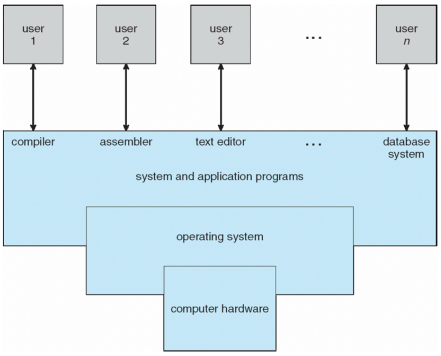
Marce Fuentes

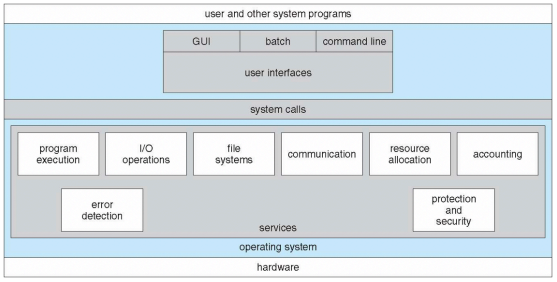
# Operating Systems

## Operating Systems Basics

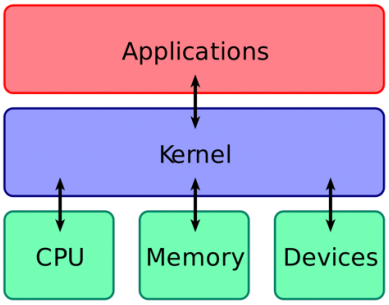
- What is an operating system?
  - A program that acts as an intermediary between the user and the hardware
  - It is a resource allocator
  - It is a control program



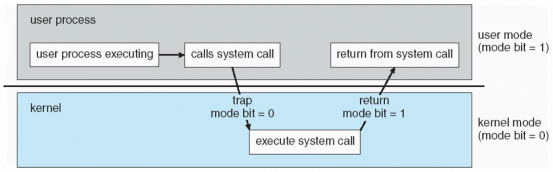

- OS Services
  - The OS has to provide multiple services to the user, and its applications, to use the hardware
  - Most OS provide a GUI and a command line to allow basic interaction of the user with the system




- OS Kernel
  - The most important part of the OS
  - It's the one that actually communicates with the devices, and allocates CPU and memory
    - It also provides services like file management
  - The user never interacts with the kernel



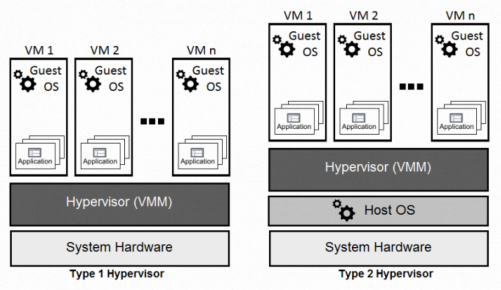

- System Calls
  - System calls are ways for the user programs to ask for an OS service
  - Since they can alter the functionality of the OS, they require to enter a special mode called kernel (or protected mode)



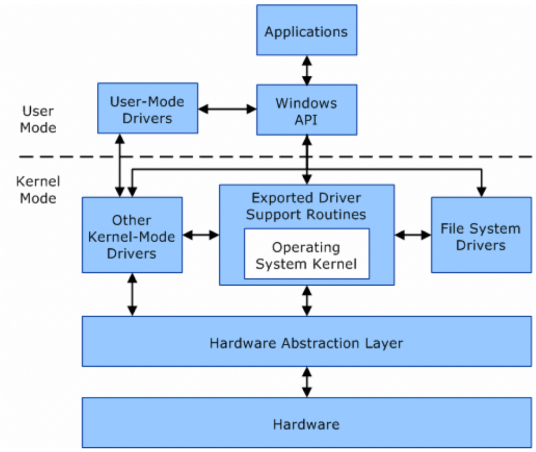
Marce Fuentes

# Virtualization, Consoles, and Kernels

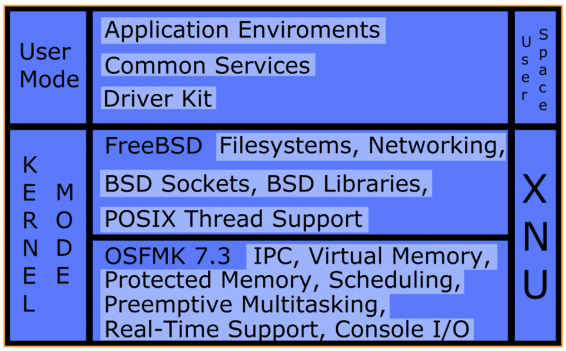
- Virtualization
  - Allows us to run an operating system without real hardware
  - Suffers a performance penalty
  - Host contains the hypervisor, guests are virtualized
  - 2 types depending on the placement of the hypervisor



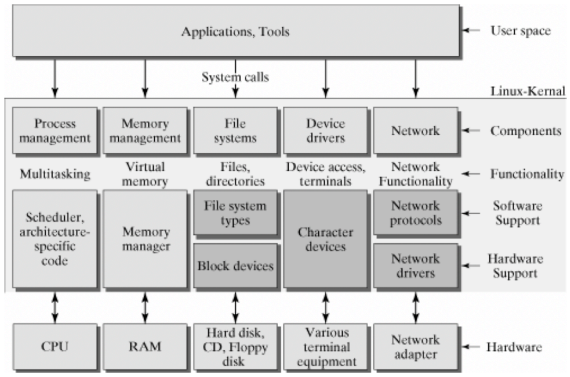

- Windows
  - Uses Windows NT Kernel
  - Current version: Windows 10 2004 update
    - Kernel is 6.4, Vista was 6.0
  - Hybrid Kernel
  - Device drivers run at kernel level




- macOS
  - Uses XNU (X is not Unix) Kernel since 10.0
  - Current version: Catalina (10.15)
  - Hybrid Kernel
  - Combines a Mach micro-Kernel with FreeBSD elements to provide POSIX compliance




- GNU/Linux
  - Uses Linux Kernel
  - Current version: 5.8.1
  - Monolithic, but allows Kernel modules



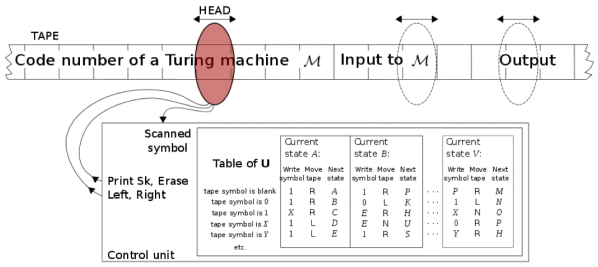

  - Used by almost hundreds of distributions

- User Interfaces
  - Command Line Interfaces
    - DOS, Bash, Z.sh, Power-shell
    - Graphical User Interfaces
      - Modern UI, Aqua, GNOME, KDE, XFCE

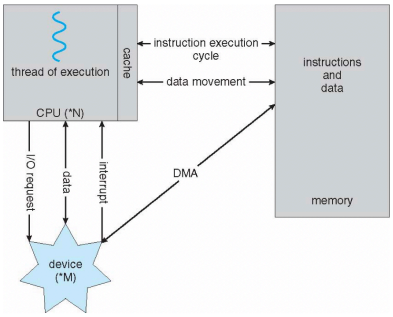
- Common Commands for CLI
  - cd: change directory
  - ls: list directory
  - mkdir: make directory
  - rm: remove file
  - rmdir: remove directory
  - touch: create file
  - ps: process list
  - cp: copy file
  - mv: move file

# Operating Systems and Hardware

- Computer Basics
  - Universal Turing Machines
    - Theoretical machine that can solve/calculate any computable problem
    - Composed of:
      - Read/Write head
      - Infinite data/instructions tape
      - Instructions table
    - Turing completeness indicates that a machine/language can solve any problem an universal Turing machine can




