OS Basics

OS: providing execution environment for apps
abstraction, protection, resource managm.

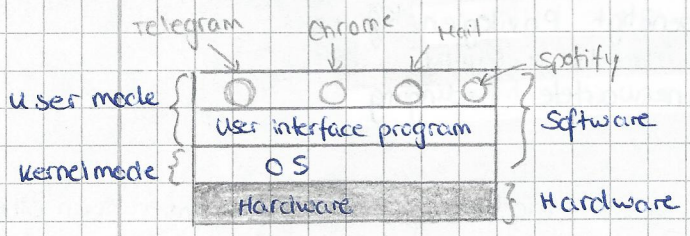
- OS ist Abstraktionslevel zw. Applikationen und Hardware
  - → verwaltet und versteckt Hardware-Details
  - → low-level Interfaces für Hardware Zugriff
  - → macht Hardware für mehrere Apps verfügbar

- OS bietet Schutz → User und Kernel Mode
  - → vor Nutzern / Prozessen, die alle Ressourcen verbrauchen (accounting & allocation)
  - → verhindert das Schreiben eines Prozesses in den Speicher eines anderen

OS Definition:

(1) OS = resource manager
- verwaltet und verteilt Hardware-Ressourcen
- entscheidet bei mehreren Anfragen wer Ressource bekommt
- bemüht sich um fairen und effizient Ressourcen Gebrauch

(2) OS = control program
- kontrolliert Programmausführung
- verhindert Fehler und missbräuchlichen Gebrauch des Computers



HARDWARE

- CPU(s), Speicher, Geräte verbunden über Bus
- CPU(s), Geräte konkurrieren um Speicherzyklen und Bus
- alles läuft gleichzeitig

CPU:

- Holt Befehle aus dem Speicher und führt sie aus
  - → instruction format / -set abh. von der CPU
- hat eigene Register, die Daten, Metadaten während Ausführung speichern

User Mode: nur nicht-privilegierte Instruktionen werden ausgeführt
→ kein Hardware-Management

Kernel Mode: alle Instruktionen erlaubt → HW-Management mit privilegierten Instr.

RAM: hält aktuell ausgeführten Befehle + Daten

Cache: RAM langsamer als CPU → speichere ausgewählte Daten näher an CPU im Cache

CPU Registerzugriff: ~1 CPU cycle
L1 Cache pro Kern: ~4 CP cycles
L2 cache / 2 kerne: ~12 CPU cycles
L3 cache / LLC per uncore: ~28 CPU cycles

priv. instructions:
- manipulate control registers for mem. translation → breaks mem isolation else
- disable/enable interrupts
- access to platform devices

- hardware managed

- Cache hit: Daten sind im Cache
- Cache miss: Daten müssen aus tieferen Speichern geholt werden
  - > Compulsory miss: erster Datenzugriff
  - > capacity miss: Cache zu klein für working set
  - > conflict miss: Platz vorhanden, Problem mit platzierungs-Stratgie

OS Invocations

! OS does not always run!

Kernel läuft NICHT immer im Hintergrund

-> 3 Fälle, die Kernel aufrufen und in kernel-mode wechseln

	involuntary	voluntary
synchr.	exceptions	Syscalls
asynchr.	Interrupts	?

man 2 syscalls
man 3 CAPI

System calls: user mode processes require higher privileges
Interrupts: CPU - external device sends a signal
Exceptions: CPU signals unexpected condition

System calls: Idea: run processes in user-mode to protect them from each other
~> processes now restricted -> who manages their hardware needs?
-> Application calls system if service needed (System call)
-> OS checks permissions
-> if app is allowed to perform that action: OS performs it in kernel-mode

- often used via APIs (application program interfaces) -> Win32 for Windows
  (eg. printf call uses write syscall) -> POSIX for UNIX / Linux / OSX

- only one syscall interface (entry point) into the kernel: **trap instruction**
  -> switches CPU to kernel mode, always enters kernel in the same way
  -> Syscall dispatcher in kernel as multiplexer
  -> syscalls id with a number passed as parameter (passed via registers / stack / main mem + location register)
    <> syscall table maps number to kernel function
    <> dispatcher decides where to jump

Kernel must validate params, e.g. check if addr is in own AS

Interrupts: Devices use them to signal predefined conditions to OS (via "interrupt line to CPU")
- Programmable Interrupt Controller manages interrupts
  -> interrupts can be masked (ignored) and are stored in a (finite) queue for when interrupt is unmasked -> interrupts can get lost

