ection 2: Processes

ocess: Program instance, contains code & activity.
xit code: Return value from child to parent process.
ddress Space: Private set of mem. address per process.
tack: LIFO memory for local thread variables.
eap: Large memory area to allocate/deallocate vars.
ork: Fork() duplicates process. On success, child PID returned in parent, 0 in child. Failure, -1 in parent.
ait: Wait() waits until child: terminates or stops.
xec: exec() executes program in process.

ection 3: Threads

hread: Smallest unit of seq. instructions. Mult. threads in process can share same addr. space, each w/ own PC.
thread_join: Waits for specific thread to terminate.
thread_create: Starts child thread in same addr space.
Atomic Operation: atomic if executed w/o interruption.
ritical Section: Code accessing shared resource, cannot be concurrently accessed by more than one thread.
emaphore: sema_up increments. sema_down waits to become positive, then decrements.

ection 4: Scheduling & Synchronization

ock: Synchronization var. providing mutual exclusion. Only one thread holds @ once. Acquiring held lock will block.
Condition Variable: Provide serialization. CV defined by associated lock, condition, wait queue. Must hold lock to access CV functions; if condition false, thread blocks. Wait queue tracks, signals to one/all waiting when cond. true.
thread_yield: Thread put back in ready queue (might be first to)
Race Condition: State of execution causing multiple threads to access shared memory w/o mut. exc. Undefined.
Scheduler: Picks which thread to run next.
Priority Inversion: Low pri thread holds lock that hi pri need,
Priority Donation: Thread waiting on lock donates its priority to other thread recursively until thread has all locks and can run to completion.
Hoare/Mesa: Hoare = imm. transfers control. Mesa = while

Section 5 CV & Spin Locks

Condition Variable: Sync variable w/: lock (CV + lock = monitor) boolean condition, wait queue. CV_wait unlocks held lock, puts thread to sleep on wait queue, CV_signal removes one thread from wait queue into ready state. broadcast removes all threads from queue, puts into ready state. When waiting thread put back into ready state, it too reacquires lock. When thread runs code to make cond. TRUE, must acquire lock, modify condition, then call signal. Hoare semantics auto-wake thread when cond. true. Mesa simply puts on ready list, use while in case cond. changes.
Spin Lock: Busy wait on lock acquire until lock becomes free.

Section 6: Scheduling & Fairness

Scheduler: Decides which processes run when.

Calling Conventions: Interface of called code. Defines order atomic params allocated/passed, registers, stack, etc.
%esp, %ebp: %esp holds last thing on stack, gets moved as things get added/removed. %ebp holds beginning of cur. stack
Virtual memory: Process believes it has full address space, each process has virtual addr., maps to physical
Physical Memory: Actual memory address in hardware
Page: Contiguous fixed size chunk allocated to process
Page Table, Memory Management Unit: MMU is hardware that translates virtual address to physical. Page table stores mapping b/w virtual/physical, resides in MMU
Sbrk: Low-level (below malloc) C function which can increase/decrease memory allocated to program.

Section 8: Wait/Exit, Address Translation

Exit: Terminates process. OS reclaims resources (memory, files). Process becomes dead process.
Wait: Process waits on another (child) to finish execution
Zombie Process: Child process becomes zombie after termination if it still lives in system process table. If parent waits on child, will be reclaimed once exit value returns to parent. If not, "reaper" OS process locates zombies, retrieves exit, deallocates process.
Address Translation Structures: Segments are linearly addressed chunks of memory containing logically-related info i.e. program code, process heap/stack, etc. Of the form (s,i), where i must be within i of base segment's. Page table stores mapping between virtual -> physical. VA used by accessing process, PA used by the RAM.
Inverted Page Table: Uses PT that contains entry per physical frame, not logical page. Ensures IPT occupies fixed fraction of memory proportional to physical mem. not virtual addr. space, only one per system. Can be like hash map: stores tag & entry.
Translation Lookaside Buffer: TLB is cache that mem. management hardware uses to speed translation. Stores VA -> physical mappings so MMU can store recently used mappings instead of multiple lookups.

