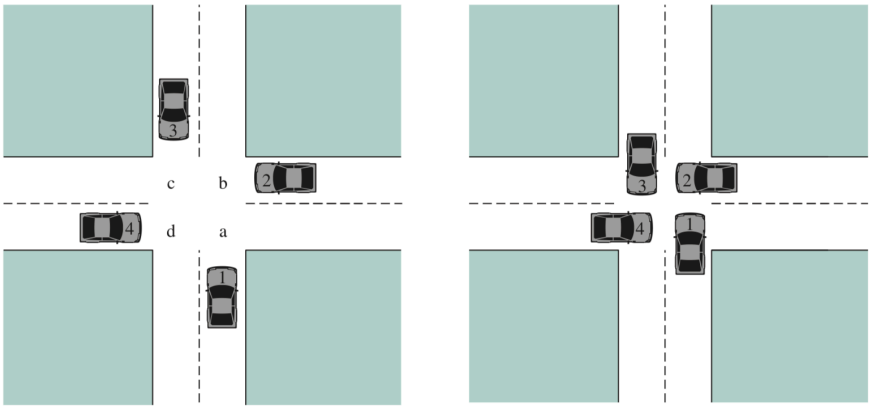
Make a process wait until a condition is satisfied.
When another process completes the critical task, the waiting process is unblocked

</td></tr>
</table>

Deadlock

<table>
<tr><td>Definition</td><td colspan="2">

Permanent blocking of a set of resources that compete for system resources
This means that a set of processes are “locked” in a situation where each process is waiting for one another (a blocked process where it is also waiting for another)



</td></tr>
<tr><td rowspan="2">Resources Competition</td><td>Reusable</td><td>

Resources that will not be depleted after one process finish using
Example: processors, memory, files, database, I/O channels
Like memory, it will not be consumed completely when an application stops running

Deadlock execution sequence: $p_0\ p_1\ q_0\ q_1\ p_2\ q_2$
<table>
<tr><th>Step</th><th>Process P Action</th><th>Step</th><th>Process Q Action</th></tr>
<tr><td>p_0</td><td>Request (D)</td><td>q_0</td><td>Request (T)</td></tr>
<tr><td>p_1</td><td>Lock (D)</td><td>q_1</td><td>Lock (T)</td></tr>
<tr><td>p_2</td><td>Request (T)</td><td>q_2</td><td>Request (D)</td></tr>
<tr><td>p_3</td><td>Lock (T)</td><td>q_3</td><td>Lock (D)</td></tr>
<tr><td>p_4</td><td>Perform function</td><td>q_4</td><td>Perform function</td></tr>
<tr><td>p_5</td><td>Unlock (D)</td><td>q_5</td><td>Unlock (T)</td></tr>
<tr><td>p_6</td><td>Unlock (T)</td><td>q_6</td><td>Unlock (D)</td></tr>
</table>

Deadlock occurs because the processes are “stuck” in a situation where two of them need each other

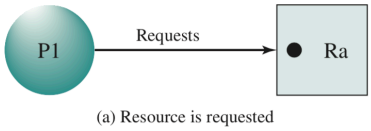
</td></tr>
<tr><td>Consumable (non-reusable)</td><td>

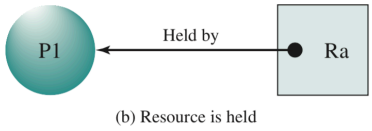
Resources that can be created (produced) and destroyed (consumed)
Such resources are only one-time usage (non-reusable)
Example: interrupts, signals, messages, I/O buffers info
Like pen drive (I/O buffers), the message communicated between the pen drive and computer will be destroyed after the pen drive is plugged out

