1) **Operating system architectures**
    1. Goals
        1. Control and coordinate system resources.
        2. Hardware abstraction layer (HAL)
            1. Disk access, memory management, task scheduling, user interfacing.
    2. Monolithic kernel
        1. Virtual interface to hardware
        2. System calls provided by modules
        3. Pros
            1. Very fast, if done correctly
            2. Dynamic extention on runtime by loading modules
        4. Cons
            1. Difficult to do correctly
            2. Bug in 1 module can crash the whole system
        5. Examples
            1. Classic UNIX (BSD)
            2. Linux
    3. Micro kernel
        1. Tiny kernel, with features split into system and user programs
        2. Provides minimal process/memory management, and communications (IPC).
        3. Pros
            1. Easy to extend
            2. More secure and reliable, due to fewer kernel processes
        4. Cons
            1. Slower, due to more system calls
        5. Examples
            1. QNX
            2. AIX
            3. Minix (Linux's basis)
    4. Virtual Machine
        1. Each VM functions as a whole computer
        2. Hardware is either passed through, or emulated.
        3. Sharing data through virtual disks and/or virtual network
        4. Pros
            1. Isolation dimisses needs for protection
            2. Good for researching/testing new operating systems
        5. Examples
            1. Mac OS 68k on Mac OS PPC
            2. x86 on PPC
            3. Very slowly PPC on x86 (registers)
            4. Transmeta CPUs
            5. Java VM, Parrot VM
    5. Exokernel
        1. Undergoing further research

2) Processes and threads

1. Processes
    1. Process State
        1. New, Running, Waiting, Ready, Suspended, Terminated
        2. Only #CPUs can be Running at any one time
    2. Process Control Block (PCB)
        1. Process state
        2. Program counter (next instruction)
        3. CPU registers
        4. CPU scheduling information (priority)
        5. Memory management information
        6. Accounting information (time used, other statistics)
        7. I/O status information (open files, etc)
    3. Parent-Child processes
        1. Child can use New or Parent resources (exec, fork)
        2. Parent can wait for children, or continue
        3. Parent can take down children with it, or allow them to continue
    4. Independent or Cooperating
        1. Data sharing
        2. Task splitting (SMP)
        3. Modularity
        4. Convenience
2. Threads
    1. Seperate stack, registers. Shared code, data, files.
    2. Pros
        1. Responsiveness (one load, one display, one blocked)
        2. Resource sharing
        3. Time economy (creation, context switching)
        4. Utilization of multiprocessor architectures (SMP)
    3. Types
        1. Kernel threads
        2. User threads
        3. Many-to-one (Solaris, GNU Pthreads)
            1. Pro: Efficient
            2. Con: One thread can block whole process
            3. Con: No SMP
        4. One-to-one (Linux, Win32)
            1. Pro: Concurrency (no blocking)
            2. Pro: SMP
            3. Con: Overhead of creating kernel threads
            4. Con: Overall thread limit
        5. Many-to-many (IRIX, Tru64 UNIX)
            1. Pro: Concurrency (no blocking)
            2. Pro: SMP
            3. No cons
            4. Two-level model (many-to-many + one-to-one)
3. Examples
    1. Apache HTTPd

