# Module 1: Processes

* Program: instructions stored on disk – an executable file
* Process: a running program
  + Program counter: which instruction to execute next
  + Registers: what the immediate data on which instructions operate is
  + Address Space: where the instructions and data stored in memory are
  + Stack and frame pointer: how to find parameters of current function
* Address space: where the program data resides
* CPU registers: what the immediate data on which instructions operate is
* PCB (Process control block): structure the OS uses to keep track of the process information
* Scheduler: part of the OS, its jobs is to manage the process’ states
  + Only one process running at a time (single core assumption)
  + Other processes that are ready to execute go in a ready queue
* Ready (state): ready queue is the set of all processes residing in main memory, ready and waiting to execute
* Blocked (state): when a process is waiting for I/O it is placed in the blocked state. Next process in ready queue is set to running
* Context Switch: the exchange of register information that occurs when one process is removed from the CPU and another takes its place
* Process tree
  + Representation of all running processes shown in a tree-like structure
  + Parent (process): every process must have a parent (by default, the parent is the process that created it – the child process)
  + Child (process): created by parent process
  + Initial (state): when a process is created, it starts here
  + Zombie (process): when a process terminates, it goes to a final (zombie) state
    - The final state is required because we can’t delete the PCB immediately, the parent may want information such as the exit code
  + Orphan (process): if a parent terminates before the child, the child becomes an orphan process and the root process becomes its parent
* Process System calls
  + fork(): create another process (child) that is a copy of the current process (parent)
    - pid\_t fork(void);
    - creates a new process by duplicating the calling process
    - child process has a copy of parents address space
    - on success:
      * both parent and child continue execution at the point of return from fork()
      * returns pid of child process to parent process; returns 0 to child process
    - on failure: child is not created; returns -1 to parent
  + exec(): change the program of the currently executing process
  + wait(): do nothing until a child process has terminated
    - pid\_t wait(int \*wstatus);
    - suspend execution of the parent until one of its children terminates
    - on success:
      * returns pid of the child process that terminated
      * wstatus is population with information about the way the child process terminated
    - if a process has terminated, but parent has not yet called wait(), the process becomes a zombie
    - if the parent terminated without calling wait(), the child process becomes an orphan and a system process becomes the parent
  + pipe(): connects two processes
    - int pip(int p[2]);
    - creates communication channel
    - typical usage is right before calling fork, each process must close the ends of the pipe it is not using
  + dup(): duplicate file descriptor
    - int dup(int fd);
    - returns new file descriptor that is the lowest numbered available descriptor
    - new file descriptor refers to the same source as fd did previously
* Pipes: unidirectional data channel that can be used to communicate from one process to another
  + Sender puts data to one end (the write-end of the pipe)
  + Receiver gets data from the other end (the read-end of the pipe)
* Time-sharing: systems in which the scheduler may preempt a process, swapping it with a different process before it has completed its job
* Multiprogramming: letting the user run multiple programs (processes) together. When one process needs to wait for I/O another can be scheduled on the CPU
* Multitasking: each process gets a time-slice, a time limit before the next process gets to execute on the CPU
* User mode (CPU): mode in which “privileged instructions” aren’t allowed and memory boundaries are enforced
  + Can’t read/write outside of address space bounds
  + Can’t read/write I/O devices
  + Normal process only execute in this mode
* Kernel mode (CPU): all instructions are allowed. The OS executes in this mode
* Privileged instruction: can only be executed in kernel mode
* System call: different from a normal function call. Transfers control to the OS
* Trap: a program initiates a system call causing this; a software generated interrupt

