Introduction to OS
OS: intermediary between user and hardware
- Time-sharing: illusion of concurrency, allocate resources, control program
- Kernel; monolithic vs. microkernel has more overhead but smaller
- Hypervisors: T1 (over hardware), T2 (over host OS)

Process Abstraction
Process: abstraction of running program
Stack Memory Region
- Stack frame (sf): for function invocation
- stack grows → address decreases
- Setup/ teardown managed by OS
- Local vars, Params, saved registers (GPRs temp in stored in mem, register spilling), saved SP, saved FP (fixed location in sf), return PC
Memory Context – Pg Table, TLB
OS Context – PID, process state
Hardware Context - GPR, PC, SP, FP
Process Table – Contains PCBs (3 contexts above), updated during context switch
- Text: Instructions, Data: Global var
- Ptrs in stack, but can point to heap
Interaction with OS - System Calls via TRAP, dispatcher finds system call handler
- Function wrapper (same)/ adapter.
- Exceptions (sync, due to program execution) and Interrupts (async, e.g. CTRL-C/ kill). Hardware interrupts, hw to interact with OS
UNIX Case Study
- fork(): creates duplicate process, returns PID of child (can exec() another process)
    . exit(), wait(&status), wait(NULL)
    . Copy on Write: dup when changed
- Zombie: Child terminates before wait, becomes zombie (PCB not cleared)
    . Parent exits before child: init becomes pseudo parent

Process Scheduling
Concurrency/ pseudoparallelism
- Has to be fair, balance of resources
- Scheduler triggered (timer interrupts), decides if context switch is needed
- Cooperative vs. preemptive (fixed time quota, time slicing)
Batch Processing
- No user interaction, non-preemptive
- Turnaround time: finish – arrival time
- Throughput: #tasks/ unit time
- CPU utilisation: %time CPU busy
First Come First Serve - FIFO, no starvation
- Convoy effect: long process runs first
Shortest Job First - Minimises average waiting time, but starvation possible
- Need to est. total CPU time for a task
Shortest Remaining Time
- New job can preempt current running job
- Good svc for short jobs, even w/ late arrivals
Interactive Systems - Response time: response - request time, predictability (variance in response time), preemptive: scheduler runs periodically)
- Time quantum: multiple of Interval of timer interrupt (ITI), context switches when time quanta over, remaining time quantum: next process can be scheduled
Round Robin - FIFO with fixed time slice
- Response time guarantee: (n-1)*q
- Larger quantum: more CPU utilisation, but longer response time
Priority Scheduling – optional preempt: high P need not stop current low P, P1 = highest
- Low priority process can starve (decrease priority after every time quantum)
Priority Inversion: P1 process depends on resource locked by P3, P2 runs instead
Multi-Level Feedback Queue (MLFQ)
- Minimises response time for I/O bound and turnaround time for CPU-bound processes
- Priority reduced if job fully utilises time slice (addition: use cumulative time, periodically shift to highest priority)
Lottery Scheduling - % tickets owned
- Responsive: new process has a chance to run

Inter-Process Communication
Shared Memory
- P1 creates M, P2 attached M to its own memspace (synchronisation problems)
- UNIX: shmget, shmat, shmdt, shmctl
Message Passing - Msg in kernel memory space [Direct: explicitly name other party (must know identity)/ Indirect: Mailbox; shared among multiple processes]
- Synchronous: blocking
UNIX Pipes - Input, output, error
- Circular bounded buffer, FIFO data
- Producer-Consumer relationship
- dup2 for input/ output redirection
UNIX Signals - Async (interrupts); kill, stop, continue, errors…

Threads
- Lightweight; share code, data and files
- Duplicated registers and stack (SP, FP points to different location in same main stack)
- More resource sharing, responsive, scalable, less overhead, less protection
- Can execute different threads in parallel
- fork(): usually only one thread. Exit()? Exec()? Which thread handles signal?
- User thread: library, kernel unaware, one thread blocks → whole process blocks
- Kernel thread: thread-level scheduling