  - Von Neumann Architecture
    - All modern computers follow the Von Neumann architecture
    - Data and instructions are in memory
    - Instructions are loaded and executed
    - The OS is in charge of leading the instructions, move data to the cache, assign execution threads to the CPU and guarantee access to the devices

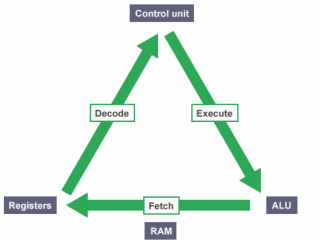



  - Memory
    - Stores data and instructions
    - In modern computers, most memory uses byte addressing, that is, each byte can be individually addressed
    - Maximum address space depends on the processor
      - 32 bits = 2^32 = 4 GB
      - 64 bits = 2^64 = 168 M TB

| Address | Content | Explanation |
|---|---|---|
| 0000 | 01000001 | Letter A in ascii |
| 0001 | 00000000 | Integer number 7 |
| 0010 | 00000000 | |
| 0011 | 00000000 | |
| 0100 | 00000111 | |
| 0101 | 00000001 | C letter in utf-16 |
| 0110 | 00000111 | |
| ... | .... | ... |
| 1111 | 01011010 | NOP operation |

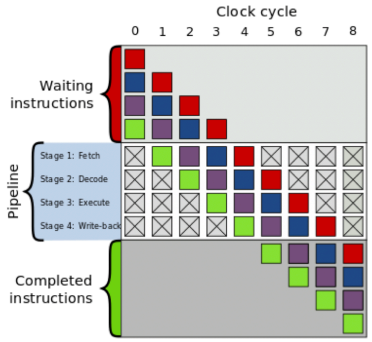
  - Control Processing Unit (CPU)
    - Performs the instructions of the programs
    - Composed of:
      - Registries
        - Contain the info that the CPU uses
        - Instruction Pointer indicates the next instruction to execute
      - Arithmetic Logical Unit
        - Performs math or logic operations
        - The execution unit
      - Control Unit
        - Decodes the loaded instructions from memory


  - Fetch-Decode-Execute Cycle
    1. The next instruction to execute is loaded from the address at Instruction Pointer (IP) registry [Fetch]
    2. The instruction is decoded in the control unit [Decode]
    3. The instruction is executed by the ALU [Execute]
    4. The instruction pointer is incremented by the size of the instruction




  - Considerations
    - 32 bits processors are commonly referred as x86, because most IBM PC compatible processors were modeled after the Intel 8086
    - Speed can be measured in Hertz (CPU CLOCK) or Floating Point Operations per Second (Flops)

- Iterations of the fetch-decode-execute cycle does not necessarily take one clock cycle
- Modern CPU use techniques like pipelining to make them faster



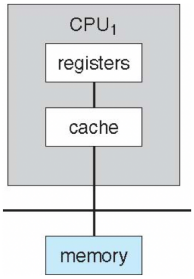

- Devices
  - Perform specific tasks in the computer, with emphasis on the Input and Output of data
  - On the device side, it contains a device controller, which indicates how to perform the I/O tasks
    - It includes registers, and a local storage buffer

  - Characteristics of I/O Devices
    - Types:
      - Block: Like HDD and SDD, can be written in blocks (sectors or SDD pages)
      - Characters: One character at a time (mouse, keyboard)
      - Memory Mapped Files: Files in a memory location
      - Network: Transfer information between hosts (Network Adapters)