# Module 2: Scheduling

* Job (or CPU burst): time when process needs CPU time
* I/O burst: time when process needs to wait
* CPU Bound: execution of a task or program is highly dependent on the CPU
* I/O Bound: programs with a large number of I/O operations
* Preemption: ability of the OS to stop or pause a currently scheduled task in favor of a higher priority task
* T-arrival: time when job first enter ready state
* T-completion: time when job finishes
* T-firstrun: time when job starts its first run on the CPU
* T-turnaround: time to complete a job
* T-response: time to first execution on CPU
* Scheduling policies
  + FIFO (First In First Out)
    - Implementation: FIFO queue
    - Preemption: none
    - Easy to implement
    - Problem: performance depends on arrival order, large upfront CPU burst can hurt turnaround and response time
  + SJF (Shortest Job First)
    - Implementation: priority queue sorted by job length (shortest first)
    - Preemption: none
    - Best average turnaround and response time when all jobs arrive at same time
    - Problem: if short jobs arrive after starting a long job, high turnaround and response time
  + STCF (Shortest Time to Completion First)
    - Implementation: priority queue sorted by time to completion (shortest first)
    - Preemption: if new job arrives with a shorter time to completion, it preempts
    - Short jobs don’t need to wait for a long job to complete
    - Problem: preemption gives STCF better response time in some cases, but there are still cases where response time can be poor
  + RR (Round Robin)
    - Implementation: FIFO queue
    - Preemption: job on CPU gets time slice, preempt when time expired
    - Low response time
    - Problem: RR has great response time with guaranteed upper bound, but bad average turnaround time, frequent context switches reduce CPU efficiency
  + Lottery
    - Each job assigned number of “tickets”
    - Every time slice, scheduler randomly picks a winning ticket and the job runs for that time slice
    - Over time, jobs run time is proportional to percent of tickets held
    - More tickets = higher priority, but low ticket jobs cant be completely starved
    - assign # of tickets inversely proportional to each job's runtime
    - no job can be starved, everyone gets their fair share of runtime
    - problem: short job can be unlucky and have higher response time
  + Stride
    - Deterministic (not random) but has same fairness as lottery; for each job, set its stride as a large number divided by its number of tickets
    - After every time a job runs increment its pass counter by its stride
    - Always picks the job with the lowest pass counter to run next
  + MLFQ (Multilevel Feedback Queue)
    - Rule 1: If Priority(A) > Priority(B), A runs (B doesn't)
    - Rule 2: If Priority(A) = Priority(B), A and B run in RR
    - Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue)
    - Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e. it moves down one queue)
    - Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level.
    - Rule 5: After some time period S, move all the jobs in the system to the topmost queue (priority boost prevents starvation of the long running jobs)
  + CFS (Completely Fair Scheduler)
    - gives a proportion of CPU time to each process
    - every process is assigned a time slice within one sched\_latency, if equal priority, each process gets a 1/N slice of sched\_latency, therefore, every process should run at least once in sched\_latency
    - run the process with the smallest virtual runtime (vruntime) next
* Time-slice (quanta in RR): unit of time
* Oracle (requirement to see into the future)
  + SJF and STCF require an oracle, because doesn’t know how long it will take for a job to complete
* Starvation: long job face this if we turn up the rate of I/O bound processes; it no longer can make progress
* Priority boost (MLFQ): after some time period S, move all the jobs in the system to the topmost queue
* Virtual runtime (CFS): a weighted version of the real runtime of each process
  + vruntime = vruntime + (weight ratio based on nice value) \* runtime
  + if nice value = 0, weight ratio = 1 and vruntime = actual run time
  + if nice value < 0, weight ration < 1 and vruntime < actual run time
  + if nice value > 0, weight ratio > 1 and vruntime > actual run time
* Nice value (CFS)
  + value set by user to indicate "priority," weights the 1/N time slice a progress gets before preemption
  + time-slice = weight / (sum of all processes weights) \* sched\_latency
  + low nice value = larger time slice
  + high nice value = smaller time slice
  + time\_slice of all processes still add up to sched\_latency
  + when current running process uses its time\_slice, or leaves for some other reason (e.g. blcoked), scheduler picks process with lowest vruntime to run next
  + when blocked process goes to ready queue, its vrutime is set to smallest vruntime of currenly running or runnable processes
  + scheduler gets called often, efficiently matters so CFS uses red-black tree (self-balancing binary search tree) to quickly find process with lowest vruntime

# Module 3: Memory Virtualization Abstraction

* Address space (virtual): a process’ view of memory is called its address space
  + Process have two forms of dynamic memory: Heap and the call stack
* Physical memory: form of very fast, but volatile data storage (random access memory (RAM))
* Heap: storage areas containing user-controlled dynamically allocated variables or data
* Stack: critical data structure used by processors in OS to manage program execution, interrupts, and exceptions
* Memory virtualization: want to give users an easy view of memory
  + Memory can become fragmented
  + Unused address space doesn’t need to be mapped to physical memory
  + Not every process will fit in memory, use disk for extra storage