Examples:

- Timer-Interrupt: periodically interrupts processes, switches to kernel
  -> can switch to different process to enforce fairness

- Network Interface Card: interrupts CPU when packet was received
  -> can deliver packet to process, free NIC buffer

- on interrupt: CPU looks up interrupt vector (table in memory) lookup
  → transfer controll to resp. interrupt service routine in OS (handle interrupt)
  to handle interrupt Entry
  ↪ must save the state of the interrupted process
  (instruction pointer, stack pointer, status word) → Execution
  ↪ CPU returns from service routine, restore previous CPU context Exit

Exceptions:
CPU interrupts the program, kernel gets control
↪ kernel can determine reason for the exception
↪ if it can resolve the problem: does so, continues faulting instruction,
else kills the process

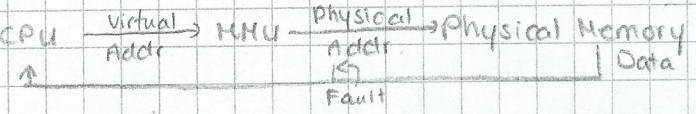
Interrupts ↔ exceptions difference:
(1) source ↪
(2) interrupts can happen any time in any context
(3) exceptions always occur synchronous and in the context of a process

OS concepts - Overview

Virtual Memory: "if you can't name it, you can't touch it"

- every job has its own address space → illusion of having all memory
→ cannot access memory of others → protection
~> always, only use virtual adresses

Indirect Addressing: MMU translates virtual addr. to physical addr.



- Kernel part of addr. space, kernel-only virtual addr. → apps can't touch kernel memory
- MMU can enforce read-only virtual addr.
  ↪ safe memory sharing btw. apps
- MMU can enforce execute disable

Page Faults:
- addr. not mapped
- access kernel memory (illegally)
- write read-only memory
- set instruction pointer to executable disable memory
- ...

Processes: = program in execution = "instance" of program

- PCB (process control block, info about allocated ressources) per process
- virtual addr. space per process, starts at 0 up to maximum
  ↪ laid out in different sections, addr. btw. sections illegal → page fault (segm. fault)
  ↪ usually kills process

Scheduling

Mechanism: implementation of what is done ⟷ Policy: rules which decide when what is done and how much

Mechanisms can be reused even when the policy changes

→ multiple processes and threads available → os needs to switch btw. them

Scheduler = policy, decides which job to run next
dispatcher = mechanism, performs the actual switch

- schedulers try to provide fairness while meeting goals and adhering priorities

Files

- device-independent files and directories for pesistent storage
- file system = ordered collection of blocks
- processes can communicate directly via a special named pipe file
- directories form a directory tree / file hierachy with root directory as root
- in Unix: multiple file systems → single file hierarchy (mounting)

Storage

OS provides uniform, logical view of information storage to file systems
- Drivers hide specific hardware devices

OS increases performance of I/O devices
- Buffering: store data temporarily while it's being transfered
- Caching: Store parts of data in faster storage for performance
- Spooling: overlap of output of one job with input of other jobs

Processes

Multiprogramming: quick process switching → models concurrency

→ when switching a process, the execution context changes
  → dispatcher switches
  → on a context switch, the dispatcher saves current registers and memory and restores those of the next process

cpu-bound: rarely I/O ops, long computations
I/O-bound: short computations, many I/O ops

! process = program in execution

↳ program describes what and how to do
↳ process actively doe executes this program

! Concurrency != Parallelism

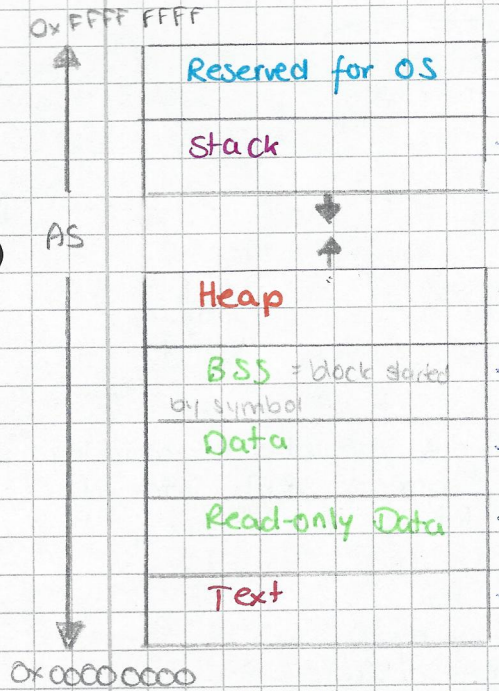
(a) Concurrency = Pseudoparallelism = multiple processes on the same CPU
(b) Parallelism = processes truly running at the same time