3) Scheduling

1. CPU
    1. Non-preemptive (cooperative: Win16, MacOS 9-)
        1. Process has to release the CPU itself
        2. Pro: Simple, stable.
        3. Con: Bad multitasking
    2. Preemptive (Win32, MacOS X)
        1. CPU is passed around via a timer
        2. Pro: Concurrency
        3. Con: Complex, potentially unstable.
    3. First-Come, First-Serve (FCFS, FIFO)
        1. Non-preemptive
        2. Con: Average wait time not minimal
        3. Con: Average wait time varies a lot
    4. Shortest-Job-First Scheduling (SJF)
        1. Provably optimal
        2. Long-term: Program has to say how long it will need
        3. Short-term: Exponential average (averaged over time)
        4. Non-preemptive mode: Will allow current process to finish time-slice
        5. Preemptive mode: Will break off current process in favour of smaller one
    5. Priority scheduling (the above SJF is a PS)
        1. Non-preempt mode: Put in front of ready queue
        2. Preempt mode: Cuts current process off in favour of high priority ones.
        3. Con: Indefinite blocking/starvation
            1. Solution: Age increases priority
    6. Round-Robin scheduling
        1. FIFO queue where each entry gets a time-slice to do its work.
    7. Multilevel Queue scheduling
        1. Foreground vs Background queues
        2. Inter-queue scheduling
    8. Multilevel Feedback-Queue scheduling
        1. Processes can be moved between queues based on various feedback
    9. Multi-Processor scheduling (asymmetric master/slave, symmetric individuals)
    10. Real-time (hard RT with special hardware, soft RT on normal hardware)
    11. Quality measurement: Maximize CPU utilization (but max 1 sec response), max throughput so average turnaround time is proportional to execution time.
2. Disk
    1. First-Come, First-Serve (FCFS, FIFO): Fair, but possibly slow.
    2. Shortest seek time first (SSTF): Like SJF, so can cause starvation/long waits.
    3. SCAN: Goes back and forth, servicing requests along the path.
    4. C-SCAN: Goes forth like SCAN, but does nothing on the way back.
    5. LOOK: Moves to the last request in a given direction, then reverses.
    6. Quality measurement: Difficult, since rotational latency cannot be accounted for.
3. Page-replacement (Belady's anomaly: more frames != less faults)
    1. FIFO, shuffles out the frame at the head of the queue, suffers from BA
    2. Optimal, which requires knowledge of the future (used for comparison; similar to SJF; no BA)
    3. Least Recently Used (LRU), shuffles out the page used longest time ago, no BA, stack algorithm (n is subset of n+1).
    4. Additional Reference Bit, which shifts the reference bit onto a byte every Xms (e.g. 100ms). Lowest number = LRU page.
    5. Second-Chance, like FIFO but skips and clear a page if its reference bit is set.
    6. Enchanced Second-Chance, looks at both reference and modify bit; if modified it won't be victim.
    7. Least frequently used/most frequently used; Shuffles out the page with the lowest/higest usage count.
    8. Quality measurement: Compare to optimal.

4) Synchronization and communication

1. Synchronization (we wish to avoid race conditions)
    1. Critical Sections
        1. Mutual-exclusion (only 1 in a critical section at any time)
        2. Progress (who gets to enter its CS)
        3. Bounded waiting (prevent infitine wait/starvation)
        4. Two threads need 3 variables to correctly satisfy these requirements.
    2. Hardware synchronization: Get-and-Set instruction, Swap instruction
    3. Semaphores
        1. Two methods: acquire (wait, decrement), release (increment)
        2. Binary semaphore: Mutually exclusive (mutex) locks
        3. Counting semaphore: Access to a given number of a resource
        4. Spinlock (busy-wait) semaphores: Bad on single-CPU systems (but 2 CPU: 1 spin, 1 work).
        5. Introduce wakeup call, and a waiting process list.
        6. Single-CPU: Just stop interrupts. Multi-CPU: Use solution from CS (3 vars).
    4. Deadlock
        1. Definition: A set of processes are waiting for an event that can only be caused by a process in the set.
        2. Indefinite blocking/starvation: A process that never gets a resource due to using LIFO buffers.
        3. Reader-Writer problem: Many readers, single writer.
        4. Dining philosophers problem: 5 thinkers, round table, chopsticks.
        5. Solution is to code correctly. Weak solution is to forcibly kill or restart processes that are deadlocked.
        6. Livelock: Like deadlock, only where a process fails at something repeadetly.
2. Communication
    1. IPC
        1. Useful in distributed computing (clusters, internet)
        2. Message-Passing System
            1. Direct communication
                1. Must know the process to send to/receive from
                2. Con: Hard-coding names is bad.
            2. Indirect communication
                1. Uses a mailbox to send/receive messages.
                2. Owner can receive, users can send.
                3. Process-owned mailbox: No confusion about ownership, but must notify when exiting.
                4. OS-owned mailbox: Creator is owner, and can assign new receivers (owners).
            3. Synchronous (blocking) communication & Asynchronous (non-blocking) communication
                1. Combination of non-blocking send/receive, and blocking send/receive. If both block, it's a rendezvous.
            4. Automatic buffering & Explicit buffering
                1. Zero capacity: Exactly 1 message at the time (very likely causes blocking)
                2. Bounded capacity: At most N messages (may cause blocking)
                3. Unbounded capacity: Unlimited messages (sender never blocks)
    2. Shared Memory
        1. Multiple processes share a single address space