# Module 4: Memory Virtualization Mechanisms

* Virtual address: memory address that is generated by the OS and presented to a program as if it were the actual physical address in the computer’s main memory (RAM)
* Physical address: the actual address in main memory where data is stored
* Address translation: translate process address into physical address
* Base and Bound: method requires two CPU registers
  + Base points to start of process in physical memory
  + Bounds points to a maximum legal address for process
* MMU (Memory Management Unit): hardware responsible
  + Typically part of the CPU but sits between the core and the address buss
  + Translates all addresses between CPU and main memory
* Segmentation: means we can locate parts of the address space independently in physical memory
  + need hardware requirements - registers for the start and size of each segment
  + a set of registers can indicate if a segment grows up or down
  + segments are in contiguous regions of physical memory (borders touching)
  + to allocate new segment, OS must keep a list of free memory - simple solution is a linked list of free regions of memory
  + on new allocation, search for first open spot that has sufficient memory (first fit strategy)
  + best fit strategy searches for smallest region of free memory that will fit the segment
  + makes code, stack, and heap independently relocatable
* Segment (address space): can locate parts of address space independently in physical memory
* Segmentation fault: fault, or failure condition raised by hardware with memory protection, notifying an OS the software has attempted to access a restricted area of memory
* Sparse address space: address space doesn’t need to be modified dynamically
* External fragmentation: wasted physical memory; not big enough to fit a full segment, so it can't be used
* Internal fragmentation: unused portion of page
* Compaction: used to reclaim the fragments
* Best fit (free-space management policy): strategy searches for smallest region of free memory that will fit the segment
* Paging (memory): address space divided into equal sized pages that can be stored in frames
* Frame: main memory divided into equal parts and each part is a frame
* VPN (virtual page number): index of the table
* PFN (physical frame number): points to the frame in physical memory
* Offset (from start of page): the number of address locations added to a base address in order to get to a specific absolute address
* Valid bit (page table): indicates if table entry is valid (not all of address space needs to be mapped)

# Module 5: Memory Virtualization Enhancements – TLB and Swap

* TLB (Translation-Lookaside Buffer)
  + store a cache of "popular" page table entries with the MMU hardware
  + cache can be accessed in single CPU cycle (main memory > 100 cycles!)
  + on each memory access -> look for VPN in TLB cache, if found (TLB hit) return entry, else (TLB miss) get entry from page table in main memory, store entry to cache (possibly replacing some other entry)
* TLB cache entry
  + contains basic fields such as VPN, PFN, and a bit that indicates whether entry is valid
  + note: valid bit in cache is not the same as valid bit in page table
* Spatial locality: address of memory access is likely to be close to the previous access
* Temporal locality: addresses likely to repeat in time
* Swap space: The allocated space where the virtual memory is stored on the hard drive when the amount of physical memory space is used up or full
  + Less used pages are moved here; divided into blocks that can hold one page
* Present bit (swap): page table has present bit to indicate if page is in physical memory (1) or in swap(0)
* Cache hit: page is in main memory
* Cache miss: page fault (page is in swap space)
* Page fault: when program tries to access memory that is on a page currently in swap
* Page out: need to decide on page to evict
* Page in: a page can be brought back to memory
* Proactive free space management: two thresholds high watermark and low watermark
  + When free memory drops below low watermark OS start background task to start pushing pages to swap space
  + Task stop when free memory passes high watermark
* Reference string
  + specific memory access patterns
  + generated artificially or by tracing a given system and recording the address of each memory reference
  + every time there is a page fault the page must be brought to main memory
  + consecutive repeated page accesses can never cause additional misses, for this reason they are not interested in them when evaluating policy performance and we remove them from the reference string
* Replacement Policies
  + OPT (optimal): assume we can predict the future and always chose to evict the page that will be used furthest out (or any page that will never be used again)
  + FIFO (first in first out): evict the page that has been in memory the longest
  + Random: replace an entry at random
  + LRU (least-recently used): replace the entry that was used the longest ago
  + Clock Algorithm: way to approximate LRU cheaply
* Belady’s Anomaly
  + larger cache does worse
  + increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern
  + occurs primarily in FIFO replacement algorithm which prioritizes pages that have been in memory the longest, page faults may increase by increasing the number of frames
* Dirty bit (swap)
  + if a page has only been read from (never written to) there is no reason to write it back to the swap space when it is evicted
  + add a dirty bit to the page table, initialize to 0, on every write to the page table the MMU (hardware) sets the bit to 1
  + can also make clock algorithm more efficient - first try to find a page with dirty bit and use bit both 0, because it is lower cost to replace
* Thrashing: when process is spending more time paging than executing
  + Starts at one processes and snowballs into several processes thrashing
  + Multiprogramming and multitasking make it worse, not better
  + Sudden and extreme drop in system performance