Address Spaces unique per process

MMU maps virtual addr. to physical addr. on each load/store

⊕
- MMU enforces protection
- programs can see more memory than available

⊖
- need special MMU hardware



→ Addresses where the kernel is mapped (no access for process)
→ Local variables, function call params, return addr. private
→ Dynamically allocated data (malloc) private
→ Uninitialized local variables declared as static private
→ initialized data, global variables private
→ Read-only data, strings } shared accross processes of same program
→ Program, machine code

- instruction pointer = addr. in text segment
- stack pointer = lower-most addr. of stack segment
- program break pointer = upper-most addr. of heap segment

ELF Dateien: code(=text), RO Data, RW Data, BSS
nicht dabei: stack, heap

(1) Fixed-size data and code-segments:

- BSS segment: (Block started by Symbol), exec file contains starting adr. and size of BSS, BSS initially zero
- data segment: fixed size, initialized data elements (e.g. global v
- read-only data segment: constant numbers and strings

(2) stack segment:

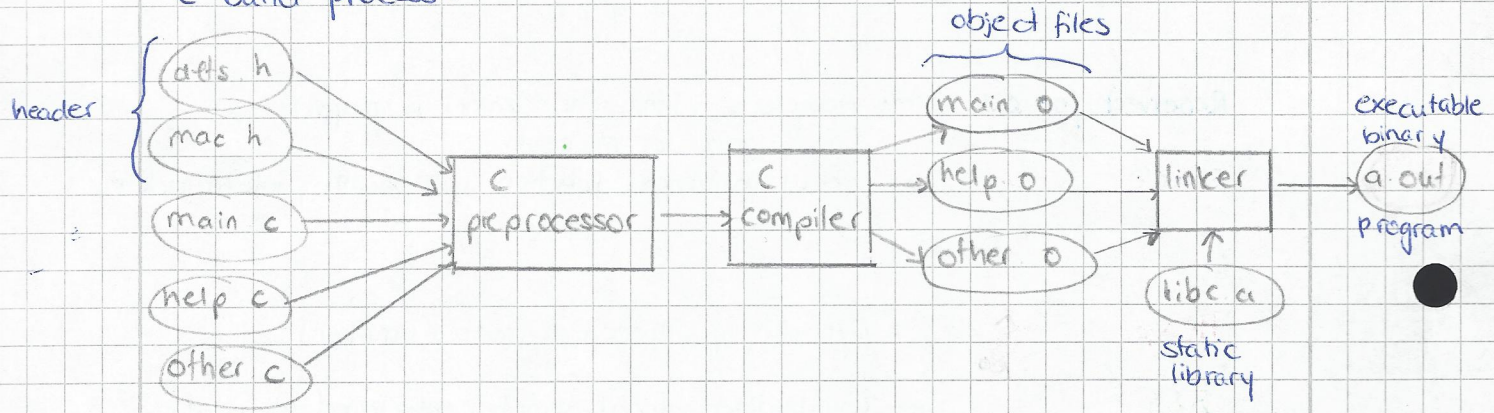
- grows downwards
- allocate = push: SP -= a; return (SP + a); a byte structure
- free = pop: SP += a

(3) Dynamic Memory Allocation: Heap segment

- allows "random" malloc/free
- allocate memory in two tiers:
  (1) allocate large chunk (heap) from OS → base adr. + break pointer
  (2) dynamically partition large chunk into smaller allocations
  → malloc / free, purely in user space at this point!

Compiling, Linking, Loading Programs

C Build process:



.o object code files: contains instructions and data generated by compiler
→ objects structured by - segments (text, data, ...)
- labels (function names, ...)

! in C label name = function name

Linker: builds the executable from .o files by
- arranging segments in non-overlapping memory regions
- construction of global symbol table (labels → addr.)
- patching addr. in code (relocation)
- writing results to binary file

Loader:
- reads code/data segment from disk into buffer cache
- allocates space for heap, stack and BSS (and nulls BSS)
- inits rw-data and data sections
- maps code read-only and executable
- lots of optimizations!