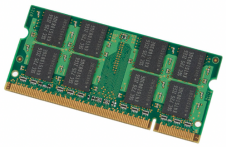
```
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```

- Storage
  - Contains the data (pictures, variables, streams, files, handlers) and the instructions (programs) to execute
  - Storage can be seen as primary, secondary, or tertiary
  - Storage devices can be separated into volatile and non-volatile
  - Has an inverse relationship between speed and capacity


- Registers and Cache
  - Registers are the closest storage to the CPU, and contain the data and instructions to be used immediately
  - Cache acts as an intermediary between the main memory and the registers, to contain data that is used continuously
  - Registers, cache and main memory duplicate information, and have to be kept up to date

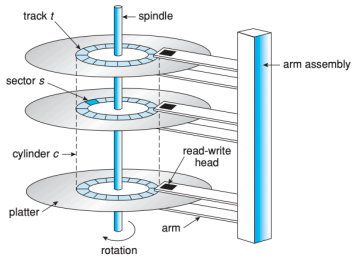



- Main Memory
  - Memory where the kernel and the applications are kept for execution
  - Uses Random Access memory which allows to read of write from any address
  - Physically comes in SIMM and DIMM



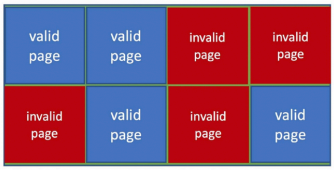
- Secondary Storage
  - Slower storage that allows to keep data without losing information when the computer is turned off
  - It is secondary because it is available to the computer and OS at every moment, but not used as much as the primary
  - Common types are hard disk drives and solid state drives

- Hard Disk Drives
  - Traditional Hard Disk Drives are plates of magnetically-coated material
  - Consider 3 different times:
    - Transfer rate: flow between drive and computer (1 Gb/sec)
    - Positioning time: move disk arm to cylinder (seek), and sector to move under the head (rotational latency); generally 3-12 ms
  - Can result in head crash, that is the head making contact with the cylinder
- Mapping of HDD/SSD Addresses
  - HDD are divided into cylinders, which contain concentric circular tracks
    - Each track is separated into sectors
  - For the OS, the storage is an array of logical blocks, where a logical block is the smallest unit of transfer
    - These are created during low level formatting
  - Logical blocks are mapped to sectors in the disk
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - The mapping is not linear, as there can be bad sectors, and the number of sectors per track is different



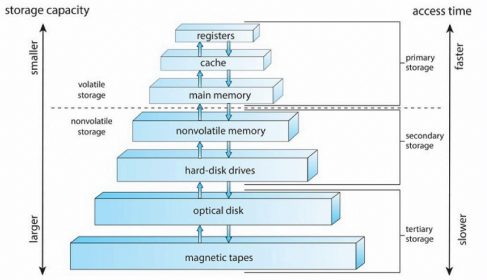

- Solid-State Disks
  - SSD use technology similar to RAM
  - More expensive per MB
  - Maybe have shorter life span — need careful management
  - Less capacity but much faster
  - No moving parts, so no seek time or rotational latency
  - Life span measured in drive writes per day
    - A 1 TB NAND drive with rating of 5 DWPD is expected to have 5 TB per day written within warrantee period without failing

- SSD Organization
  - Organized in pages (different than memory pages), which are similar to sectors
  - Pages have to be deleted first before overwriting, which is a costly operation
  - SSD and OS keep track of the written cells and try not to use pages that have to be overwritten
  - Since writing reduces the lifespan of the pages, it is important to balance the write operations between pages



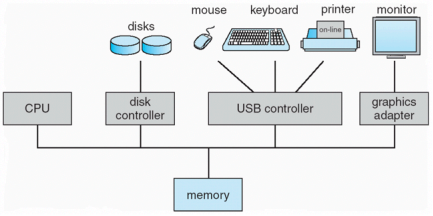

- Tertiary Storage
  - Commonly used for backup or very big files
  - Has to be loaded every time it is needed
  - Includes optical disks (CD/DVD/Blu-Ray) and magnetic tapes

- Comparison of Storage

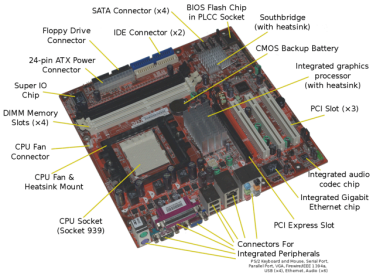



| Level | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Name | registers | cache | main memory | solid state disk | magnetic disk |
| Typical size | < 1 KB | < 16MB | < 64GB | < 1 TB | < 10 TB |
| Implementation technology | custom memory with multiple ports CMOS | on-chip or off-chip CMOS SRAM | CMOS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

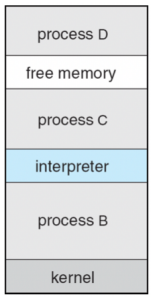
- Connecting all the components
  - The CPU, the memory, and the devices (via their controller) are interconnected by a bus (from busbar and Latin word omnibus, not the vehicle)
  - All the devices share the same main memory, but each can have its own buffers, cache, and registers
  - A Motherboard has 3 main types if buses:
    - Control, Data, Address



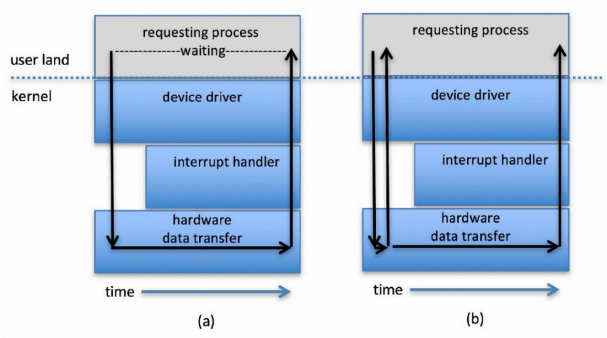

  - There are multiple types of peripheral buses, with different throughputs
    - Peripheral Component Interconnect (PCI)
    - Accelerated Graphic Port (AGP)
    - Universal Serial Bus (USB)
    - Integrated Drive Electronics (IDE)
    - Small Computer System Interface (SCSI)
    - Serial Advanced Technology Attachment (SATA)
    - Serial Attached SCSI (SAS)



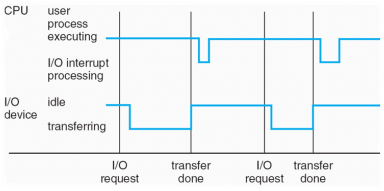

- Input Output and the OS
  - Whenever a program requests access to a device via a system call, the CPU enters into kernel mode
  - The OS is in charge of talking with the devices to get the operations done
  - To learn how to talk with the device, the OS uses a device driver
- OS and Storage
  - The kernel is loaded into main memory by the boot loader or the BIOS, and resides there until the computer is turned off
  - The OS loads processes into main memory, and assigns or frees memory via system calls
  - Also, the OS is responsible for managing the file system, keeping accountability of files and folders usage, and their location in the disk



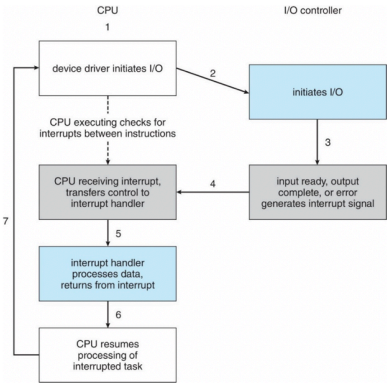

- Blocking and Synchronous I/O
  - Blocking: Refers if the process is suspended (waiting) while the I/O call is done
    - Blocking waits while all the data is read
    - Non-blocking uses buffers to read parts of the data, or returns whatever is available and expects the process to call again
  - Synchronous: When a process asks for an I/O operation, if it waits for the response it is synchronous; while if the operation can be performed at a later time, it is non-synchronous




- Waiting for I/O
  - While waiting for the different devices, two approaches can be used
    - Polling: Asking if its ready, devices have a busy bit.
      - Consumes CPU (Synchronous Non-Blocking)
      - The process can wait between polls (Synchronous Blocking)
    - Interruption: Wait for the device to notify (Blocking and Synchronous or Asynchronous depending on the problem)
    - Direct Memory Access can be also used



- Interruptions
  - Way for the hardware to communicate with the OS
  - Consists of signals in the bus
  - The interrupt handler determines the correct action based on the interrupt vector table

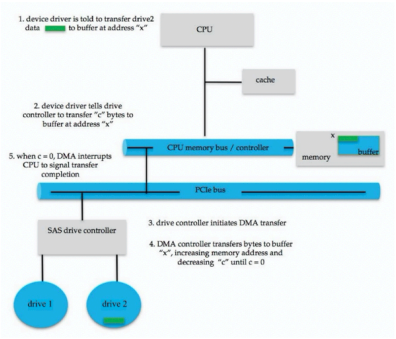



  - Exceptions also generate interruptions
  - Software can generate interruptions by using traps
  - Originally, system calls were done using an interruption (0x80 for Linux, 0x21 in DOS)
  - Interruptions are used by the system clock

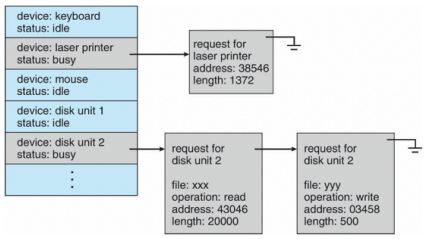
| vector number | description |
|---|---|
| 0 | divide error |
| 1 | debug exception |
| 2 | null interrupt |
| 3 | breakpoint |
| 4 | INTO-detected overflow |
| 5 | bound range exception |
| 6 | invalid opcode |
| 7 | device not available |
| 8 | double fault |
| 9 | coprocessor segment overrun (reserved) |
| 10 | invalid task state segment |
| 11 | segment not present |
| 12 | stack fault |
| 13 | general protection |
| 14 | page fault |
| 15 | (Intel reserved, do not use) |
| 16 | floating-point error |
| 17 | alignment check |
| 18 | machine check |
| 19–31 | (Intel reserved, do not use) |
| 32–255 | maskable interrupts |

- Clocks and Timers
  - Modern OS are Interrupt driven
    - The clock indicates how many times per second an interruption is generated
    - These interruptions allow to check the time and update the OS clock
    - These interruptions allow also to determine how long a process will execute
    - The clock depends on a variable in the OS, not the number of ticks of the CPU

- Direct Memory Access
  - Instead of using the CPU for transferring between memory and devices, use the DMA Controller
  - While it occupies the bus, it leaves the CPU free to do other computation
  - Interrupt-driven



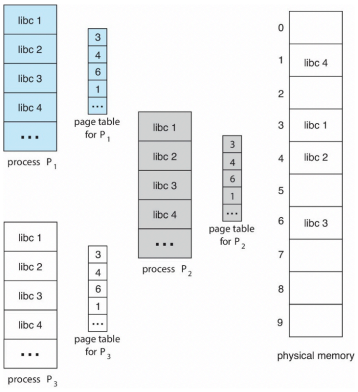

- Kernel and I/O
  - The kernel schedules all the calls to the different I/O devices
  - It also implements the following elements:
    - Caching, buffering, spooling, error handling, protection



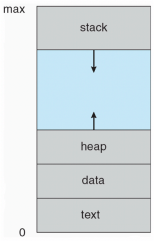
# Processes

- What is a program?
  - A program is a set of instructions that tells the computer how to do something, usually in secondary storage
  - A program passes through multiple states to transform source code into an executable file
  - There are multiple types of executables, depending on the OS:
    - Windows uses the Portable Executable format (PE)
    - Mac OS/iOS uses Mach object file format (Mach-o)
    - Linux uses Executable and Linkable Format (ELF)

- Compiling, Linking and Loading
  - When compiling, the source code is converted to object code
  - Linking adds the functions from the libraries and syscalls
    - Static linking copies the libraries into the executable
    - Dynamic linking adds a stub for calling the library
      - The stub is changed by the function address
  - Loading copies the program into memory
  - A program in memory is called a process




- A process in memory
  - Text: The code of the program
  - Data: Global and initialized data
  - Stack: Temporary data
  - Heap: Memory allocated dynamically

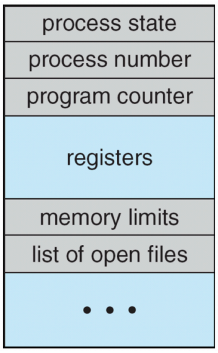



- Example of a function in memory

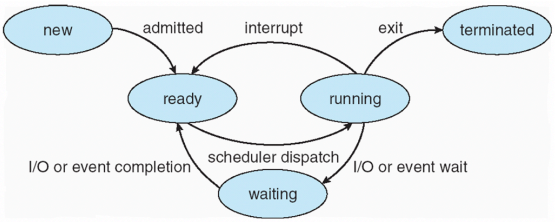
```
d=5
5 int function(int a, int b) {
6       c=a+b;
7.      return c;
}