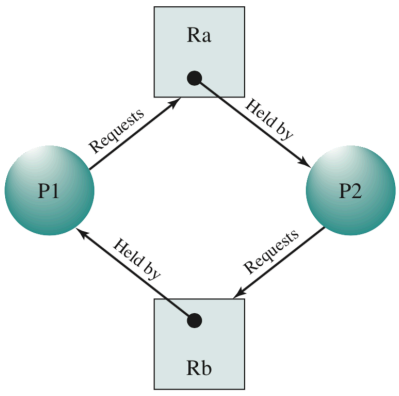
<table>
<tr><th>P1</th><th>P2</th></tr>
<tr><td>…
Receive (P2);
…
Send (P2, M1);</td><td>…
Receive (P1);
…
Send (P1, M2);</td></tr>
</table>

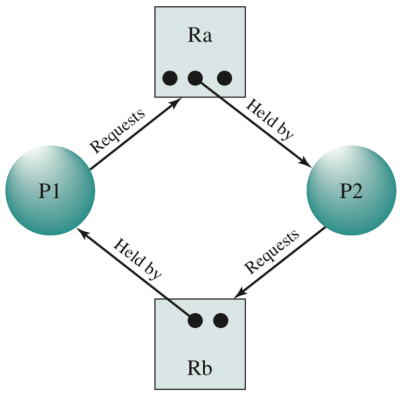
Deadlock occurs when both P1 and P2 are trying to receive the messages send from each other, but unable to receive because the sending part happens after the receive. (i.e., receiving is blocked)

</td></tr>
</table>

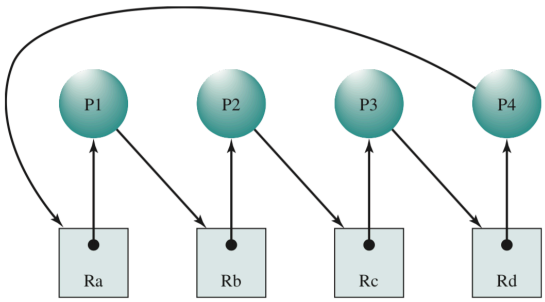
<table>
<tr>
<td>Resource Allocation Graph</td>
<td>


(a) Resource is requested


(b) Resource is held


(c) Circular wait


(d) No deadlock

Circular Chain:


</td>
</tr>
<tr>
<td>Conditions Required</td>
<td>
1. Mutual exclusion: only one process using a resource at a time
→ causes the process to “hold” the resources, not releasing it

2. Hold and wait: a process is holding its allocated resources while waiting for assignment of other resources
→ causes deadlock because if the assignment of other resources (is also waiting for others, it may wait infinitely, results in circular wait)

3. No preemption: no process can be forcibly removed from a process holding it
→ causes a process to hold it “forever”, not releasing it

4. Circular wait: each process is in a closed chain, such that each process holds at least one resource needed by the next process in the chain
→ causes one process to wait for the next process infinitely

<table>
<tr><th>To make deadlock possible</th><th>To make deadlock exist</th></tr>
<tr><td>1. Mutual exclusion
2. Hold and wait
3. No preemption</td><td>1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait</td></tr>
</table>
</td>
</tr>
</table>

Solution	
Prevention	• Objective: to adopt a policy that prevents deadlock conditions • Two approaches: 1. Indirect – prevent the 3 conditions from occurring a) Mutual exclusion • Mutual exclusion is where when one process is in a critical section, no other processes are allowed to enter and should be waiting for it • This waiting may possibly cause deadlock where they are locked in a situation waiting for one another • To prevent this, mutual exclusion should not be disallowed b) Hold and wait • A process holding and waiting too long causes deadlock • To prevent possibility of deadlock, this requires the process to request all of its required resources at one time, and • blocking the process until all requests can be granted simultaneously c) No preemption *(only useful when resources can be saved and restored)* • If a process holding a certain resource is denied further request, that process must release its original resources • Example: A request for rA but rA is hold by B, so B needs to release rA and preempting it so that B can resume its operations later on, like B requesting for rA again in the next attempt after being preempted 2. Direct – prevent the circular wait condition from occurring d) Circular wait *(slow down processes)* • Use a linear ordering of resources type
Avoidance	• Objective: simulate sequence of resource allocations possibilities to determine if there is any resource allocation request that will lead to a deadlock • Two approaches: 1. Process initiation denial • Never start a process if its demands might lead to a deadlock • Avoid running this process at all cost, to avoid all chances of deadlock 2. Resource allocation denial (Banker's algorithm) • To avoid deadlock, this approach looks at the allocation of resources to the processes and define it as safe state or unsafe state • Safe state: there is at least one sequence of resource allocation that does not result in deadlock • Unsafe state: there is no sequence of resource allocation that result in non-deadlock (i.e., all possible sequences result in deadlock)