Dynamic Shared Libraries: loading shared lib at any virt. addr.
→ position-independent code (PIC)
- Global offset table (GOT): maps stubs to functions in various dynamic libs
- Procedure linkage table (PLT): contains stubs pointing to the GOT for functions that are linked in dynamically
- Lazy dynamic binding: link each function on its first call, not at startup

Execution Model:

OS interacts directly with compiled programs
- switch btw. processes/threads → save/restore data+state
- deal with + pass on signals and exceptions
- receive requests from applications

x86 Stack
- stack pointer register holds addr. of top of stack
- base pointer register (frame pointer) used to organize larger chunks of the stack (stack frames)

Application Binary Interface (ABI)
- standardizes binary interface btw. programs/modules/OS
  - specifies executable/object file layout, calling convention, alignment rules
- calling conventions standardize the exact way function calls are implemented to achieve interoperability btw. compilers

general purpose registers:
- accumulator
- base
- counter
- data
- stack pointer
- base pointer

x86 Calling Conventions

- function caller: saves stak of local scope, sets up params where subroutine can find them, transfers control flow
- called function: sets up new local scope (local vars), performs its duty (probably calls other function), puts return value where caller can find it, jumps back to caller

example: cdecl: - accumulator (A), counter, data (D) caller-saved
others callee-saved
- function args passed via stack (reversed order)
- return addr. saved on stack
- return value passed via stack or A+D registers

Parameter Passing to the System on syscall

! syscall number always passed, other params optional

- ABI specifies how to pass params
  > either via CPU registers (max. ~6) + rest via main memory (heap/stack)
  > or all via heap/stack → to access them: (1) save user stack pointer in a register (2) map user stack to kernel (3) copy args from user stack to kernel stack
- return code returned to app: <0: error, ≥0: success
  <> usually via A+D registers

System call handler: implements actual service
(1) saves registers it taints → (2) read passed params → (3) sanitizes/check params! → (4) checks permission of process performing action → (5) performs the request → (6) returns to caller with success/error code

Process API

Process Creation

processes created when
- booting (system init)
- dedicated syscall issued by running process
- users requests to create a new process
- initiation of a batch job

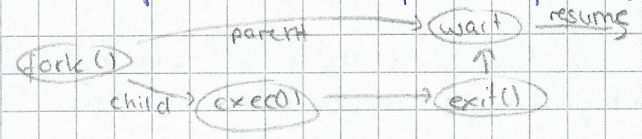
⇓ done by two mechanisms

(1) Kernel spawns initial user space process on boot (Linux: init)
(2) user space processes spawn further processes, e.g.
  - Windows: createProcess / Posix: fork
  - Windows: click on file (calls createProcess)
  - Linux: cron daemon is started on boot

POSIX: fork

- PID = process identifier for each process
- pid = fork() duplicates current process
  → returns 0 to new child
  → returns new PID to parent
  } can continue differently in child and parent after fork

- exec(name) replaces own memory based on an executable file (name)
- exit(status) terminates own process and returns exit status
- pid = waitpid(pid, &status): wait for term. of child (pid), status returns info about process (exit status, ...)
  ↪ returns pid on success, -1 on failure



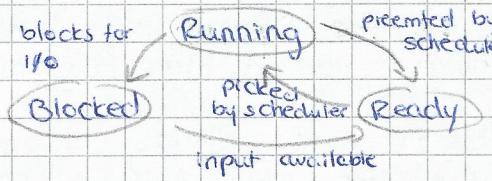
POSIX: Process Hierarchy

- parent process creates child processes, create child processes, ...
  → process tree

- parent and children share resources (part of the AS)
- parent and children execute concurrently
- parent waits for childs termination to collect their exit status (waitpid)

Daemons: processes that run in the background
↪ detached from parent after creation, reattached to root of process tree (init)
  → init automatically collects exit status (and ignores it)

Process states:



Process termination: (1) (voluntary) normal exit: return 0; at end of main / exit(0)
(2) (voluntary) error exit: return x, x != 0; at end of main / exit(x) / abort()
(3) (invol.) fatal error: killed by OS after exception / exceeds resources
(4) (invol.) killed by other process (only with permission ↪ parent or admin)

exit status: - return int on voluntary exit (last 8 Bit significant on Linux)
- zombie / process stub exists until collected via waitpid
- orphans: running childs after parent was killed → init usually adopts them