I int main( ) {
2.      b=6;
3.      g=function(b, d);
4       return 0;
}
```

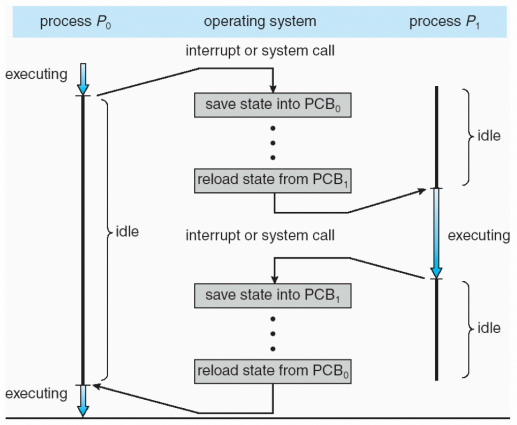
- Process Control Block
  - Process state, running, waiting, etc
  - Program counter: location of instruction to next execute
  - CPU registers: contents of all process-centric registers
  - CPU scheduling information: priorities, scheduling queue pointers
  - Memory-management information: memory allocated to the process
  - Accounting information: CPU used, clock time elapsed since start, time limits
  - I/O status information: I/O devices allocated to process, list of open files



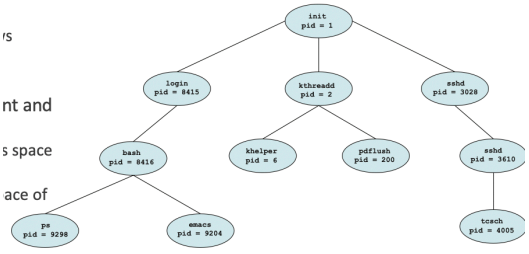

- Process States
  - New: The process is being created
  - Running: The process is being executed
  - Terminated: The process finished
  - Ready: The process is waiting to execute in the CPU
  - Waiting: The process is waiting for an event to occur
  - Suspended: The process is taken out of memory



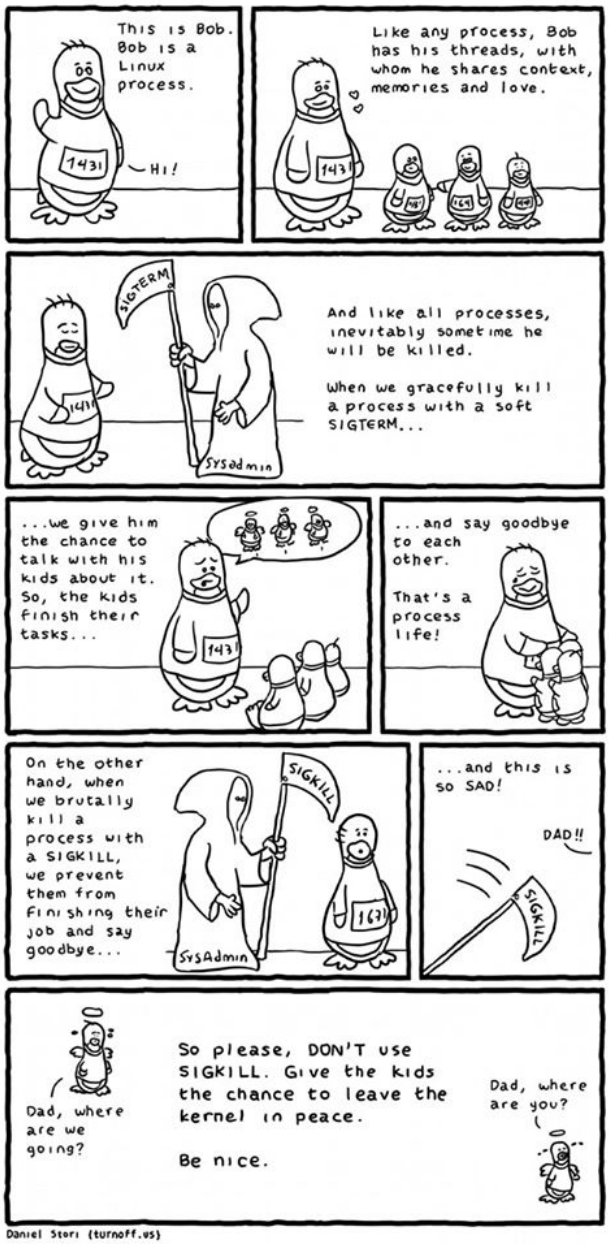
- Context Change




- Process Creation
  - Most processes have a parent process, which creates or invokes them
    - Systemd, initd in Linux, System in Windows
  - Processes are identified by a Process ID
  - Children share the resources of the parent and the program
    - Fork creates a child with the same address space of the parent
    - Exec replaces the program and address space of the child

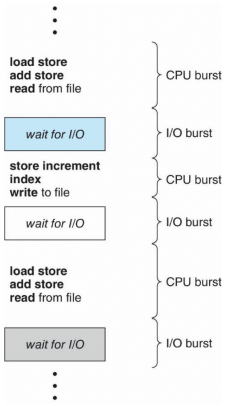



- Process Termination
  - A process ends by calling exit, and allows the OS to deallocate their resources
  - When a process exits, it notifies its parent
  - Parents should be waiting for their child processes to finish
    - If a parent finishes, some OS enforce that all the children finish too
    - If no one waits for the child to exit, it becomes a zombie
    - If the parent ends and didn't issue the wait call, the child becomes an orphan

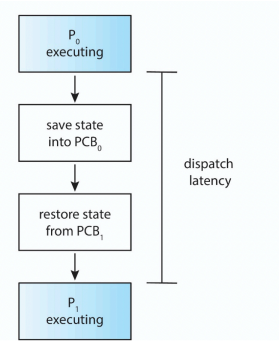


# CPU Scheduling

- Processes and the CPU
  - Processes work in bursts, using the CPU and IO devices, one or the other
  - Preemption is when a process leaves the CPU (context changes and terminations)
  - A CPU/Program can block preemption, so it only leaves the CPU when it terminates




- Dispatcher
  - The dispatcher is in charge of giving the control of the CPU to the process selected by the short term scheduler
  - This includes:
    - Save the state of the PCB
    - Load the next process PCB, including registers
    - Switching to user mode
    - Go to the proper location of the next instruction as pointed by the IP
  - All of these times "wasted" are known as the dispatch latency




- Scheduler
  - The CPU Scheduler is in charge of selecting a process from the ready queue, and allocating a CPU core to it
  - Short term scheduler: Decides next process from the ready queue
  - Long term scheduler: Decides which processes are going to be loaded next (new state)
  - Middle term scheduler: Decides which processes should go into suspended state

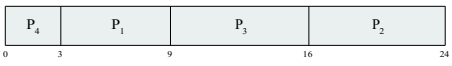
- Calling a scheduler
  - The schedulers can be called when needed ir in intervals of time
  - Scheduling decision may happen when a process:
    - Switches from running to waiting state
    - Switches from running to ready state
    - Switches from waiting to ready
    - Terminates
      - 1 & 4 depend on the process that is running
      - 2 & 3 depend on other processes or the OS

- Scheduling Criteria
  - CPU utilization: Keep the CPU as busy (maximize)
  - Throughput: Number of processes that complete their execution per time unit (Maximize)
  - Turnaround time: Amount of time to execute a particular process (Minimize)
  - Waiting time: Amount of time a process has been waiting in the ready queue (Minimize)
  - Response time: Amount of time it takes from when a request was submitted until the first response is produced (Minimize)

- First Come, First Served
  - Allocates the CPU in the order the process are in the ready queue
  - Is non-preemptive, process executes until termination


- Shortest Job First
  - Takes into account the length of the process CPU burst
  - Allocates the CPU to the process with the next shortest CPU burst
    - Preemptive version is known as Shortest Remaining Time
  - Hast the shortest waiting time of all schedulers, however knowing in advance the length of the CPU burst is impossible

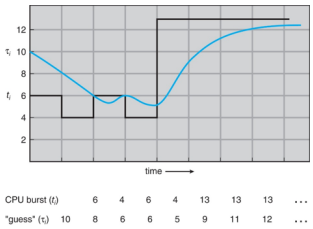
| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |



- Prediction of next burst length
  - Uses a weighted average, where alpha represents the weight

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_{n-1}.$$

  - Depending on the weight is how much emphasis we will give to the past
  - SJF uses the next process with the shortest estimated time




- Example with arrival time
  - Processes arrive at the processor at different times

| Process | *Arrival* Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |


- Round Robin
  - Each process is given a time quantum
    - If the quantum expires the process is taken out of the processor and into the end of the ready queue
    - If quantum too big = FCFS
    - If quantum too small = Too much time in dispatcher latency
    - Each process gets 1/quantum duration of CPU

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