How to determine safe states?

Step 1: Find the process in which its remaining (in remaining matrix) is ≤ the available resources (i.e., there are resources left to accommodate the leftover processes)

Step 2: Add on the allocated resources to the balance resources to compute the total resource (after allocating to the process's remaining)

Step 3: Repeat step 1 and 2 until it is over all processes are completed

Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

(d) P3 runs to completion

- All process is able to complete based on available resources
- Request is granted based on this set of resources

Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3

- After P1 request for R1 and R3 (b), then no resources can be granted further to the processes (each process need 1 R1 but no available R1 to be granted)
- So problem is because P1 requested for R1 and R3, resulting in unsafe state
- When this happens, block P1 from running until it is safe to grant P1 request
- After blocking P1, it avoids the possibility of deadlock

Detection	• Objective: detect the presence of deadlock and take actions to recover when needed <u>How to determine deadlock?</u> Step 1: Check if there are any available resources that could accommodate/satisfy any current process's requirement Step 2: If there is, then add the allocated resources (of the process that could be given resources to) to the current resources Step 2(a): Otherwise, there isn't any resources that could satisfy any process, means deadlock occurs here (because processes are holding and waiting for resources, but insufficient resources to grant) <u>Recovery strategies for deadlock</u> • Abort all deadlock process • Backup each deadlock process to some previously defined checkpoint and restart all processes (but waste memory cause store checkpoints in history) • Successively abort deadlocked processes until deadlock no longer exists • Successively preempt resources until deadlock no longer exists

Comparison of Deadlock Solutions

Solution	Idea	Pros	Cons
Prevention	Limit access to resources by imposing restrictions on processes (Like block certain processes from accessing resources to prevent deadlock such as preempting a process to release the resources hold)	• Good for resources whose state can be saved and restored (because can easily resume back to pre-preemption state)	• Inefficient due to constantly preempting it for releasing resources • Stops incremental resource requests EG: A already request for 2 units of rA, supposing should request for another rA, but needed to release all rA for preemption, so the next time A request for rA, the resources are longer in incremental way.
Avoidance	Manipulate to find at least one safe path	• No preemption	• Future resources requirements must be known • There must be a fixed number of resources to allocate • No process may exit while holding resources • Process under consideration must be independent with no synchronization requirements • Processes can be blocked for long periods (if cannot find safe path)
Detection	Test periodically all possible resource allocation sequence to determine deadlock	• Grant resources whenever possible • Early detection • Simple algorithm	• Frequent checks consume processor time

Lecture 9: Virtual Memory

- Using logical memory addressing (virtual / fake address) which will be translated to physical address in the actual memory
- One requirement: avoid swapping too often because it may cause trashing where the time used to swap the data is more than time used to execute the instructions data
- Reasons for virtualizing memory:

Reason	Description
Efficiency	• Main memory is limited • Swapping some of the unused memory to secondary storage prevent user from buying additional memory • This swapping is known as paging or segmenting • Allows more processes to run (with lesser memory) • A process can run even if larger than all main memory by diving the process into multiple pages / segments placed in different parts of the main memory (only need to modify in page/segment table) by loading only necessary pages into memory
Security & Concurrency	<u>Individual memory (security):</u> • With virtualization, other process cannot access another process's memory • Because in virtualization, each program only has its own logical address to be translated into physical address, so can't access other process's logical address, strengthening security <u>Shared memory (security + concurrency):</u> • With virtualization, when a process is forked into multiple instances, the program code is only loaded once into the main memory • Then each fork copy has its own logical address (fake address) to be translated into a common physical location • With the usage of mutual exclusion mechanisms (semaphores), the OS can control over who can access the shared memory while making sure sharing the same code (no duplicate)
Flexibility	• When the process is swapped back into the main memory from secondary storage, it is relocated to different part • The page table maps the logical address to the physical address • So, with virtualization, the logical address remains unchanged (ensuring the program's pointer is valid) by only changing the reference to the main memory location in the page table