Synchronisation
Race Condition - Non-atomic instructions, outcome depends on order of execution
Critical Section (CS) - Mutual exclusion, progress, bounded wait, independence
- Deadlock: all blocked, Live Lock: no progress due to deadlock avoidance mechanism, Starvation: some blocked forever
TestAndSet - Atomic, assembly-level
- Load content at memory location into register, set memlocation = 1
- Busy waiting (processes still scheduled)
- Bounded wait: depends on scheduler
Peterson Algorithm – sync 2 processes
- Want; ensures independence, P1 does not have to wait for P0 to enter CS first
- Turn; Ensures mutual exclusion: only holds one value at each time, prevents deadlock: both “want” = 1. Lets other have turn first.
- Busy waiting
Semaphore - Atomic wait, signal
- Wait: s <= 0: block; s--;
- Signal: s++; wakes up one sleeping process (if any)
- *Invariant*: $S_{current} = S_{initial}$ + #signal(s) - #wait(s), wait() must be completed
- Binary semaphore: mutex
- #in CS = #wait(s) - #signal(s) <= 1
- No deadlock ($S_{current}$, N_{CS} cannot both be 0)
    . Unless two semaphores blocked
- No starvation, assuming fair scheduling
Producer-Consumer (e.g pipe)
Reader-Writer (e.g. files) - mutex used to protect CS of nReader, writer can be starved (use turnstile)
Dining Philosophers
- deadlock: all have left only
- livelock: all unable to pick right
- States: Think, Hungry, Eat
Tanenbaum Solution
- TakeChopsticks: wait on chopsticks, if available signal to self to eat
- PutChopsticks: check if left and right available to eat, if so signal to them
- Limited Eater: n -1 people for n seats

Memory Management
Memory Abstraction
- compile time: bind to memory addr in RAM
- Memory segmentation (Base+Limit register)
- Logical address != physical address
Contiguous Memory Management
- For multitask: multiple proc in physical memory, free physical memory by removing when terminated or swap to secondary storage
Fixed-size partitioning: internal frag
Dynamic partitioning: external frag
First Fit, **Best-Fit** (least ext frag), **Worst-Fit** (remaining space usable)
- Merge (adjacent only) vs compaction
- Linked list: [T/F | Start Addr | Size | Ptr]

Buddy System – repeatedly divide into 2
- Array of size k, arr[i] = linked list of blocks of size I. Fix smallest allowable block size.
- De-allocation merge: only Sth bit complement, leading same (buddies)

Disjoint Memory Schemes

Paging
- physical frames ↔ logical page (same size)
- Page Table for logical addr translation
- Page/Frame # + Offset
- May still have internal frag, no ext frag

Translation Look-Aside Buffer (TLB)
- 2 mem access (PT in OS-RAM + frame)
- Cache, part of hw context (need to flush)

Page Protection - access right bits, valid bit (process need not cover logical mem range)
- Page Sharing: page table has same frame# (system calls, copy-on-write)

Segmentation – text, data, heap, stack
- Segment Id (base) + limit (size)
- Access = segment name + offset
- Valid: offset < limit
- Segments are independent
- W paging: segment table points to page table, limit is now for #pages
- Address = [Segment | Page | Offset]

Virtual Memory Management
- logical memory space >> physical memory
- Use a memory resident? bit
- Secondary storage (non-memory resident) – raise page fault (TRAP OS), use swap space
- Temporal/ spatial locality → less thrashing

Page Table Structure

Direct Paging – e.g. virtual addr 32 bits, page size 4KiB, total = 2^{20} pages, Page Table Entry (PTE) = 2bytes, requires 2^{21} = 2MiB

2-Level Paging – not all processes use full range of virtual mem space
- Smaller page tables, page table#
- Page directory pointing to page table

Address = [PgDir9 | PT ptr11 | w/in Pg12]
- PgDir: each entry 2bytes * 2^9 entries

Inverted Pg Table map frame → pid, page#

Page Replacement Algorithms
- dirty page evicted → have to write back

Optimal OPT (benchmark standard) – page not used for longest period of time

FIFO – evict oldest memory page (Belady's Anomaly – more frames but more pg faults)

Last Recently Used (LRU) – use counter with last used (have to search through table) or use stack (can remove anywhere, push)

Second Chance (CLOCK) – FIFO-like
- Reference bit = 1, recently accessed
- Clear ref bit until victim page reached

Frame Allocation Policies – equal vs proportional allocation btwn processes
- Local replacement: within process (can hog I/O), vs global (cascading thrashing)

Working Set Model – time + delta before it
- Allocate enough frames for pages in WS

File System – abstraction for physical media
- Self-contained, persistent, efficient
- vs (mem) disk sectors (addr), explicit access