Threads

shared btw. all threads: code, data, file
per thread: registers, stack

- Processes provide abstraction of addr. space and ressources
- Threads provide abstraction for execution states of that AS/container
  -> in Linux: threads = regular processes with shared ressources, addr. spaces
  -> some data thread local, some thread global (but process local)

- use threads instead of processes when you want to share data

POSIX Thread API:
each thread has ID, register set (incl. IP, SP), stack area
- pthread_create: new thread, params: pointer to pthread_t, attributes, start func, args
  returns 0 on success
- pthread_exit: terminate calling thread, param: exit code, frees resources
- pthread_join: wait for thread to exit, params: id of thread, pointer to pointer for exit code
  returns 0 on success
- pthread_yield: Release CPU to run another thread

Process: group ressources <-> thread: encapsulate execution

PCB / TCB
PCB = Process Controll Block: info needed to implement process
TCB = Thread control Block: per thread data

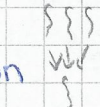
	PCB	TCB
process state	addr. space	instruction pointer
-"- identif.	Open files	registers
Progr. counter	Child processes	stack
	pending alarms	state

- PCB known to OS
- whether OS knows about thread depending on thread model

Thread models

Kernel threads = threads known to OS kernel
User threads = threads known to process

Many-to-One Model: User Level Threads
- kernel only manages process, threads unknown
- threads managed in user-space lib

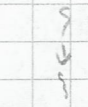


⊕ faster thread managment operations, flexible scheduling policy, few system resources can be used even if OS doesn't support threads

⊖ no parallel execution, whole process blocks if one thread blocks, needs re-implementation of parts of OS (e.g. scheduler)

"Main stack" + thread stacks allocated dyn. on heap

One-to-one Mode: Kernel Level Threads
- kernel knows and manages every thread

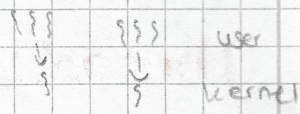


⊕ real parallelism possible, threads block individually
⊖ OS manages all threads in the system (TCB, stacks, ...), syscalls needed for that, scheduling in OS

one stack per thread in stack segment, Shared heap

ULT = entirely implemented in user level (control structur, scheduling, ...) depending on model, kernel not aware of ULT

M-to-N-Model: Hybrid Threads
- M ULTs mapped to N KLTs (M ≥ N)
- flexible allocate ULTs on KLTs



⊕ flexible scheduling policy, efficient execution
⊖ hard to debug, hard to implement (blocking, # of KLTs, ...)

Dispatching + Scheduling

Dispatcher: performs process switch (mechanism)
→ saves/restores process context, switching to user mode

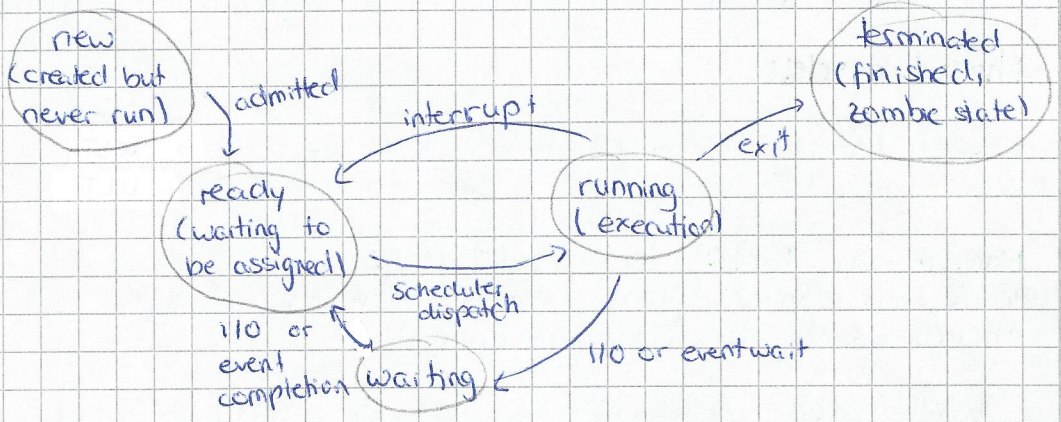
Scheduler: selects next process to run (policy)