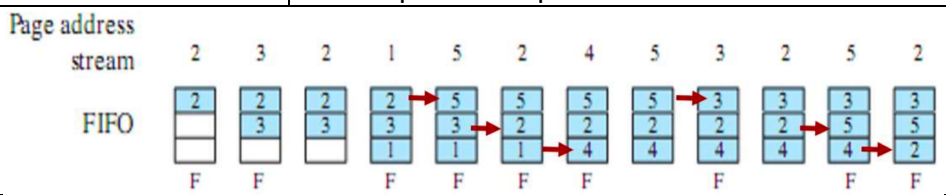
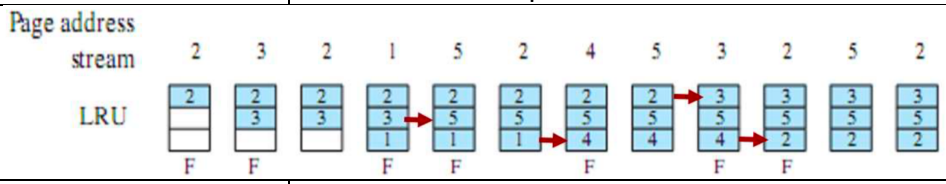
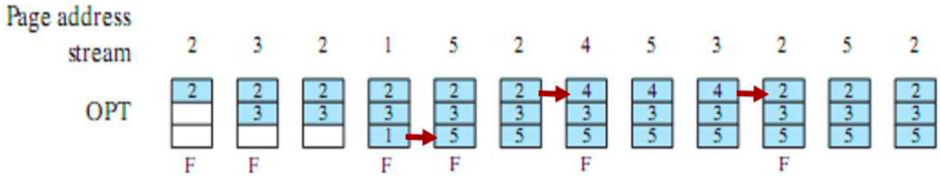
Translation of Logical Address to Physical Address

<table>
<tr><th>Type</th><th>Description</th><th>Issue Caused</th></tr>
<tr><td>Paging</td><td rowspan="2">Consists of page/segment number and offset
Page/segment number is the logical address
Offset is the remaining bits from the start of the page/segment</td><td><u>Page fault</u> ← caused by unable to lookup the page in the physical memory</td></tr>
<tr><td>Segmentation</td><td><u>Segmentation fault</u> ← caused by offsets > segment size, so logical address is invalid</td></tr>
<tr><td>Example</td><td colspan="2">
<table>
<tr><th>Segment #</th><th>Starting Address</th><th>Length (bytes)</th></tr>
<tr><td>0</td><td>660</td><td>248</td></tr>
<tr><td>1</td><td>1752</td><td>422</td></tr>
<tr><td>2</td><td>222</td><td>198</td></tr>
<tr><td>3</td><td>996</td><td>604</td></tr>
</table>
(segment, offset)
1. Physical address of (0, 198) → 660 + 198 = 858
2. Physical address of (1, 515) → 515 > 422 = segmentation fault!
</td></tr>
</table>

Shared Memory

- Same segment memory can be allocated to multiple processes, allowing different processes to access each other's memory
- Requirement is that shared memory must be allocated manually where the size of this "shared memory block" must be specified
- Reason for such requirement is to set a concurrency protection like using semaphores to control over this shared memory block to prevent concurrency issue