# Module 8: Deadlock and IO

* Dining Philosophers Problem
  + N philosophers alternate between thinking and eating. There are 5 chopsticks placed between the philosopher
  + To eat, a philosopher must pick up two chopsticks (to their right and left), to think they put both chopsticks down.
  + Problem is that a philosopher must wait until chopsticks is put back to use it
* Deadlock: situation involving a set of processes in which each process waits for an event that can be caused only by another process in the set (i.e. resource deadlock, synchronization deadlock)
* Conditions for deadlock (hold simultaneously): *need to stop one of these to prevent deadlock*
  + Mutual exclusion: only one process at a time can use a resource
  + Hold and wait: a process holding at least one resource is waiting to acquire resource held by another process
  + No preemption: a resource can be released only by the process holding it after the process has completed its task
  + Circular wait: there exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain
    - E.g, there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, …, Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.
* Deadlock detection using resource allocation graph
  + If graph contains no cycles, then no deadlock
  + If graph contains a cycle, then a deadlock may exist
    - If only one instance per resource type, then deadlock
    - If the cycle involves a set of resource types, and each resource type has only a single instance, then deadlock
    - If several instances per resource type, then possibility of deadlock
* Methods for handling deadlocks
  + Removing mutual exclusion
    - If we treat data as non-mutable (read only) there is no need for mutual exclusion
    - if resources are only created, never modified, no need for mutual exclusion
  + removing hold and wait
    - when possible, it’s a good idea not to hold multiple locks at the same time (i.e., don’t hold a lock and wait for another)
    - more efficient to have multiple resources locked at the same time
    - perfectly find to hold a lock as long as the thread doesn’t wait for another one while still holding the lock
  + adding preemption
    - if a thread can have its resource preempted, then deadlock would be broken
    - if not careful, preemption can lead to race condition, one strategy is to restart the thread or process
    - alternative solution: trylock – try to get the lock but don’t block (wait) if it not available
  + preventing circular wait
    - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