→ kernel can only dispatch when it is run!
↔ kernel can switch at any syscall

- yield: process frees CPU voluntarily (cooperative multitasking)
- preemption: forcefully pause process to schedule another
  ↔ preemtive scheduling requires kernel
  ↔ timer interrupt as trigger after every time-slice (1ms - 200ms)

on context switch: save state of current process in its PCB
restore state of new process from its PCB

Process State:



short-term scheduler = CPU scheduler
- selects next executed process, allocates CPU, invoked frequently (millisec.)

long-term scheduler = job scheduler
- selects processes for ready queue, invoked infrequently (secs, mins)
  ↔ controls the degree of multiprogramming

Job queue: set of all processes in the system
ready queue: processes in main memory (waiting or ready)
device queue: processes waiting for I/O device

Scheduling Policies

goals: fairness, balance (keep the system busy)

- Batch scheduling: no quick response needed → less switches → less overhead
- Interactive scheduling: quick responses needed
- Real-time sched.: guarantee completion in time x

- turnaroundTime = completionTime - arrivalTime = time from submission to completion
- process utilization: percentage of time processor is not idle
- throughput = # processes / threads completed per unit of time
- waiting time = time spent in the ready queue = $t_{finish} - t_{arrival} - t_{burst}$
- response time = time from submission to first response

FCFS: Schedule processes in order of arrival — first come, first serve
↳ good for short bursts, urgency

SJF: shortes job first, optimal average turnaround times
↳ estimate length of job
t_n = actual length of n^{th} CPU burst
→ $T_{n+1} = \alpha t_n + (1-\alpha) T_n \quad (0 \leq \alpha \leq 1)$

Problem: Starvation

PSJF: preemtive shortes job first; SJF with periodical preemtion → new sched. dec.

Round Robin: queue of processes, regular preemtion, if not finished → to end of queue
problem: time slice length needs to balance interactivity and overhead
= FCFS with infinite time slice = FCFS
→ unfair for I/O-bound jobs (block before using up time slice)
↳ idea: virtualRR: extra queue for this case with higher priority

Priority sched.: each job has priority → stored in priority queues
↳ run job with highest priority (smallest int usually)
prevent starvation with aging: increase prio over time of never run jobs

inside queues: RR

MLFQ: Multi-Level Feedback Queue, higher prio to I/O-bound jobs; lower Prio to CPU-bound jobs, but longer time-slices
↳ different queues for prios and time-slices
↳ increase prio if job does not use up time-slice
↳ double time length for next-lower prio
↳ demote jobs that repeatedly use up their time-slice

Priority donation: B waits for A, A has lower prio than B
→ B gives its prio to A while waiting

Lottery scheduling: # of tickets for each job, more tickets for higher prio
↳ draw random ticket number, assign owner to CPU
ticket donation: transfer tickets to jobs this job is waiting for

IPC = Interprocess communication

- Message passing: explicit send/receive information using syscalls
- shared memory: physical mem region with multiple access

Direct / Indirect Messages:

> direct messages: processes name each other explicitly (send, receive)
> indirect messages: send/received from mailboxes with unique ID
(created by first process, destroyed by last)
↳ processes can only communicate if they share a mailbox

Sender/Receiver Synchro:

> Blocking = synchronous: blocking send blocks sender until message is received
blocking receive blocks receiver until message is available

> non-blocking = asynchronous: non-bl. send sends the message, sender continues
non-bl. rec.: receiver receives valid message or null

Buffering:

queue messages
> 0-capacity: sender must wait for receiver (rendezvous)
↳ no latency
> Bounded-capacity: finite # and length of messages
- sender can send as long as pipe is not full, else wait

> unbounded-capacity: sender never waits, memory may overflow

Shared memory:

shared region in memory
btw. processes: shared region in one addr. space
↳ every write visible to all other processes
!carefull!: race conditions!

in general: no sequential consistency (memory ops in program order, write is atomic)
→ synchronization needed

Synchronization:

Race condition: no atomic memory ops → more than one thread enters critical section, can overwrite each others data

Solution needs:
- Mutual Exclusion: only one thread can be in critical section at any time
- Progress: no thread running outside the CS can block a thread from getting in
- Bounded Waiting: faire share of access to CS between requesting threads