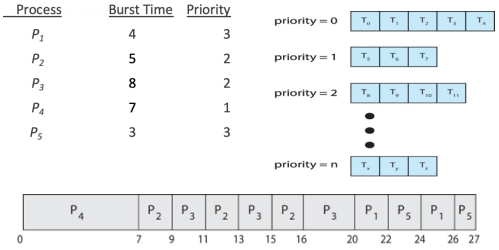

  - About the quantum
    - Context change usually takes 10 microseconds
    - Quantum are 10 to 100 milliseconds generally

- Priority Scheduling
  - Each process has a priority
  - The CPU is given to the process with the highest priority
  - Pure SJF is priority scheduling where the priority is the inverse of the burst time
  - Low priority processes may starve unless aging is implemented

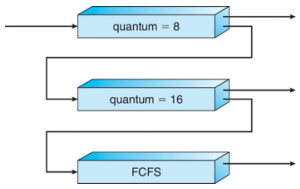
| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |


  - Priority with Round Robin
    - Processes with the same priority use Round Robin
    - There is a queue for each process

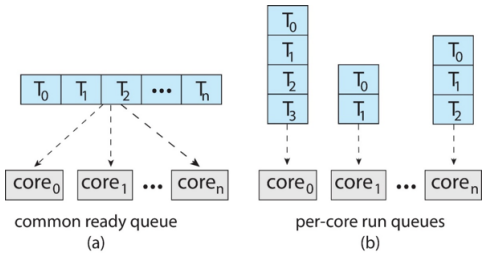
| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 4 | 3 |
| $P_2$ | 5 | 2 |
| $P_3$ | 8 | 2 |
| $P_4$ | 7 | 1 |
| $P_5$ | 3 | 3 |




- Multilevel Feedback Queue
  - A process can move between different queues
  - Each queue has its own quantum, priority and scheduler
  - It is important to define when a process enters or leaves a queue

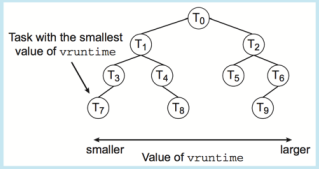



- What happens with multiple processors?
  - All threads may be in a queue or in a per processor queue



- Linux Scheduling Example
  - Prior to 2.5 kernel, used UNIX Scheduler
  - 2.5 version used a O(1) scheduler, preemptive with priorities, and Round Robin
  - Higher than 2.6.23: Completely Fair Scheduler
  - Uses priorities and quanta based on CPU time
  - Uses nice value from -20 to +19 (Nice=Shares)
  - Running time depend on priority
  - The next process is the one that has less running time in the CPU

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



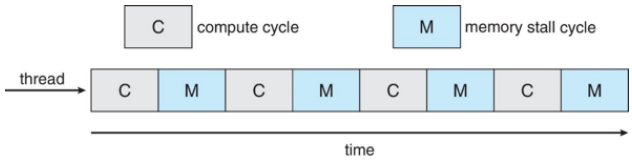

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where $N$ is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

- Windows Scheduling Example
  - Priority queue
  - Runs next process in the highest priority queue
  - If no process is available, runs idle thread
  - Priority can be configured by the user

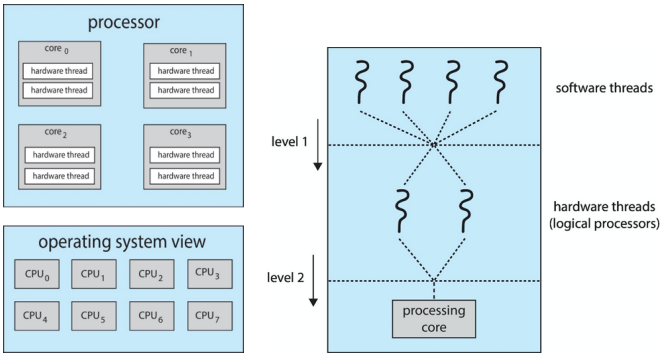
| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

# Threads

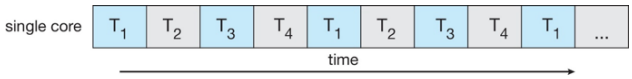
- Multicore processors
  - Recently, processors have multiple cores that can be assigned
  - These cores are faster than Memory Access, so it can happen that processors have to wait for a memory or cache operation
  - A processor can be multithreaded
    - That is, run multiple processes per core using memory waiting time (memory stall)



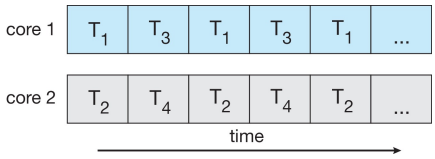

- Multi Threading (Concurrency)



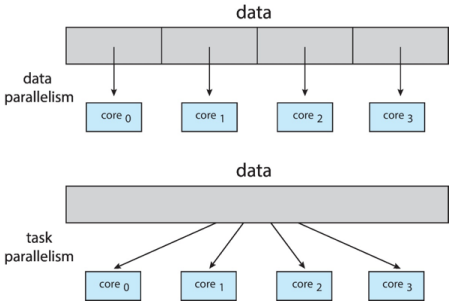

- Parallelism vs. Concurrency
  - Concurrency: More than 1 task/process advances in its execution




  - Parallelism: 2+ processes are running simultaneously



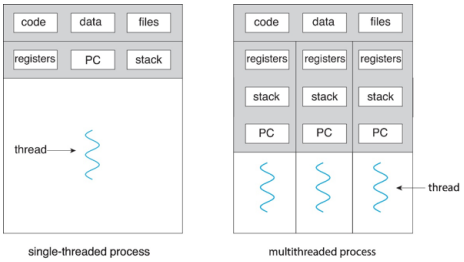

- Types of parallelism
  - Processes and threads can do 2 types of parallelism
    - Data parallelism separates the data but executes the same instruction
    - Code parallelism runs different instructions at the same time




- Ahmdal's Law
  - Why having multiple processors doesn't result in a lineal speedup

  speedup $\leq 1/(S+(1-S)/n)$
    - S is the serial percentage of the code, and n is the number of processors
    - As n approaches infinity, the speedup approaches 1/S
    - A speedup of 1x means the same time if needed to execute the program
    - To obtain the time, divide the original time between the speedup

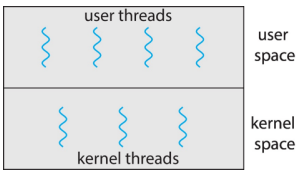
-Threads
- Run multiple things in the same application
- Threads run within application
- Multiple tasks like
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight



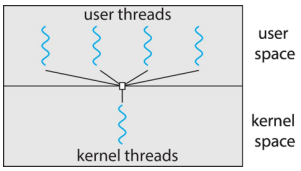

- Advantages of threads
  - Responsiveness
    - Waiting part of the application
  - Resource sharing
    - Easier than shared memory
  - Economy
    - More lightweight than a fork

- Scalability
  - Allow to use multicore systems

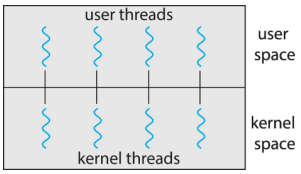
- Types of threads
  - User threads: Management done by user-level threads library
    - POSIX Pthreads, Windows threads, Java threads
  - Kernel threads: Supported by the Kernel
    - Windows, Linux, MacOS X, iOS, Android



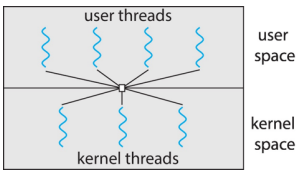

- Many-to-one
  - 1 kernel thread for all user threads
  - Creates a bottleneck if the system call has to wait
  - Not really used




- One-to-one
  - For each user thread, a kernel thread is created
  - Allows for greater concurrency of all thread models
  - Thread number may be limited by the OS
  - This is how Linux and Windows work




- Many-to-many
  - Has a pool of available kernel threads that get shared
  - If a process needs a new system thread and there is not one available, enters to a waiting queue
  - Not really used



# Synchronization

- Why do we need synching?
  - We know that processes can share memory
  - Threads share text, data and heap sections
  - Processes can run in a concurrent or parallel manner
  - There is preemption, connect changes and interruptions
  - Processes are interleaved using Hyperthreading
  - What happens if a process stops at the wrong time?

- Producer-consumer

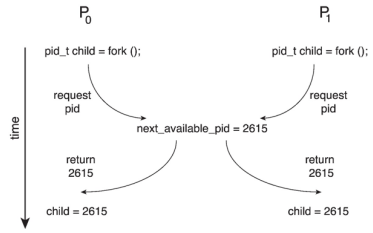
```
while (true) {
  /* produce an item in next produced */