# Module 9/10: HDD and File System API/ File Systems

* I/O Computer Architecture
  + CPU: primary processing unit in a computer system. It executes instructions fetched from memory and coordinates the operation of other components.
  + Memory: stores data and instructions that the CPU needs during execution.
  + Bus(es): communication channels that transfer data between different components of the computer.
  + Peripheral/Device: external hardware connected to the computer
  + Device Controller: component that manages the operation of a specific type of peripheral or device
  + Device Driver: software that acts as an interface between the operating system and a specific hardware device
  + Interrupt: signal that the hardware or software sends to the CPU to gain its attention
  + DMA (Direct Memory Access): feature that allows peripherals to transfer data directly to and from memory without involving the CPU
* Hard Disk Drive (HDD)
  + Sectors: address space of the drive is divided into 512 byte blocks
  + Atomic Write: writing to a block is atomic
    - Either write to block fully succeeds or it doesn’t change anything
    - No partial write to block even If power goes out in the middle
  + Torn write: multi-sector reads and writes are supported but no guarantee of atomicity
  + Platter: circular hard surface were data is stored using magnetic persistence
  + Surface: a platter has a top and bottom surface, both can store data
  + Track: single circle around the center, a surface consists of thousands of tracks
  + Spindle: center of the platter which connects to a motor
  + Disk arm: moves the head across the platter to select a specific track
  + Disk head: transforms the magnetic field into electrical current
  + Rotational delay: time for sector to rotate under disk head
  + Seek time: time for disk arm to change position to the correct track
  + Random access: software requests addresses in any order it choses
  + Sequential access: software requests addresses in increasing
  + SSTF (Shortest seek time first): picks requests on the nearest track to complete first
* File: array of bytes given an identifier (inode number in Unix)
* Directory: list of (user-readable name, inode number) pairs that is also given an inode
* Root directory: head of the directory tree, often referred to as /
* Absolute path: a path that begins at the root and includes all successive subdirectories
* File descriptor: an identifier the process uses to reference a resource when making system calls (read(), write(), etc.). Think of file descriptor as pointer to an object that stores information about an open file
* System calls
  + open(): open or create a file
  + read(): read data from a file into a buffer
  + write(): write data from a buffer to a file
  + close(): close file descriptor after finishing the file operations
  + lseek(): change the file offset (position in the file) associated with a file descriptor
* hard link: connects a filename to an inode
* soft link: gives an alias to a name
* metatdata region (simple file system)
* data region (simple file system)
* contiguous allocation
  + file system has similar choice of contiguous allocation or block allocation (like paging)
  + contiguous allocation has problem of external fragmentation
* block allocation
  + block allocation has problem of internal fragmentation, block allocation usually wins because internal fragmentation is less of a problem and block allocation is much more efficient in allocating and resizing files
* inode: contains metadata information about a particular file
* data bitmap: stores which blocks are free/used in the data region
* inode bitmap: stores which blocks are free/used in the metadata region
* direct indexing:  the inode for a file has pointers to the data blocks of the file
* indirect indexing (single, double and triple)
  + an indirect pointer points to an inode with more pointers
  + double indirect means two levels of indirection
  + triple indirect means three levels of indirection
* multiple-level indexing: head inode may combine multiple levels of indirection
* caching (file system blocks): holds popular blocks to decrease number of times blocks are read from disk
* write buffering: batch multiple updates into a smaller set of I/O operations
* Berkeley Fast File System (FFS)
  + principle: keep related stuff together
  + two heuristics to improve performance - try to allocate data blocks for a file in the same block group as the files inode, try to locate files that are together in a directory in the same block group
  + FFS heuristics are based on another form of locality - path locality
* Block groups: consecutive portions of the disk's address space. Most hard drives don't provide enough information to choose cylinder groups, most file systems are organized by block groups
* Path locality: consecutive file accesses are likely to be to file paths that are near each other
* Fsck: Unix tool fsck (file system check) is used to check and fix file system on boot
* Journaling (write-ahead logging): improve the consistency and reliability of data in the event of system failures. (WAL) is a specific form of journaling where changes to the file system or database are first recorded in a journal or log before they are applied to the actual data structures
  + Metadata: information about data in a file system or database
  + Transaction: sequence of one or more operations (read or write) that are executed as a single unit
  + Commit: operation that indicates the successful completion of a set of changes
  + Checkpoint: point in time at which the system ensures that all previously logged transactions have been applied to the data structures
* Flash-based SSDs
  + Cell, page, and block: a silicon transistor technology
    - a transistor forms a cell which stores a bit in a single-level cell or up to three bits in a triple-level cell
    - to write to a value the cell must first be erased, setting all bits to 1
    - only after a bit is set to 1 can it be set to 0
    - the need to erase before write is important to understand SSD performance characteristics
  + Read page: involves retrieving data stored in a specific page of the flash memory
  + Erase block: Before new data can be written to a block, the entire block must be erased
  + Program page: writes the page by setting 1's to 0's where needed, time is somewhere between a read and erase
  + Flash translation layer (FTL): a flash controller virtualizes the flash (i.e., mapping logical address to virtual address); on chip memory is required for caching and buffering of blocks and pages
  + Log-structured FTL: in-memory table is used to map virtual to physical pages. on every write the page is moved to a different physical location
  + Garbage collection: reclamation of dead locks. if the same logical page is written multiple times, the old versions of the page will remain in physical memory as garbage (unusable)