Disabling Interrupts: "do not interrupt"-bit per thread, disables interrupt until thread leaves CS
⊕ easy and convinient in the kernel
⊖ only works on single-core systems, only feasible in kernel

„Spinlock"
Atomic lock: test + set lock atomically → use atomic instructions
⊕ ME, Progress ⊖ no bounded waiting (no upper bound)

Spinlocks don't work well if: lock is congested or threads on different cores use the lock

Coarse Locks: small # of locks, each protect large segment
Fine-grained Locks: larger # of locks, each protect small segment

typical problems: Producer-Consumer shared use of buffers/queues
Readers-Writers shared access to data
Dining Philosophers limited resources

Semaphore: wait (&s): if s > 0: s--, continue, otherwise caller sleeps
signal (&s): no waiting thread: s++, otherwise wake one
s = max # threads to enter CS
s = 1 → binary semaphore = mutex

wait and signal must be synchronized → race condition for s
→ signal loss

weak sem.: wake random thread
strong sem: FIFO
→ solves all 3 problems

better: futex

Futex:
- userspace + kernel component
- try to get in CS with userspace spinlock
- if CS busy: syscall to sleep
- otherwise: enter CS with now locked spinlock completly in user space

Spinlocks → good on multiprocessor systems, if blocking is not viable option
① quick when waiting time is short
☹ wastes resources on long waiting time

Semaphores
☺ efficient when long waiting time
☹ syscall overhead at every operation

Condition Variables: allow blocking until condition is met (pthread_cond_*)

Readers-Writers Locks: multiple reads, only on writer in CS
if readers present while writer tries to enter CS:
don't let more readers in → block until all readers finish
→ let writer in

Deadlocks:

arises when 4 conditions hold simultaneously:

(1) Mutual exclusion: limited access to resource, resource can only be shared with finite amount of users

(2) Hold and wait: wait for next resource while already holding at least one

(3) No Preemption: once the resource is granted, it cannot be taken away forcefully

(4) Circular wait: Possibility of circularity in graph of requests

Countermeasures:

Prevention: negate at least one of the four conditions

(1) Mut. excl.: buy more resources, split into pieces, virtualize

(2) Hold. & wait: get all resources en-bloque, 2-phase-locking

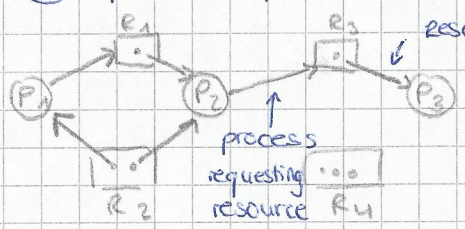
(3) No Pr.: virtualize to make preemptable (virtual vs physical mem, spooling (printer))

(4) Circ wait.: ordering of resources, partial order on resources (always m_1 befor m_2, ..)

Avoidance: no deadlocks in safe state
→ on every resource request decide if system stays in safe state
↳ needs a-priori info

→ RAG = Resource Allocation Graph

(P1) process , R1 [• •] resource with two instances



resource assigned to process

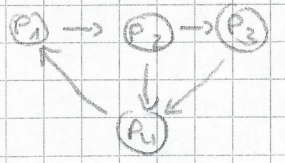
process requesting resource

----> : "claim edge" process may request resource

! cycle != deadlock

Detection: allow deadlock → detection → recovery scheme

→ WFG = Wait-For Graph , nodes = processes
edge = "wait for" (like RAG without resources)



cycle => deadlock

Recovery: possibilities

(1) Process Termination: abort all deadlocked processes
or one process at a time until cycle eliminated
(choose order by priority, running-time, resources used/needed, activity)

(2) Resource Preemption: select a victim (minimize cost) → Rollback: periodic snapshots
abort process to preempt resources, restart process from last safe state
Problem: starvation if always same victim
<> cost-factor depends on # of rollbacks

Memory Management

MMU = Memory-Management Unit

program/data stored in background storage → must be brought to main memory

Swapping:
- roll-out: save program's state on background storage
- roll-in: replace it with another's state

⊕ Hardware-support only to protect kernel, not needed to protect processes from eachother

⊖ very slow, no parallelism since main memory is used entirely by one process AS

Properties for shared memory:

Protection: bug in one process mustn't corrupt anothers memory, protect processes from memory reads/writes from other processes

Transparency: processes shouldn't require particular physical memory, should be able to use large amounts of contiguous memory