File System Abstraction – file + directory
- data + metadata (attributes)
- File type (windows = ext, unix = magic no.)
- Protection: owner, grp, universe (rwx) or Access Control List (file → allowed users)
- File data: fixed vs variable length records
- Sequential (no skip, can rwnd) vs random (seek) vs direct (each record = 1 byte)
- Track opened files (file pointer, disk location, open count): System-wide (Open File Table): 1 entry per unique file vs Per-process (File Descriptor Table): pt to OFT
  . Fork(): share fd, same process: diff fd
- Directory = grouping of files, *Single-Level*, *Tree* (absolute ('/') / relative current wd)
  . Alias – DAG: hard link (copy file) vs symlink (store path of file/ directory)

File System Implementation
- logical blocks mapped to disk sector
- sector 0 = master boot record (MBR) with partition table (each can be independent FS)
- Partition: OS boot block, partition details, directory structure, file info, file data

Implementing Files
- file = collection of logical blocks
- contiguous (ext frag) vs linked list (random access is slow, part of disk block used for ptr)
- Improved LL: File Allocation Table (FAT), all block ptrs in one table (in memory)
  . [Index] → [Next ptr / -1], large table even though not all disk blocks used
- Indexed Allocation: index block stores disk blocks used by files in order [9|16|1|-1|-1]
  . file → index block #
  . LL of index blocks, multi-level index

Free Space Management

Bitmap, 0 = occupied, 1 = free

Linked List of free space disk blocks (store free block numbers

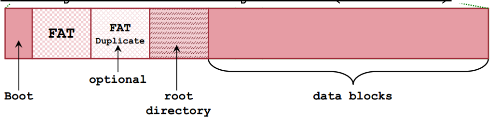
Implementing Directory
- track files, metadata + map file name to info

Linear List with cache

Hash Table with chained collision (multiple linked list), hash e.g. len(filename)
- file info: store metadata in dir, or just ptr

Disk I/O Scheduling
- rotational latency, seek time
- FCFS, Shortest Seek First (SSF), SCAN (1→21, 21→1), CSCAN (1→21, 1→21)
- *deadline*, 3 queues (sorted – non-urgent, read 500ms, write 5s), *noop* for SSD, *completely fair queuing (cfq)* with time slicing, per-process sorted queues, *budget fair queuing/ multiqueue* fair sharing based on # sectors requested

File System Case Study – FAT (Windows)



FAT Table: FREE, EOF, BAD, next block #
- cached in RAM to facilitate LL traversal

Directory: 32byte entries
- Each dir is a block, including root (special)
- [Filename 8 | Ext 3 | Attribute 1 | Reserved 10 | Creation Date, Time 2, 2 | First Disk Block 2 for FAT16 | File Size in Bytes 4]
- Attribute: directory? Hidden? System?
- Delete file: del dir entry, update FAT entries, first char of filename = E5
- Undelete: can recover all except first byte of name, first data block
- Support larger hard disk: (1) disk cluster, change smallest allocation unit (2) larger FAT size [actual less, special val cant ref blocks]
- FAT32: only 2^{28} clusters, 4 bits reserved
- Long file name: VFAT up to 255 char or use multiple directory entries, first byte indicates sequence – use invalid file attr to ignore these extra dir entries

File System Case Study – EXT2 (Linux)
- blocks form block groups
- file/ directory inode (index node, metadata)
- Each block: [Superblock | Group descriptors | Block bitmap | Inode bitmap | Inode table | data blocks]
  . Superblock: describe whole FS total# inode, group, block, repeat in all block groups
  . Group desc: describe block group (#free blocks, Inodes, bitmap location), duplicated across all block groups for redundancy
  . 1 = occupied, 0 = free
- Inode = 128 bytes, [mode (file/ dir) 2 | owner uid/gid 2/2 | file size 4~8 | timestamp (create, modify, delete) 3*4 | data blk ptrs 15*4 (12 direct, 1 single/ double/ triple indirect) | ref count 2]
- Directory data blocks: linked list of directory entries (no fixed size)

[Inode# | entry size | type (F/D) | len(name) | name], Inode# 0 = unused entry
- e.g. root at 2, go to inode2 to find datablock2. Find inode# of subdir
- Delete file: remove dir entry (update ptr), update inode/ datablock bitmaps
- Undelete: recover ptr in LL, update bitmaps
- Hardlink: dir entry with same inode#, increase reference count
- Symlink: new file, content = pathname of file

Extra Topics
- Consistency checks in FS: redundancy
- Defragmentation
- Virtual FS, abstraction vs. NFS (distributed)