  while (counter == BUFFER_SIZE)
  ; /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}

while (true) {
  while (counter == 0)
  ; /* do nothing */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
      counter--;
  /* consume the item in next consumed */
}
```

- Murphy's Law
  - counter++ could be implemented as
    - mov eax, [counter]
    - add eax, 1
    - mov [counter], eax
  - counter-- could be implemented as
    - mov eax, [counter]
    - sub eax, 1
    - mov [counter], eax
  - Consider this execution interleaving with "count = 5" initially:
    - producer execute mov eax, [counter] {eax = 4}
    - producer execute add eax, 1 {eax = 5}
    - consumer code before counter—
    - consumer execute mov eax, [counter] {eax = 4}
    - consumer executesub eax, 1 {eax= 3}
    - producer execute mov [counter], eax {counter = 5}
    - consumer execute mov [counter], eax {counter = 3}

- Race condition
  - When 2+ processes contend to change a variable and leave it in an inconsistent state
  - Difficult to replicate due to the seemingly stochastic nature of processes execution time
  - Thanks to this multithreaded applications require a careful implementation
  - Special care needed in critical sections



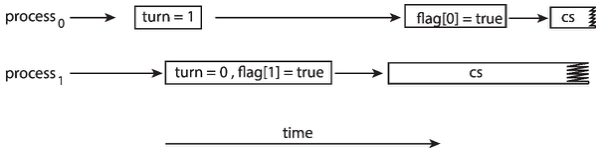

- The critical section problem
  - Happens when there are shared variables, files, tables, or other resources that can be left in an inconsistent state by a specific part of the code
  - Only 1 process should be in the critical sections, while others should wait and ask for permission to enter
  - Solutions to the critical section problem must consider:
    - Mutual exclusion
    - Progress
    - Bounded time waiting

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

- Peterson's solution
  - Use 2 shared variables, turn and ready
  - For a process to enter its critical section, it must be ready to enter, and be its turn
  - Its a nice programmatically solution that grantees mutual explosion, bounded waiting and progress

```
//when process A wants to enter
Ready[A]=true;
Turn=B;
While(ready[B] and turn==B);
Do nothing
Enter critical section
Get out of critical section
Ready[i]=false;
```

- … Sounds good, doesn't work
  - Modern compilers reorder the instructions for efficiency
    - Compilers are so good at this, that direct assembly code optimizations tend to perform worse
  - This happens with non-related instructions and variables
  - In multithreaded environments it may cause inconsistencies




- Addressing synchronization in hardware
  - CPU can block interruptions or preemption so an instruction finishes (affects progress and bounded waiting)
    - Memory barriers: Allow changes in memory to be propagated and known to all the processors, while blocking parts of memory from changes
    - Atomic operations: Set of operations that have to be completed or aborted if there is a context change
    - Atomic variables: Variables that cannot be left in inconsistent states
  - Synchronization is generally implemented in software by the programmers

# Software Synchronization

- Mutex Locks
  - Perform mutual exclusion between variables
  - Provide an atomic operation for locking and unlocking
  - The only thread that can call unlock is the one that called lock
  - If the mutex cannot be locked, the process goes to sleep
  - Mutex's have a busy waiting counterpart called spin-lock

```
while (true) {
  acquire lock
  critical section
  release lock
  remainder section
}
```

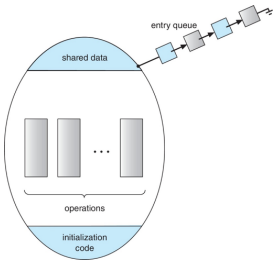
- Semaphores
  - Provide more sophisticated methods than mutex locks for synchronization
  - Consists of 2 calles, P (wait) and V (signal/post)
  - Unlike mutex, different processes may wait and signal