Section 9: IO & File Systems:

Second Chance Algorithm: Modified FIFO used to approx LRU. Each page has use bit; when evicting, if ref. set, clear & send back of list, else evict.
Clock Algorithm: More efficient second chance b/c pages don't need to go to back. Keeps hand pointing @ pages in circular order; hand moves round, clears ref bit, evicts
I/O: Input/output is processes by which OS gets/puts data to/from
Controller: OS performs I/O by communicating w/ device controllers which contains mem./registers for communication w/ CPU, & interface for communicating w/ hardware. Communication done via programmed I/O, transferring data through registers, or Direct Memory Access, which allows controller to write directly to memory.
Interrupt: I/O interrupts OS, interrupt handler runs. Useful for infrequent, sporadic events
Polling: OS checks regularly for IO. Less overhead, better for regular events such as mouse input.
Response Time: Time b/w IO request & completion.
Throughput: Rate operations performed over time.
Async IO: IO returns immediately, notifies later when done
inode: Filesystem object, contains attributes, pointers, etc.

Section 10: File Systems & Queuing Theory

Simple File System: Disk treated as big array; Table of Contents (TOC) at beginning followed by data. Files stored contiguously, but can have unused space between. In TOC, file descriptor entries w/ name, start location, and size. **Pros:** Simplicity, quick IO. **Cons:** External fragmentation, files not easily extended, no directory hierarchy, file type

External fragmentation: Free space b/w allocated regions, split so much not big enough for a new file. **Internal fragmentation:** Leftover space at end of blocks

File Allocation Table: FAT views disk as array. First block is boot sector, which contains bootup info. Super block w/ file system metadata right after. Followed by File Allocation Table, and then data in 4 KiB blocks. File viewed as linked list of data blocks, pointers stored in FAT so blocks are 100% data. 1:1 correspondence b/w FAT entries & data blocks, each FAT entry stores block index. Entries: N>0: No next block index. N=0, end of file. N=-1: Free block. Files can be stored non-contiguously. Maximum external frag = 4 Kbytes - 1 = 4095 bytes. Directory file stores entries. **Pros:** No external frag, can grow file size, has directory hierarchy. **Cons:** No pre-allocation, not contiguous (slow IO), assume FAT fits in RAM.

UNIX/Fast File System: Disk space divided into: Boot sector, Super block, **Free Block bitmap** (one bit per block, more efficient than table), Inode table, Data blocks. **Inodes:** Data structure w/ file metadata. Contains: ownership, size, mod. time, permissions, refcount, data block pointers; NOT name. 12 direct, 1 indirect, 1 Doubly, 1 Triply

New Technology File System: NTFS represents file as record in **Master File Table**. First record describes MFT itself, followed by MFT mirror.

Log-structured FS: Data written to circular log buffer. Throughput improved b/c avoids seeks, can batch writes. Writes create multiple chronologically advancing versions of file data. Simpler crash recovery from consistent state. **Unlink:** Syscall to delete file. Decrements ref count, removes entry from directory. Garbage collected @ 0 ref count.

Queuing Theory: μ (average service rate). $T_{ser} = S = \frac{1}{\mu}$, average service time. λ = average arrival rate (jobs/sec). $U = \rho = \frac{\lambda}{\mu} = \lambda S$ = utilization [0, 1]. $T_q = W$ = average queuing time. $T_{sys} = R = T_q + T_{ser} = W + S$ = response time. $L_q = Q$ = average length of queue. **Little's Law:** $L_q = Q = \lambda T_q$.

Section 11: FS Details & Reliability

Transaction: Indivisible set of ops that fail or succeed.
ACID: **Atomicity:** Transaction must occur in entirety or not at all. **Consistency:** Data must be in consistent state. **Isolation:** Transactions should not interfere. **Durability:** Should persist despite crashes.
Idempotent: Operation can be repeated w/o change.
Logging FS: FS where modifications done via transactions, recorded in append only log, not disk. Once committed, can be written to disk; on crash, can be recovered.
2PL: Two phase locking ensures atomic, isolated commitment of concurrent transactions. Each transaction must acquire shared/exclusive lock for read/write. **Growing phase:** can only acquire, followed by **Shrinking**, can only release. **Strict 2PL** has all released at once. 2PL ensures acyclic dependency graph, serialized transaction.