5) Memory management

1. Address Binding (relative to absolute addresses)
    1. Compile time: Relative address is known at compile time (very unlikely today) = Absolute code
    2. Load time: Relative address is known at load time, and doesn't change = Relocatable code.
    3. Execution time: Relative address change between execution slices = Current prevailing model
    4. Logical address (virtual address) map to physical addresses via a Memory Management Unit
2. Dynamic loading
    1. DLLs, shared libraries. Update shared code without recompile.
3. Memory Allocation
    1. First-fit: First hole large enough to accomodate the program. Fastest.
    2. Best-fit: Smallest hole. Equally good as first-fit.
    3. Worst-fit: Largest hole. Worst.
    4. All lead to external fragmentation (small chunks of unallocated space)
4. Virtual memory (swap)
    1.
5. Page based
    1. Page number P + Page offset D. P goes into the page table, and outputs an address F, so that F+D is the physical address to be accessed. This happens behind the scenes (coder sees 1 address).
    2. Leads to internal fragmentation (unused allocated space) of average ½ page per process.
    3. Process only sees its memory as a continuous block, but OS sees it scattered all over.
    4. Hardware support:
        1. Dedicated page register: Very fast, but cannot contain many pages.
        2. Page-table base register (PTBR): Page table in main memory, which is slow (by x2).
        3. Translation look-aside buffer (TLB): Fast parallel search, expensive hardware. Some entries can be wired down (permanent), such as kernel pages.
            1. Address-space identifiers (ASIDs): Keys the entry to the process. If no ASID, TLB must be flushed before use to prevent corruption.
    5. Structure
        1. Hierarchical Paging: Splitting P+D into P1+P2+P3+D or more. Bad for 64bit.
        2. Hashed Page Table: Using a hash table to find where to search.
        3. Inverted Page Table: PID+P+D. Smaller to store, but slower (TLB helps).
    6. Shared pages – processes can share common pure code.
6. Segment based
    1. Supports the users' view of the code. Addresses are passed as <segment-id, offset>. Coders usually don't have to bother with it, though, since compiler handles the segmentation.
    2. Hardware: Segment table with a segment base and a segment limit. Segment S + Offset D, where D must be < limit.
    3. Protection & Sharing
        1. Segments can be defined read-only or execute-only.
        2. Processes can share segments of code, but self-references are an issue (use: relative offsets).
    4. Fragmentation
        1. Segments are variable length, so have to use first/best-fit
        2. External fragmentation. Solved by paging...
7. Demand Paging
    1. Pager, that only reads in pages we need.
    2. Valid/invalid bit: Valid = Legal & loaded; invalid = illegal or not loaded
    3.
8. Example:
    1. x86 uses segments that are paged.