```
Wait(S){
        While(s<=0){
          wait;
        S--;
}

Signal(S){
        S++;
}
```

  - Types of semaphores
    - Binary: Only 1 thread can execute at a time
      - Almost the same as a mutex
    - Counting: Multiple threads can execute at a time, locking and unlocking

- Monitors
  - Abstract data type where synchronization is already provided
  - Instead of accessing the locks, functions are created where the locks already are in place
  - Functionally, only one process is inside the monitor



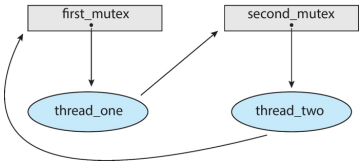

- Bounded-Buffer Problem
  - A limited buffer, with producers and consumers modifying the buffer
  - Can result in race conditions
  - Can use 3 semaphores, indicating the number of empty slots, the number of full slots, and allowing only 1 producer or consumer to be in the buffer at a time
  - Used in cloud computing

- Readers-Writers problems
  - Shared data between processes/threads
  - Some processes only read but don't modify the data
  - Other processes write and modify
  - How to guarantee exclusion between the writers and the readers?
    - Have a count of how many readers there are
      - Writer waits until no reader is inside
    - Make all readers wait when a writer is ready
  - Both solutions may lead to starvation

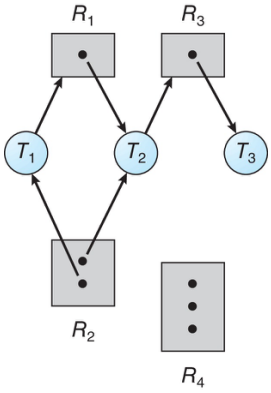
- Deadlock
  - Situation in which 1+ processes are waiting for resources that will not be available, and cannot continue executing
  - Rarer than race conditions, but still need to be addressed

  - Conditions:
    - Mutual exclusion: Only 1 process can use a resource at a time
    - Hold and wait: 1 process has a resources and is waiting for another
    - No preemption: A reserved resource by a process can only be relinquished voluntarily
    - Circular wait: There are n processes where P0 is waiting for a resource that P1 is using, P1 waits for P2, and so on, until Pn waits for P0

  - Deadlock in a Multithreaded Application
    - Deadlock is possible if thread 1 acquires first_mutex and thread 2 acquires second_mutex
      - Thread 1 then waits for second_mutex and thread 2 waits for first_mutex
    - Can be illustrated with a resource allocation graph:

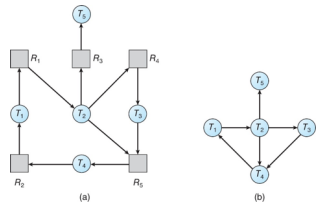



- Resource Allocation Graph
  - A set of vertices V, and edges E
  - V is partitioned into 2 types:
    - T is the set consisting of all the threads in the system
    - R is the set consisting of all resource types in the system
  - Edges from thread to resource are requests
  - Edges from resource to thread are assignments




- Deadlock prevention
  - Eliminate a condition for a deadlock
  - Mutual exclusion
    - Allow read only resources
    - However, cannot be eliminated for critical sections
  - Hold and wait
    - Either allocate all the resources at the start, or allocate all the resources when it has none
  - Preemption
    - Allow processes to relinquish their resources when commanded
    - If a process cannot allocate all the resources it needs, it has to release them
    - Have a waiting table where allocated resources on waiting processes are stores
      - If needed, deallocate them
  - Circular wait
    - Give information to scheduler so an order without deadlocks is guaranteed

- Deadlock Detection
  - This is used when no prevention strategy is used
  - A detection algorithm is necessary
    - Maintain a wait graph and look for loops in the graph
    - The invocation of the algorithm needs to be timed correctly
  - Also, the system needs to recover from deadlocks



- Deadlock Recovery
  - If processes are deadlocked, they need to be restarted/ rollbacked
  - A rollback would require to maintain the state of processes, which currently is not viable
  - Restart requires to decide if restarting all deadlocked processes or just 1
    - Restart can happen with associated costs for each process
    - Restart in order of execution until deadlocked is solved
    - If the same process is always selected for deadlock, it could lead to starvation

# Memory

- Memory and Processes
  - Memory doesn't distinguish between code and data
  - Instructions and data have to be loaded from memory to the registers
  - Access to registers usually takes one cycle, access to memory may take more (stall)
  - Memory only sees read and write request at different addresses

- Protection
  - A process should not modify other processes memory space
  - Protection given by checking the limits between memory base and memory limit
  - Memory used by the OS usually sits at the top or the bottom

- Address assignment
  - Processes need to know the address for instructions and data
  - Addresses can be static if the base loading address is known at programming time
  - Generally, addresses are re-locable at loading or execution time

- Logical vs Physical Addresses
  - Needed for relocatable code
  - The physical address is the one in main memory, logical is the one the process sees
  - Changes when executing the program
  - Memory Management Unit is in charge of translating logical to physical addresses

- How is memory allocated?
  - Contiguous Allocation
    - Memory in 2 blocks, OS and user
    - Process memory assigned next to each other
    - Can grow dynamically
    - Uses relocation registers to separate the processes
  - Variable Allocation
    - Each process is allocated memory as needed
    - OS keeps a list of allocated sections as "holes"
- Dynamic Storage Allocation Problem
  - First fit: Allocate the first big enough hole
  - Best fit: Allocate the first hole that is big enough
  - Worst fit: Allocate the largest hole
    - First fit and best fit have the best speed and storage utilization
- Fragmentation Problem
  - External fragmentation: There is space for a new process in memory, but the space is not contiguous
  - Internal fragmentation: Allocated memory is more than needed, resulting in wasted memory
    - These problems are solved by relocation, but only can be used if memory is allocated dynamically
- Paging
  - Divide physical memory in equal sized chunks called frames
  - Divide logical memory in chunks of the same size of frames, called pages
  - A program with N pages, needs N free frames to run
  - Keep a page table and the address of free frames
  - Problems with Paging
    - Requires extra OS resources to keep the page table
    - Assignation of pages to frames
    - Does not solve the internal fragmentation problem
      - Worst case scenario, a process requires N pages+1 byte
      - In average, half a page gets wasted

- Free Frame List
  - Used by the OS to keep a record of the free frames
  - Usually a linked list
  - When a new frame is requested, a free frame is taken out of the list, and generally zeroed out
  - At startup, all memory is in this list

- Page Table Lookup
  - Requires 2 memory accesses, one for the page table and one for the instruction/data
  - Can be solved by using Translation look-aside buffers (TLBs)
  - Do not keep entire table, so it can hit or miss

# Virtual Memory

- Virtual, Logical, and Physical Memory
  - Physical memory is the real one, and uses frames
  - Logical memory is the view of the process, and is mapped to physical
  - Virtual memory allows to separate the 1-to-1 mapping of logical to physical

- Demand paging
  - Brings entire process into memory or pages when needed
  - When a page is needed, if the reference is not memory resident, loads the required page
  - Necessary to check and restore the state of the process before loading a page

- Valid-Invalid Bit
  - Can be used to indicate if the page is read or write
  - Commonly used to indicate which pages are memory resident and which are in the swap partition
  - If a page to be used is invalid, create a page fault

- Page Fault
  - Operation that indicated that a required page is not in memory to load it
  - First searches for the page in storage, loads it, and restarts the instruction that caused the page fault
  - A process with all the pages in memory causes no faults
    - Pure demand paging causes faults for each one

- Page Replacement
  - What happens if all frames are used?
  - Memory is usually "overbooked"
  - Process for page replacement:
    - If free frame found use it
    - If not, find a victim, take it out of memory and load the new page

- Performance Overhead
  - 3 actions needed for a page fault:
    - Service the interrupt
    - Load the page
    - Restart the process
  - Effective Access Time: (1-p) memory access time + (p)(page overhead + swap in + swap out)

- Thrashing
  - If a process does not have enough pages, it has a high rate of page faults
  - Since in page fault the process blocks, the scheduler might be led to believe that more processes can be handled
  - Eventually, no process is doing any computation, all the time is spent doing page replacement

- Optimizations
  - Use a dedicated swap partition
  - Copy entire process in swap space
  - Use dirty bits (if modified or not)
  - Copy on write
  - Reclaim read-only pages

- Locality
  - A process generally uses memory from near locations (think next instruction)
  - This is known as the principle of locality or locality of reference
  - Spatial locality: The next memory address accessed will be near the current one
  - Temporal locality: In a short time, the same memory space is going to be accessed multiple times
  - If we can guarantee a whole locality in memory, we reduce the number of page faults

- Working Sets
  - The number of pages that the process needs to work
  - A high number encompasses several localities
  - A low number of pages requires multiple page faults for the same locality
  - An infinite working set is the same as loading the whole process

- Frame replacement algorithms
  - Need to take into account that pages can be reused
  - The objective is to minimize the number of page faults
  - Evaluation is done on strings that represent frames

- First-In First-Out
  - The pages are assigned in the order they come
  - Works under the assumption that older pages will be needed less

- Belady's Anomaly
  - Until 1969, it was believed that more frames generated less page faults
  - Belady's anomaly indicated that this is not true for some special cases
  - Not applicable to stack-based replacement

- Optimal Algorithm