# Module 11: Crash Consistency and Virtual Machines

* Virtual Machines: applications may require different OS environments, therefore desire to have multiple OSes on single machine
* Virtual Machine Monitor (VMM) or hypervisor: Creates illusion of being hardware to a OS above it. Don’t want to modify the OS to run on VMM, transparency is major goal of VMM
* Motivations for VMM
  + consolidate multiple OSes onto fewer hardware platforms, lowering cost and ease administration
  + accesses to applications that only run on a different platform, e.g. run Linux application on Windows host
  + isolated and reproducible environment for testing and debugging
* Limited Direct Execution (VMM)
  + OS achieves virtualization through limited direct execution (context switching between processes)
  + VMM also operates on limited direct execution, need to context switch (or "machine switch") between virtual machines, to achieve "machine state", VMM must save entire machine state of one OS and restore state for state being switched into
  + a problem is that OS normally operates in kernel mode allowing it to execute all privileged instructions
* Trap Handler from OS to VMM
  + Standard system call: application in user mode executes privileged instruction, which causes trap, which switches processor to kernel mode
  + in standard configuration, it is safe for trap to execute in kernel mode because the OS installed the trap handlers at boot time
  + the difference on virtual machine is the VMM installed the trap handelers
* How does VMM know what to do with trap?
  + When the guest OS boots it tries to install trap handlers by writing them into specified memory locations
  + VMM know where in memory the guest OS trap handlers are located
  + when VMM trap handler executes it calls the guest OS trap handler
  + When guest OS trap handler returns from trap, it returns into the VM trap handler which performs the actual return-from-trap into the application
* Supervisor mode
  + before calling guest OS trap handler, VMM reduces the privilege mode (some processors have an additional supervisor mode)
  + when OS executes privileged instruction, a trap occurs giving VMM control
  + Guest OS executing return-from-trap is also a privileged instruction, which is how control
* VMM page table - “physical memory” vs machine memory
* Information gap between host OS and VMM
  + because of transparency, VMM doesn't know what guest OS is trying to achieve
  + there's an information gap between OS and VMM that can lead to significant inefficiency
  + for example, when OS has no useful work (i.e., no runnable processes) it will spin in its scheduler loop
  + Para-virtualization: potential solution breaks transparency, in para-virtualization guest OS has small modifications to operate more effectively in virtualized environment

# Module 12: Security

* Confidentiality: information maintained by a computer system is accessible only by authorized parties
* Integrity: computer system’s resources can be modified only by authorized parties
* Availability: computer system be accessible at required times by authorized parties
* Authenticity: computer system can verify the identity of a user
* Intruder: those who attempt to breach security
* Vulnerability: weakness in the security of the system (buffer that is not protected from overflow)
* Threat: anything that leads to loss or corruption of data or physical damage to the hardware/infrastructure
* Attack: attempt to breach security
* Malware: software designed to exploit, disable, or damage computer systems
* Trojan horse: program that looks legitimate but can take control of computer
* Spyware: program frequently installed with legitimate software to display ads, capture user data
* Ransomware: locks up data via encryption, demanding payment to unlock it
* Sniffing: technique for intercepting computer communications
* Masquerading: pretending to be another person online
* Man-in-the-middle: A security attack in which network communication is intercepted in an attempt to obtain key data
* Denial of service (DoS): overload the targeted computer preventing it from doing any useful work; Distributed Denial-of-Service (DDoS) comes from multiple sites at once
* Access Control: regulating what actions subjects can perform on general objects
  + Object (access control): resources on which an action can be performed: files, tables, programs, memory objects, hardware devices, strings, data files, network connection processors
  + Subject (access control): human users or agents that represent users from which objects are protected
  + Access mode: controllable actions of subjects on objects
  + Access Control Matrix: describes the access modes granted to subjects on objects
  + Access Control List: an alternative to the access control matrix
  + Capability
* Encryption: process of converting information or data into a code to prevent unauthorized access
* Decryption: process of transforming data that has been rendered unreadable through encryption back to its unencrypted form
* Plaintext: unencrypted message
* Ciphertext: encrypted message
* Symmetric (key system): Ek = Dk so the key must be kept secret
* Asymmetric (key system): Ek ≠ Dk, Ek, can be made public; Dk is secret and can’t easy be derived from Ek
* Public key: published key used to encrypt data
* Private key: key known only to individual user used to decrypt data
* Authentication: need method to identify users and verify they are who they say they are
* Nonce: pick once in a lifetime number for each session
* Defense in Depth (design principle): multiple layers of security controls protect the system
* Layered Architecture: follows the principle of defense in depth. Often relies on hardware support to protect inner layers
* Trust: strong motivation for defense in depth in OSes
* Trusted Computing Base (TCB): is the name given to everything in the trusted operating system that is necessary to enforce the security policy
* Trusted Platform Module (TPM): hardware that controls what can be done on the machine
* Virtual Machine: present to the users only the resources they need, giving the user the impression their program is running on its own machine
* Sandbox: similar to virtualization, a protected environment in which a program can run and not endanger anything else on the system
* Honeypot: a fake environment intended to lure an attacker