6) File systems

1. Bad Block management: Keep spare sectors available, and dynamically remap. Data lost.
2. Files
    1. Allocation
        1. Contiguous allocation: Causes external fragmentation, but file access is non-fragmented.
        2. Linked allocation: Blocks are allocated in a linked list. Causes major file-fragmentation, but uses the disk space optimally. Can be made safer/faster with a cached FAT.
        3. Indexed allocation: A few blocks contain the index of data blocks.
    2. Free-space management
        1. Bit vector
        2. Linked list
        3. Grouping: Indexed allocation, but for free space.
        4. Counting: A disk address, and a count of free blocks at that address.
    3. Efficiency
        1. META-data (access time, creation time, etc)
        2. File structures, types
        3. Access control (chmod, chown)
    4. Performance: Cache
        1. Encryption, compression
        2. Recovery (Journaling)
3. Directory structure
    1. Linear list: Simple, but slow. B-tree helps a lot.
    2. Hash table: Much faster; avoids hash collisions with linked lists.
    3. Single-Level directory: Everything in /
    4. Two-Level directory: chroot to user's own single-level directory.
    5. Tree-structure directory: Absolute paths, relative paths.
        1. Whether a directory can also function as a fil (usually no)
    6. Acyclic-graph structure: Allows for links
    7. General-graph structure: Allows for cyclic links
4. File references (symbolic links, hard links)
    1. Links are not resolved when doing backups
5. Virtual File Systems (VFS)
    1. Provides an abstraction layer to the various types of local file systems. Can traverse networks.

7) IO and disk management

1. Device Driver
    1. Provides an abstraction layer with a stable API to a piece of hardware, whose API may vary.
    2. Can be loaded and unloaded dynamically based on demand (USB/FireWire devices)
    3. Accessing a device
        1. Polling/busy-waiting: Inefficient.
        2. Interrupts
            1. Non-maskable: Can happen at any time
            2. Maskable: Can be turned off during critical sections.
        3. Direct Memory Access (DMA): For large read/write, let the device bypass the CPU.
    4. Appliction I/O Interface (abstraction layer)
        1. Character stream or blocks: Byte by byte, or blocks of bytes.
        2. Sequential or random access: Device-dependent, or seekable.
        3. Synchronous or asynchronous: Predictable, or unpredictable response time.
        4. Sharable or dedicated: Multi-process, or not.
        5. Speed of operation: b/s to GiB/s
        6. Read/Write/Both: Access modes
2. Kernel I/O sub-systems
    1. I/O scheduling: Requests are queued up, then re-ordered to be executed in an efficient manner.
    2. Buffering
        1. Double-buffers (1 for download, 1 for writing to disk)
        2. Data transfer size disparity (network packet sizes)
        3. Clean semantics (copying files)
    3. Caching: A copy of existing data, for faster access
    4. Spooling: Creates the data on disk, and then sends it to the device one at the time (printer)
    5. Device reservation: Enables locking a device to only one process at the time.
    6. Error-handling: Retries failed operations, or makes sure a permanent error does not crash us.
3. Disk management
    1. Disk scheduling
    2. Disk formatting
    3. Partitioning (not really, there is a standard)
    4. Boot blocks
    5. Bad blocks
    6. Swap management (partition, or inline in file system)
    7. RAID (better in hardware)