Memory Virtualization Methods

<table>
<tr><th colspan="4">Method 1 – Paging</th></tr>
<tr><td rowspan="2">Description</td><td>Memory</td><td colspan="2">Fragmentation</td></tr>
<tr><td>Memory is divided into fixed-size frames</td><td colspan="2">Internal: Yes – there is wasted internal space inside a page when a page is not filled fully
External: No – all frames are fixed-size so there always be a page that fits into the empty frame</td></tr>
<tr><td>How it Works</td><td colspan="3">• Different chunk of data (pages) are then fit into the frames, 1 frame = 1 page
• When data is needed, the pages will be loaded into the frames and swapped out to secondary memory when not needed.
• This allows the physical memory (actual memory) to virtualize more slots
• OS maintains a page table to lookup the location of each page
• Possible issue: page fault – where logical (virtual) memory address sis valid, but the page is in secondary memory (haven't loaded into the frames in memory)</td></tr>
<tr><td rowspan="6">Replacement Algorithms</td><td>Type</td><td>How to Replace</td><td>Inference</td></tr>
<tr><td rowspan="2">First in First out (FIFO)</td><td>Replace the earliest frame that has entered the memory</td><td>• Pages are processed in queues for replacement scheduling
• Not so efficient as compared to others
• Simplest to implement</td></tr>
<tr><td colspan="2"> 
</td></tr>
<tr><td rowspan="2">Least Recently Used (LRU)</td><td>Replace the page that has not been referenced for long time (look back at history to find)</td><td>• Looking back at history exploits the principle of locality because the least referenced page is least likely to be used
• But this looking back cause lots of overhead problems (keep checking at all time)
• Difficult to implement</td></tr>
<tr><td colspan="2"> 
</td></tr>
<tr><td rowspan="2">Optimal Policy (OPT)</td><td>Replace the page that will not be referenced for long time (look forward at future to find)</td><td>• Looking forward at future allows OS to know which page is least likely to be referenced again, preventing unwanted page faults
• Impossible to implement (can't see future)
• Main purpose is to act like a benchmark over the best possible outcome</td></tr>
<tr><td colspan="2"> 
</td></tr>
</table>

<table>
<tr><th colspan="4">Method 2 - Segmentation</th></tr>
<tr><td rowspan="2">Description</td><td>Memory</td><td colspan="2">Fragmentation</td></tr>
<tr><td>Memory is divided into <u>dynamic-size</u> segments</td><td colspan="2"><u>Internal</u>: No – the segments will grow dynamically according to the required size, no wasted space within a segment
<u>External</u>: Yes – small segment will be swapped out from memory, making the hole too small to be fit by another segment (that is larger than the hole)</td></tr>
<tr><td>How it Works</td><td colspan="3">• Variable sizes of segments are placed in empty slots
• Swapping out segments creates a hole for other segments to fill in
• Possible issue: page fault – where logical (virtual) memory address sis valid, but the page is in secondary memory (haven't loaded into the frames in memory)</td></tr>
<tr><td rowspan="4">Replacement Algorithms</td><td>Type</td><td>How to Replace</td><td>Inference</td></tr>
<tr><td>Best-fit</td><td>Replace the block that is closest in size to the request</td><td>• Best to allocate all possible segments</td></tr>
<tr><td>First-fit</td><td>Replace the first block (starting from beginning) that is large enough</td><td rowspan="2">• May cause some segments can't be allocated</td></tr>
<tr><td>Next-fit
(or Worst-fit)</td><td>Replace the next block (starting from the last placement location) that is large enough</td></tr>
<tr><td colspan="3"><u>Processes of 212K, 417K, 112K, and 426K (in order):</u>
<table>
<tr><th>Free Partitions</th><th>First-fit</th><th>Best-fit</th><th>Worst-fit</th></tr>
<tr><td>100K</td><td></td><td></td><td></td></tr>
<tr><td>500K</td><td>(1) Allocate 212K from 500K; Leave a hole of size 288K