# Module 13: Networking

* Protocol: devices on the network need to speak the same language
* Link: physical connection between nodes (computers, devices, routers…)
* Packet: block of data being communicated
* Switch: nodes with multiple links that forward data/packets from one link to another
* Host: computers/devices connected to the network
* Router: forwards data between networks
* Client/server
  + Two types of communication channel: Request/Reply or Message Stream Channels
* Layers of OSI Model
  + Application
    - Standardize common type of exchanges
  + Transport
    - Implements a process to process channel
    - Unit of data exchanges in this layer is called a message
  + Network
    - Handles routing among nodes within a packet-switched network
    - Unit of data exchanged between nodes in this layer is called a packet
  + Data Link
    - Collects a stream of bits into a larger aggregate called a frame
    - Network adaptor along with device driver in OS implement the protocol
    - Frames actually delivered to hosts
  + Physical
    - Handles the transmission of raw bits over a communication link
* Socket: combination of IP address, transmission protocol, and port
  + The point where a local application process attaches to the network
  + An interface between an application and the network
  + An application creates the socket
* IP Address: identifies a device in the network
* Port: numbers identify individual process
* TCP (Transmission Control Protocol): standard that defines how to establish and maintain a network conversation by which applications can exchange data
* UDP (User Datagram Protocol): connectionless communication protocol for transporting packets across networks
* Client-Server Model with TCP
  + Bind
    - Binds the newly created socket to the specified address i.e. the network address of the local participant (the server)
    - Address is a data structure which combines IP and port
  + Listen: defines how many connections can be pending on the specified socket
  + Accept
    - Carries out the passive open
    - Blocking the operation
      * Doesn’t return until a remote participant has established a connection
      * When it does, it returns a new socket that corresponds to new established connection and the address argument contains the remote participant’s address
* connect (what a client does): doesn’t return until TCP has successfully established a connection at which application is free to begin sending data. Address contains remote machine’s address
* bandwidth
  + width of the frequency band
  + number of bits per second that can be transmitted over a communication link
* latency = propagation + transmit + queue
* propagation = distance/speed of light

# Module 14: Distributed Systems

* Transparency (design principle of distributed systems)
  + Requires that the presence of the network and the physical separation of components are hidden such that distributed application components operate as though they are all local to each other
* Stateful (protocol): the server maintains information about the client’s state (e.g. what files the client has open, location (lseek) of next read…)
* Stateless (protocol): server without client state is stateless
* NFS (Sun’s Network File System)
  + Defines an open standard client/server protocol for making a distributed file system
  + Server can be built on top of the most traditional file systems
  + Stateless protocol, server doesn’t remember anything from previous client requests
* File handle (NFS): has volume identifier, inode number and generation number
* NFS commands
  + LOOKUP: obtain file handle
  + READ: read from file at specified location a number of bytes
  + WRITE: write to file at specified location a number of bytes
  + GETATTR: get the attributes for a file (e.g., time of last modify)
* Generation Number: identifies the version of the inode. Unix filesystems often allow reusing inode numbers after a file has been deleted
* Retry on failure: solution is client retires the request after a timeout
* Idempotency: principle that performing an operation multiple times is equivalent to the effect of performing the operation a single time
* Cache consistency problem
  + When client can’t get most recent version of file from server it is an update visibility cache consistency problem
  + When client reads from out-of-date cache it is a stale cache consistency problem
* Flush-on-close (cache): semantics mean cache is always flushed when the application closes a file. Ensures that subsequent opens from another node will see the latest file version
* Attribute cache
  + Check if file has changed before using cached contents of file
  + The GETATTR command will indicate time of last modification to file
  + Results in a flood of GETATTR commands, solution is to add a local attribute cache that updates contents only after a timeout
* AFS: whole-file caching, different than NFS which caches blocks of data
  + designed with goal of scalability
  + client-side code is Venus and server-side is Vice, both sides take advantage of existing Unix file system to store files
  + performs whole-file caching on local disk
* Scalability: server should support as many clients as possible
* AFS commands (only need to know these three)
  + TestAuth: test whether a file has changed (used to validate cached entries)
  + Fetch: fetch the contents of file
  + Store: store this file on the server
* Whole-file caching
  + When application calls open() the entire contents are copied to the local disk
  + All read() and write() operations are performed only on the local copy of the file
  + On close() file is flushed back to server
* Volumes: attempt to solve load balancing problem – directories are mounted to volumes
* Callback: to solve problem of too many AuthTest messages, use callback to reduce number of interactions with server
* Heartbeat protocol: client sends periodic message and expects response