Section 12: Two Phase Commit (Byzantine: Need majority nodes)

2PC: Two phase commit coordinates transactions b/w master and slaves. State-changing transactions must be logged and tracked. TPC ensures write happens atom on all replicas or none. Master sends message to slaves, which return COMMIT or ABORT. If all commit, master sends GLOBAL-COMMIT and slaves ACK; else, Master sends GLOBAL ABORT

Security: Computing w/ adversary. Need reliability, robustness, fault tolerance. **Protection:** Mechanism for controlling access of programs, processes, resources - Page Table Mechanism, Round-Robin scheduling, data encryption. **Security:** Protection mech. to prevent use of resources. **Requirements:**
- **Authentication** (user is who claim to be).
- **Data Integrity:** Data not changed from src->dest.
- **Confidentiality:** Data only read by auth. user
- **Non-repudiation:** Sender/receiver can't claim didn't send/get data

Passwords: auth measure. Use **cryptography**. **Symmetric Keys:** Encrypt/decrypt w/ same key. Vulnerable. Can XOR, use **block cipher** to encrypt, decrypt. **Data Encryption Standard:** 56 bits. **Advanced Encryption Standard:** 128/192/256 bits. No proof of strength. **Secret Key:** A/B share key K, encrypt/decrypt message to verify. Vulnerable to man in the middle. **Crypto Hash:** H(x) public hash. Digest $d = HMAC(K, m) = H(K \mid H(K \mid m))$. Send d, m to receiver. Receiver uses K to recompute HMAC == d. MD-5, SHA-1 broken, SHA-2 in use. **Asymmetric encryption:** Use e to encrypt, d to decrypt. Knowing one still keeps other safe. **RSA:** Use receiver's public key to encrypt, decrypt w/ private. **Digital Certificate:** Binds entity w/ public/private key. Trusted authority distributes, anyone can extract public key. **HTTPS:** Uses **SSL** (secure socket layer), TSL (Transport Layer Security) to distribute digital certificates. Validates, RSA.

Networking: **Broadcast Medium:** Shared comm. medium. **Arbitration:** How to negotiate shared medium. **Aloha network:** sent blind broadcast w/ checksum, resend if garbled. **Carrier Sense:** Don't send unless idle. **Collision Detect:** Sender checks if packet trampled; if so, abort, wait, retry. **Adaptive Backoff scheme:** Choose wait time before try again. Adaptive and random, increases wait time after each failure.
Internet Protocol (IP) **IP addr** - 32 bit/host. Share through NAT. **Subnet:** Network of hosts sharing **prefix** (mask # bits). **Routing Tables:** (Addr range -> next hop). Set up dynamically. IP sends machine -> machine, add **ports**: process -> proc. UDP
For multiple packets: **Window based acknowledgements.** Send up to N packets w/ **sequence number**, receiver **ACKs**. **TCP:** Sender tracks: sent & ACK'd, sent/not ACK'd, to be sent. Receiver tracks: ACK'd, buffered, not received (or out of order). **Congestion Avoidance:** Slow start (increase send window on success, cut in half on fail). **Additive increase, multiplicative decrease**
3 way handshake: 1. Syn, Seq = x. 2. SYN ACK, seq num = y, ACK = x+1 3. ACK, ACK = y+1. Num = x+1. TO **FIN**: 1. FIN 2. FIN ACK

Sockets:

Client
socket()
↓
connect(host, port)
→ write req
→ read resp
↓
close()

Server
socket()
↓
bind(host:port)
↓
listen() ←
↓
accept()
read req
write resp → close client → close server

Microkernel: Most kernel services (FS, network) run in user-level. **Fault isolation**, **modularity** (apps update on own), **location transparency** (run on server)

Distributed File System: Transparent access to files on remote. **Name:** Hostname/localname. **Mounting:** Can mount remote by giving local name e.g. /users/sue/foo, foo:foo on server, (new) file has single global name.

