CHAPTER 1
Purposes: Interface, convenience, efficiency, information protection
Objectives: Small kernel, run fast, stable
Von Nuemann Arch: No real difference between data/instructions. Data has no inherent meaning. Memory is 1D array.
Interrupts: Software/hardware. Save processor state → load interrupt vector → execute ISR → return from interrupt → reolad saved state
Programmed I/O: Polling. Periodically checking state of device
Interrupt Driven I/O: Continue normal execution until event. ISR handles (usually sets a volatile flag)
Direct Memory Access (DMA): Direct access to DMA controller. Requires data to read/write, device address, starting address, and amount of data.
Multiprocessor Systems: Asymmetric (boss-worker relation) and symmetric (all peers)
Clustered Systems: Provide high /performance service over LAN. Redundancy/fault tolerance
Functions: Resource management, security, communication, environment, logging, UI
Timer: Prevents processes from getting stuck. Allows real-time control

CHAPTER 2
System Call: Provide interface to service by made available by OS.
Parameters: 1.) Registers 2.) Stored in block memory. Address in registers 3.) Stack
Policy: What will be done?
Mechanism: How to do it?
Kernel: Implementation of OS. Provides services to interface between user space and hardware space
Monolithic OS: Entire kernel in 1 file. Very fast, but difficult to implement. Insecure.
Layered Approach: Modular. Changes in one section don't affect others. Security. Simple to construct/debug.
Mikrokernel: Tiny kernel. Only contains essential services. Rest lies in user space. Very fast. Main function is to provide messaging interface between services (both user space and kernel space). Overhead in message passing.
Bare Metal: Virtualization technique where there is NO host OS, just a "virtualization layer" that lies between virtual operating systems and hardware
Hosted Architecture: Virtualization technique where there IS a host OS. Virtualization layer lies between host OS and guest OS
Boot sequence: Power on → system BIOS (bootstrapping) → POST (core test) → video BIOS → Cold/warm start (cold = dozens tests;→ warm = fewer) → CMOS Boot Order → Look for bootable disk in specified order → look for boot sector (MBR sector contains info about partitions, configuration, and boot loaders) → if necessary, launch second stage bootloader, display TUI, give control to OS

CHAPTER 3
Process states: new (creation) → running → waiting → ready → terminated
Daemon: Background, non interactive. Detached from keyboard & interfaces
Service: Responds to request from other programs over IPC (usually over a network)
Process: Program in execution
Control Block: Process state, Program counter → CPU registers → CPU-scheduling info (priority, pointers to queues, etc) -> Memory-management information (value of base, limit registers, page tables, etc) → Accounting info (amount of CPU and real time used, time limits, account numbers, pids) -> I/O status info (list of I/O devise allocated to the process, open files, etc)
Zombie: Completed process. Waiting for release/return
Orphan: Parents returned. Child still running.

Scheduling Queues: CPU scheduler (short term, allocates CPU time) → Job scheduler (long-term, ready queue) → Medium term (reduces multi-programming, store job on disk while waiting)
Context Switch: CPU switch from one process to another
Resolve 3 processes on 1 mailbox: Allow only 1 P to receive @ a time. Round robin receive.
Zero buffer capacity: Everyone must block until message received
Bounded buffer: Don't block while space left on buffer
Unbound buffer: Don't ever blocking
Ordinary pipes: Unidirectional → only accessible by parent process (accessible by child) → requires parent/child relationship → must be used on same machine → closes w/ process terminate
Named pipes: Bidirectional → no relationship necessary → continue to exist after process terminate → referred to as FIFOs
Connection Oriented Socket: TCP → Provides mechanisms for message verification
Connectionless Socket: UDP → just send data fast, don't care to check if it was received
Sockets: Send unstructured byte stream. Up to clients/servers to interpret w/ data structure
RPC (Remote Procedure Call): Provides well-structured message interface → Clients and servers make "stubs" → stub converts message to data structure (w/ External Data Representation (XDL) format) → allows for use between multiple OS
Steps for RPC: Client produces stub → stub build msg, calls local OS → client OS sends to remote OS → Remote OS passes to server stub → stub unpacks msg, calls server → server process, return to stub → stub sends to OS → server OS sends to client OS → client OS sends to client sub → client stub unpacks, returns to client

CHAPTER 4

Amdahl's Law: $speedup \leq 1/(S + \frac{1-S}{N})$, where S = serial portion % and N = # processing cores

Thread: Block: thread id, program counter, register set, stack. **Share codes section, data section, OS resources w/ other threads of same parent**
Concurrency: Execution is interleaved over time
Parallel Execution: Threads execute on different processors simultaneously
Programming Challenges: Identify tasks to be divided → balance work → split data → manage data dependencies (if exists, threads **must** be synchronous)→ testing/debugging
Data parallelism: Parallel processing of data (eg array)
Task parallelism: Parallel processing of task (eg user input & GUI)
1:1 model: 1 k thread for 1 u thread. Better concurrency. Limited # threads. Used on modern OS
Many:1: 1 k thread for many uthread. No concurrency. Blocking system calls blocks everything
Many:many: Many user thread to multiplexed kernel threads (<= # user threads). Difficult to implement b/c multiplexing
2 level: User gets to chose between 1:1 or many:many for specific application
Implicit threading: Transfer creation & management of threading from app developers to compilers and run-time libraries using thread pools, OpenMP, and Grand Central Dispatch
Thread pools: Create many threads at process start. When new thread needed, wake a thread from pool & pass request. When request complete, return thread to pool & tell it to sleep. If no threads available, wait.
OpenMP: Uses simple #pragma omp parallel <operation> before block of code to run in parallel. Supports C, C++, and Fortran. Creates as many threads as are in system
Grand Central Dispatch: Used in OS X and iOS.
Issues w/ multithreading Does fork call on all threads or just calling → signal handling (interrupts) → thread cancellation (asynchronous (by parent) or deferred (by thread)) →thread-local storage → scheduler activations
Scheduler Activations: Communication between user-thread library and kernel. Kernel creates lightweight process on virtual processors. Application can schedule threads on LWP. **Upcall** = kernel informs application about event → handled by thread library w/ upcall handler.