8) Networking

1. Definition: A communication path between two or more systems.
2. Types:
    1. Client-Server
    2. Peer-to-Peer (access via a central coordinater, or through broadcast)
    3. Distributed (SETI, distributed.net, GRID)
    4. Clusters (redundancy, load balancing)
3. Protocols
    1. LAN: IPX, TCP/IP, UDP (also /IP)
    2. WAN: TCP/IP, UDP, Border Gateway Protocol (BGP, routing)
4. Communication
    1. Sockets
    2. Remote Procedure Call (RPC)
    3. Remote Method Invocation (RMI, Java)
5. Distributed file systems
    1. Transparency (usually through VFS)
    2. Issue: Consistency
    3. Issue: Stateless vs stateful
    4. Network Files System (NFS)
        1. Heterogeneous (can work on all machines, OSs, network archs)
        2. Mount Protocol
            1. User-mode server process
            2. Export list (root-only editing)
            3. Returns a file handle
            4. RPCs: Mount, unmount, view list.
        3. NFS Protocol
            1. RPCs: Search dir, read dir, alter dir/links, alter attributes, read/write files.
            2. Stateless design: No open/close calls. No locking.
            3. Path-name translation: One call for every path component (slow, but can be cached).
            4. Remote Operations
                1. Concept: Remote-service paradigm (meaning no buffer/cache)
                2. Practically: Buffering and caching are used for performance.
                3. Slow; no concurrency/synchronization between clients.
    5. Andrew File System (AFS)
        1. Scalable (5000+ workstations)
        2. Local name space (client, Virtue) vs. Shared name space (server, Vice)
            1. Client has small storage to cache what they are currently working on (temp, etc).
            2. Split into clusters of a few clients and a server, that in turn connect to the other clusters.
        3. Pros
            1. Client Mobility: Client can log in anywhere
            2. Security: Everything executed on client, so server is secured via authentication/encryption.
            3. Protection: Access lists enables fine-tuning permissions (both allow and deny lists).
            4. Heterogeneity: Some files are links to system-specific binaries.
        4. Shared name space
            1. Volumes (which are not whole disks or partitions)
            2. FID: Volume number, vnode number (inode on volume), uniquifier (multiple-access).
            3. Volume-location database. Volumes can be moved atomically.
            4. Only 1 master read-write of a volume, and several read-only slave volumes.
        5. Operations and Consistency
            1. Whole files are cached on clients, and are only updated on close, or on a callback from Vice. Venus manages this. Same goes for symbolic links. This greatly reduces RPCs since it's passive (vs active where it'd ask all the time).

9) Protection

1. OS
    1. Access control, Reliability/Stability
    2. Policy: What to enforce (may change over time; best if seperates from mechanisms)
    3. Mechanism: How to enforce policies (general mechanisms are better)'
    4. Domain of Protection
        1. Objects (hardware & software), have specific operations.
        2. Processes: need-to-know only current variables and rights (minimally)
        3. Domain Structure: Split out access rights to domains, which can overlap
            1. User domain
            2. Process domain
            3. Procedure domain
        4. UNIX: User domains, setuid
        5. MULTICS: Ring domains, doesn't satisfy need-to-know
2. Access matrices (chmod)
    1. Read, write, execute, copy*, owner
    2. Confinement problem (unsolvable, information wants to be free)
    3. Implementation
        1. Global table: Too large, bad management.
        2. Access lists (objects): Specific rights, and a default list.
        3. Capability lists (domains): List of objects and their capabilities with that domain. Secure.
        4. Lock-Key mechanism: Cross between Access and Capability. Objects has locks, domains have keys to those locks.
    4. Access Revocation
        1. Immediate vs Delayed: How to know when, if delayed?
        2. Selective vs General: Who it affects, all or some.
        3. Partial vs Total: What it revokes, all or some.
        4. Temporary vs Permanent
        5. Access lists are immediate, any, any, any.
        6. Capabilities need schemes:
            1. Reacquisition: Capabilites are deleted over time, but can be reacquired if not revoked.
            2. Back-pointers: Slow, but general and partial.
            3. Indirection: 2 tables, one with capabilites and one with objects.
            4. Keys:
3. Network issues

10)Security

1. Types of vulnerabilities
    1. Buffer-overflow (int vs unsigned)
2. Definitions
    1. Virus: A process that will inject its own code into other programs, thus altering them. A virus is by definition passive, meaning a user has to initiate the first infection (usually not on purpose).
    2. Worm: A process that will actively seek out exploitable vulnerabilities in a system, usually though the network. Then infect them, and use them for spreading itself.
    3. Trojan-horse (trojan): A virus or worm that asides from spreading itself also provides a backdoor to access the system remotely.
3. Prevention
    1. NX-bit