CAP Theorem: Cannot have consistency (changes appear to everyone in serial order), availability (can get results at any time), Partition-Tolerance (system continues to work when network partitioned) @ same time → only two.

Remote Procedure Call (RPC): Abstraction for executing code on remote machines using stubgen & runtime lib. **Stub Generator:** Server sends out interface of function calls; Client programs have client stubs w/ all more functions, client simply calls w/ args. **Internally, client stub code:** - Generates message buffer (array of bytes). - Pack info into buffer (includes function identifier, args) **Marshalling:** packing. - Send to RPC server (handled by runtime lib) - Wait for reply, synchronous. - Unpack return value, unmarshalling. - Return from call, client stub back to client. **Internally, server code:** Unpack buffer - unmarshalling - Call into function. - Package (marshal) results, - send reply. Server handles concurrency by keeping thread pool. **Run-Time Library:** Handles performance & reliability issues. Handles naming by building on existing protocols: resolves IP address, hostname, port. RTL then directs. Doesn't send over TCP b/c ACKs slow, uses UDP w/ retries on failure. Uses binding to bind network name to readable name for client. **Issues** include non-atomic failure (machines do different things on failure) and performance (time + caching, memory).

Simple DFS: reads/writes forwarded to server using RPC. No caching client, only server. Server consistent view of FS to clients but slow performance, high traffic. Use caching to improve performance, but must maintain cache consistency (clients may have data not @ server). If server fails, clients can lose data / be inconsistent. Can fix w/ stateless protocol: server has no state, message contain all info. Might lose some data and client crashes.

Network File System: 3 layers: 1. Unix FS interface (open, read, write, close + file descriptors) 2. VFS layer (distinguishes local files from remote) 3. NFS service layer: bottom layer, implements NFS protocol: RPC for server calls. **Write through caching:** Data committed to disk as soon as modified (slow writes). NFS server stateless. **Idempotent:** Same request mult. times has same effect. **Failure Model:** transparent to client → can hang or error out. NFS enforces weak cache consistency: polls for changes (30s). Means values only updated after poll → read new 30s after write. **Pros of NFS:** Portable, simple. **Cons:** Inconsistent, doesn't scale.

Andrew File System: **Callbacks:** Server records who has copy of file. On change, server alerts them w/ old copy. Write through only on close() / updates to client only when file closed. Data cached on local disk of client + memory. Set callback on miss/write. Less server load than NFS, but still server bottleneck.

Virtual File System Switch: Virtual abstraction similar to local file system; Virtual superblock, inodes, file etc. Allows same API for diff filesystems. Four primary object types: superblock (mounted FS), inode (specific file), dentry (directory entry), file (open file). Microkernel (see other page)

Key-Value Store: Distributed Hash Table, can put, get, del keys/values. Partition set of key-values across multiple machines. **Challenges:** fault tolerance (handle machine failures w/o data/performance loss), consistency (in face of failures, message loss), scalability, heterogeneity (1ms to 1000ms latency, 32 kbps-100mbps bandwidth). **Directory-based structure:** Have a node maintain mapping b/w keys and machines that store them. **Recursive query:** has server w/ mapping contact node, get value, return. **Iterative query** has server return node to client, client contacts directly. Recursive is faster (server closer to nodes), and easier to serialize, but has server bottleneck. Iterative is more scalable, but slower, hard to enforce consistency. To improve fault tolerance, replicate values across multiple nodes. To improve scalability: increase storage (more nodes), improve concurrency (replication → multiple GETs), implement **Load Balancing** (balance usage amount across nodes). To improve consistency: make sure nodes replicate correctly. **Consistency models:** atomic (r/w to replicas appear as if there's a single replica; transactions), eventual (given enough time, all updates will prop through system). **Quorum Consensus:** Improve put/get performance → Define replica size N; put waits for ACK from @ least W replicas, get waits for @ least R; $W + R > N$. To scale up, use consistent hashing: give node ID ≈ hash(key), store KV pair in node w/ id right above.