  - Based on replacing the page that is going to be less used in the future
  - Same problem that SJF, needs to know the future
  - Useful for theoretical comparisons

- Least Recently Used
  - The victim is the page that has been used less
  - Uses the past instead of the future
  - Better than FIFO, but worse than Optimal

- LRU Implementation
  - Counter:
    - Keep the time when it was last used

    - When needed, search for the page that has not been used the longest
  - Stack:
    - Move the most recently used page to the top of the stack
    - The victim is at the bottom

- LRU Approximations
  - LRU is slows, so it has been implemented with different approximations
  - Reference bit: If used, mark the bit with 1, and replace pages with 0
  - Second chance:
    - FIFO, but when a page is selected as a victim, check the reference bit
      - If 1, put 0
      - If 0, replace the page
    - Enhanced Second Chance
      - Uses 2 bits, reference and modification to decide the next victim
        - Reference 0, Modification 0
          - Best page to replace, not recently used nor modified
        - Reference 0, Modification 1
          - Not quite as good as before, because it needs to be copied to memory when swapping out
        - Reference 1, Modification 0
          - Recently used, but without modification, so it is faster to swap out
        - Reference 1, Modification 1
          - Recently used and modified, the worst case because probably is going to be used again, and needs to be copied to memory when swapping out

# Disks, Files, and Filesystems

- Booting into a OS
  - Boot block initializes system
    - The bootstrap is stored in ROM, firmware
    - Bootstrap loader program stored in boot blocks of boot partition
  - The Master Boot Record contains the information about the partitions and where the kernel of each Bootable partition is

- Storage Device Management
  - Physical formatting or low level formatting divides the disk into sectors
    - Dividing a disk into sectors that the disk controller can read and write
      - Each sector can hold header information, plus data, plus error correction code (ECC)
      - Usually 512 bytes of data but can be adjusted
  - To used a disk to hold files, the OS still needs to record its own data structures on the disk
    - Partition the disk into 1 or more groups of cylinders, each treated as a logical disk
    - Logical formatting or "making a file system"
    - Raw disk access for apps that want to do their own block management, keep OS out of the way (DB for example)

- File System
  - Part of the OS in charge of files
  - Each file system is known as a volume
  - Addresses the following requirements
    - File storage and retrieval
    - Directory structure
    - Attributes
  - Parts:
    - Device driver and I/O Control: Physical part
    - Basic File System: Simple interaction with hardware
    - File-Organization module: Translation between logical and physical
    - Logical file system: Organization for the user

- Files in the File System
  - For each volume, the FS has to keep several records
  - If the partition is used to boot, then the FS needs to keep a Boot Control Block
    - The MBR uses the info of this partition to boot the OS
    - If the Boot-loader cannot read the FS, then this partition is not bootable
  - Volume Control Block: Contains information about the files and their locations in the HDD
    - Master File Table for Windows NTFS
    - Superblock for Unix File Systems

- Files
  - Contiguous logical address space
  - Usually of 2 different types
    - Plain text
    - Binary (Data and programs)
  - Have multiple attributes associated
    - Name, identifier, type, location, size protection time, data, and user identification

- Opening a File
  - To open a file, it has to be searched in the secondary storage
  - The OS loads a copy of the File Control Block into the opened files list or increases the process count
  - In the PCB, a pointer to the FCB is added
- Open Files
  - In the PCB, the position within the file is kept for reading and writing
  - The OS can provide locking:
    - Mandatory and advisory
  - When the file is closed, the contents in memory have to be copied to the HDD if not committed

- File Handlers/Descriptors
  - For each process accessing a file, its PCB contains a pointer to the bytes read or written
  - The access to a file can be sequential or direct
  - In Windows, this structure in the PCB is known as the File Handler, in Linux as the File Descriptor

- Reading and Writing
  - Uses the file as consecutive logical addresses
  - Sequential Access: Reads the file in order
    - The read and write calls indicate how many bytes from the current position have to be accessed
  - Direct Access: Has methods for moving the current position within the file
    - Can access specific positions without traversing previous ones

- Files Allocation: Contiguous
  - Files can be allocated in the HDD in contiguous spaces
  - Just like memory pages, this can lead to external fragmentation

- Allocation: Linked
  - Each sector uses some bits to indicated a pointer to the next sector
  - Requires extra storage
  - Has no external fragmentation
  - Makes direct access slower

- File Allocation Table
  - Like indexed, but the pointers are in a table
  - Faster in memory
  - Originally used by MS-DOS
  - Common implementation is FAT 32

- Indexed allocation
  - Uses a page to contain an index table for each sector of the file
  - Requieres the overhead of an index block
  - Most useful for direct access without creating external fragmentation

- HDD Scheduling
  - The OS is in charge of using the resources efficiently, including the HDD
  - One way to add efficiency is by reducing seek times in the HDD
  - Since many processes require to use the HDD, the OS maintain a queue of requests that include
    - Read or write mode, number f sectors to transfers, memory address to save the data
  - HDD is in charge of translating the logical address to the physical one, OS is in charge of indicating the addresses needed
  - There are many sources of disk I/O request
    - OS, System processes, Users processes
  - I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
  - OS maintains queue of requests, per disk or device
  - Idle disk can immediately work on I/O request, busy disk means work must queue
    - Optimization algorithms only make sense when a queue exists
  - In the past, OS responsible for queue management, disk drive head scheduling
    - Now, built into storage devices, controllers
    - Just provide LBAs, handle sorting of requests
      - Some of the algorithms they use described next

- First-Come First-Served
  - Equals to no scheduling
  - Petitions are addresses as they come
  - As with other schedulers of this type, this results in time loss while waiting
- SCAN
  - The disk starts at one end and moves to another, attending requests as it advances
  - Better than FCFS as petitions are served faster
  - But note that if requests are uniformly dense, petitions at the other side of the disk wait the longest

- C-SCAN
  - Provides a more uniform wait time than SCAN
  - The head moves from one end of the disk to the other, servicing requests as it goes
    - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
  - Treats the cylinders as a circular list that wraps around from the last cylinder the the first one

- Directories
  - For each directory, the FS needs to know which files and subdirectories are inside
  - A collection of bodes that contain information about the files
  - Used by the user to administer its space
  - Have a sense of "locality" where related files are usually put in the same directory
  - The directories can be structured in different manners

- Tree Structured Directory
  - Better than single directory
  - Allows to keep a better organization
  - Deleting a directory in a tree requires policy decisions:
    - Delete only empty directories
    - Recursively delete all the subdirectories

- Error detection and correction
  - Fundamental aspect of many parts of computing (memory, networking, storage)
  - Error detection determines if a problem has occurred
    - If detected, can halt the operation
    - Detection frequently done via parity bit
  - Parity one form of checksum – uses modular arithmetic to compute, store, compare values of fixed-length words
  - Another errordetection method common in networking in cyclic redundancy check (CRC) which uses hash function to detect multiple-bit errors

- Error-correction code (ECC) not only detects, but can correct some errors
  - Soft errors correctable, hard errors detected but not corrected

- Journaling and Recovery
  - FS need to recover from errors
  - One alternative is using journaling, which keeps a log of the changes
  - NTFS uses journaling, saving first the actions and the running them asynchronously
  - Asynch journaling can cause problems if the power is lost

- Volumes
  - Each disk can contain files and directories in a volume
  - A volume contains the same file system and can be divided into different partitions
  - A partition is a logical space used by a file system

- Mounting partitions
  - A partition needs to be loaded into the OS in order to be accessible to the user
  - Mounting links a partition to a directory, allowing it to be traversed
  - Partitions can be mounted at boot time, or at running time

# Mass Storage

- RAID
  - Redundant Array of Inexpensive Disks
  - Increases mean time to failure
  - Composed of 2 elements:
    - Mirroring: Keeping a copy of files in different HDD
    - Stripping: Separating the bits or blocks from each byte of a file into different HDD
  - Helps to increase efficiency and provide redundancy

- Network-Attached Storage
  - NAS is storage made available over a network rather than over a local connection (such as a bus)
    - Remotely attaching to file systems
  - Implemented via remote procedure calls (RPCs) between host and storage over typically TCP or UDP on IP network
  - iSCSI protocol uses IP network to carry the SCSI protocol
    - Remotely attaching to devices (blocks)

- Storage Area Network
  - Common in large storage environments
  - Multiple hosts attached to multiple storage arrays-flexible