(3) Allocate 112K from 288K; Leaves a hole of size 176K</td><td>(2) Allocate 417K; Leave a hole of 83K</td><td>(2) Allocate 417K (83K over allocated)</td></tr>
<tr><td>200K</td><td></td><td>(3) Allocate 112K; Leave a hole of 88K</td><td></td></tr>
<tr><td>300K</td><td></td><td>(1) Allocate 212K; Leave a hole of 88K</td><td></td></tr>
<tr><td>600K</td><td>(2) Allocate 417K; Leave a hole of size 183K</td><td>(4) Allocate 426K; Leave a hole of size 174K</td><td>(1) Allocate 212K; Leave a hole of 388K

(3) Allocate 112K; Leave a hole of 276K</td></tr>
<tr><td></td><td>(4) Cannot allocate for the demand of 426K</td><td></td><td>(4) Cannot allocate for the demand of 426K</td></tr>
</table></td></tr>
</table>

Lecture 10: Interprocess Communication

- Deals with how processes (programs) communicate with each other
- Threads communicate with each other by default because they share the same memory so they can easily communicate within a process
- Example of threads communicating: multiple tabs (threads) in Chrome (process)
- But different related / unrelated process needs a mechanism to communicate with each other because they don't share the same memory space
- Example: communication between Chrome and display server to project the browser on screen needs a method to communicate

Communication Mechanisms

Method	Description
Signals	Purpose: inform a process of an asynchronous event •
Shared Memory	Purpose: to share the same virtual memory segment among multiple process • A memory segment is shared among multiple processes • This shared segment size must be set manually so that OS can utilize a mutual exclusion mechanism to enforce concurrency protection (allowing only one process to modify its data at a time – integrity) • The mutual exclusion resources can be stored in shared memory segment • Methods of using shared memory: 1. System V style: using a common key to access same shared segment 2. POSIX style: using a file descriptor to read/write the shared segment
Pipes	Purpose: •
Sockets	Purpose: to communicate between multiple clients (multiple Chrome window) and one server (display) • Client connects to a socket file (in the filesystem) to reach the server • Server receives and accept the connection • OS then open a file descriptor to allow both client and server to communicate with each other (acting like a socket) • This socket connection can send data in both directions, using two circular buffers (one send-to and one send-from)
Message Queues	Purpose: to send fixed-length messages between processes •

Lecture 11: Security

How OS deals with Security Threats

Threat Type	Description	
Intruders	What is it	• Intruders are unauthorized people that access a system • All OS have a system of verifying if the user is part of the system • This system of verification is the username and password used to login a computer profile • This profile could possibly be intruded by unauthorized people
	How OS deal with it	• OS deals with intruders by having an authentication method in the filesystem • In the filesystem, each file has a unique UID (user id), used to identify the user, along with a GID (group ID) for shared files among multiple users • If a file is only granted permission to UID 1, then only UID 1 can access this file, unless granted with GID permission, preventing unauthorized access (Intruders)
Malware	What is it	<u>3 types of malwares</u>: 1. Parasitic (viruses) • A virus that can't exist outside of an application but is usually embedded into the application's code 2. Independent malwares (worms, bots, trojans) • A malicious utility that can be scheduled remotely and run by the OS • Example: installed an application with malware in a computer that has access over the filesystem of the user (aka modifying files), then user controls remotely via this application to "view" the user's files – invading privacy 3. Replicating • A virus that could produce copies of themselves
	How OS deal with it	<u>Two parts of the OS that deals with it:</u> 1. Using virtual memory to enforce malware **If the program tries to inject malicious code:** • While a program (process) is running, it tries to inject malicious code (malware) into the computer (via the process) • But this is blocked by the OS because every pages / segment in the main memory can only be read by the running process but can't be modified • This means that even when a program is running, the program cannot re-write its program code in the main memory, preventing malwares being executed by the OS

		If the program tries to inject malicious code via other process: • The OS stops this through the virtual memory system because each page / segment can only access its own logical address (unless it is shared memory) • So, injecting the malicious malware via other process is not possible 2. Using IPC to enforce malware •