MapReduce: Map: $(K_{in}, V_{in}) \to list(K_{inter}, V_{inter})$. Reduce: $(K_{inter}, list(V_{inter})) \to list(K_{out}, V_{out})$. Restricted KV model, deterministic, idempotent, same fine-grained op on bigdata. **Pros:** Distribution is transparent, automate fault-tolerance (rerun failed tasks), Scaling, Load-balancing. **Cons:** Restricted programming model, high latency, [illegible]

Four Requirements for Deadlock: (1) Mutual exclusion (Only one thread @ a time can use resource) (2) Hold and Wait (thread w/ resource waits for [illegible]) (3) No preemption (thread only voluntarily releases resources after finished), (4) Circular Wait (wait on each other)

Banker's Algorithm: Keep system in SAFE state, only grant resources if some thread has enough to proceed after allocation

Address Translation: **Base and Bound:** Thread given VA block, can only access if above base, below bound. **Cons:** Fragmentation, bad for sparse space, can't share. **Segmentation:** VA made of segment + offset, maps to table w/ base and bound, PA = base + offset, offset [illegible]. **Page Table:** VA made of VPN + offset, VPN maps to PPN in PT, PA = PPN + offset, PT has access per[illegible]. **Multi-Level:** VA = VPN1 + VPN2 + offset, VPN1 maps to second level PT, PN = PPN from PT2 using VPN2 + [illegible]. **Inverted:** VPN hashes to PPN + offset. **Calculations** for single level: offset bits: lg(pagesize), VPN = 32 - offset bits = PPN. PTE = PPN + meta. PT = (PTE bits) × (num pages), numpages = 2^{VPN}

Caching: Cache misses: Compulsory (first access), Capacity, conflict (mapped to same), Coherence (invalidated). **Direct Map:** Addr = Tag + Index + Byte, index → row, store tag, select w/ byte. **Set associative:** Index maps to rows, store tag in any of section. **Fully associative:** Tag + byte, store tag anywhere. TLB caches translation for PT's [illegible]

Demand Paging uses memory as cache for disk
mmap maps disk to memory, improves IO

MT1: disabling interrupts on CPU does nothing for other
Simultaneous multithreading = hyperthreading = switch
threads every cycle. For HTTP sockets, child & parent
must both close. Kernel mode happens @ well-
defined entry points. Thread blocked only on
one CV @ once. Pintos doesn't use floating
point b/c it doesn't save in registers.
CV, use queue of semaphores so need to
implement scheduling for CV, not semaphores.
Printf buffer, doesn't need kernel.
Disabling interrupts to create crit. section: (1)
user level code cannot (2) lock out H/W, important
events can be missed (3) Slow performance.
Only works when: interrupts would cause incorr. behav
Hoare hands control to waiting thread immediately
Mesa puts on queue, no guarantee on waking
Context switch: Threads: save registers, program
counter, condition registers, thread execution state
Processes: additional save PT pointer, seg. registers
User->Kernel: Syscall, exception, interrupt
Fork(): Create child w/ dup. addr. space.
Exec(): throws away addr. space, runs executable
Overuse of threads bad: (1) waste cycles on switch
(2) waste memory for stacks & TCB
Spinlocks only efficient when expected wait time
is less than time to switch threads

MT2: mmap maps VA to pages in buffer cache
which hold contents of file. FFS doesn't
include name. MMIO maps device memory
to physical addr. space, threads can access.
This means not all physical address in DRAM.
2PL guarantees conflict serializable b/c
conflicts turn into deadlock which are aborted.
Combo of polling & interrupts to handle events.
Network Addr. Translation allows more than 2^{32} comps.
IP router needs prefix & next hop to them.
Caches effective when working set fits in them.
Precise exception is when everything before exception
guaranteed to have happened; helps restart.
RAID5: XOR blocks to recover failed one
Copy on write: copy addr. space read only, make copy
of page on page write, change in PT.
Syscall # & syscall args passed from user to kernel
by pushing on stack in reverse.
Direct Memory Access: DMA controller handles
data transfer to memory [illegible]