Resource exhaustion: sum of sizes of all AS can be greater than physical memory

→ needs hardware support: MMU maps VA to PA, checks flags

Base + Limit Register
- on every load/store: VA ≥ base ?, VA < base + limit ?
  ↪ VA as PA in memory

⊕ only load base+limit to switch AS
  quick at run-time → only two comparisons

⊖ no AS grow possible
  no data/code sharing

Segmentation
- multiple base+limit pairs per process → keep some segments private, share others
- VA = <seg#, offset>
- map with segment table: base in memory, limit = size/length, protection bits (read/write/...)
- two registers to id current AS:
  - segment-table base register (STBR): pointer to location of current table
  - segment-table length register (STLR): # of segments used by process

⊕ data/code sharing safely possible
  easier placement: no contiguous mem. needed in physical mem.
  don't need entire process in mem.

⊖ segments need to be contiguous in mem.
  fragmentation of physical mem.

Fragmentation: = inability to use free memory

external: sum of free mem. enough, but not contiguous → compaction needed
  is expensive: needs to halt process, reload caches

internal: unused parts of page of process 1 can't be used by process 2

Paging: - divide physical mem. into fixed-sized blocks: PAGE FRAMES (size power of 2 bytes, e.g. 4KiB, 2MiB, 1GiB)

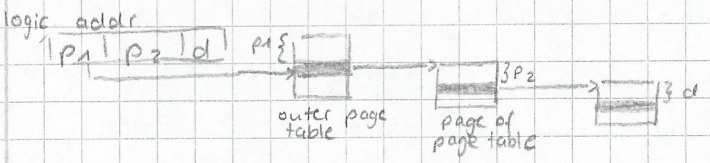
- divide virtual mem. into blocks: PAGES → same size as frames!

OS { - page table stores mapping of VPN to PFN for each AS
- keep track of free frames

- Present bit: indicates if virtual page is currently mapped to physical mem. = valid bit

linear page table { - Virtual Adr. = virt. page # (index in the page table which contains base adr. of each page in physic. mem.)
page offset: concatenated with base adr = physic. adr.

Hierarchical Page Table:
subdivide VAs further into multiple page table indices



PTE content:
> valid bit: if page currently available or needs to be brought by OS (page fault)

> page frame number: if page present: at which phys. adr. is page located

> write bit: may the page be written to? → page fault on unallowed write

> caching: should the page be cached and with which policy?

> Access Bit: set by MMU if page was touched since bit was last cleared by OS

> Dirty Bit: set by MMU if page was modified since bit was last cleared by OS

OS involvement:
- bring data into memory, page allocation
- page replacement
- context switching (set MMUs base register to point to page hierarchy of next process AS)

Page Size: Large vs. small pages

> Fragmentation
LP: internal fragm. → wastes mem.
SP: less wasted mem.

> Table size
LP: fewer bits for PFN → more for offset
fewer PTEs
SP: More, larger PTEs

> I/O
LP: more data loaded from disk to make page valid
SP: trap into OS more often when loading large program

2 level PT: < page dir. index, page table index, offset >

Linear inverted Page Table:
map phys. frame to virt. page → single table for all processes, one entry for each phys. page frame

⊕ less overhead for PT meta data
⊖ increases search time

Hashed Inverted PT:
VPN → Hash function → Index into Hash anchor table, points to part of inverted page table

TLB = Translation Lookaside Buffer
stores recent memory translations
↳ maps <vpn> to <pfn, protection>

typically: 4-way to fully assoc. hardware cache in MMU
64-8k entries, ~95-99% hit rate

On every load/store:
check if translation already cached in TLB → TLB hit
otherwise walk PTs and insert result in TLB → TLB miss

⊕ quick: compare many TLB entries in parallel in hardware

TLB Miss:
evict entry from TLB and load new entry

MIPS → software-managed: OS receives TLB miss exception, decides for victim
→ walk PTs in software and fill new entry (format defined by ISA)
⊕ faster miss handling

x86-64, ARM → hardware-managed: select victim based on policy encoded in hardware (no OS)
→ walk PTs in hardware to resolve addr. mapping
⊖ flexible in replacement policy, page table format

ASID = Address space identifier

Problem: vpn dependent on AS → clear TLB on AS switch

⇝ every entry gets ASID → map <vpn, ASID> to <pfn, protection>
→ no TLB flush on AS switch
⇝ less TLB misses (some entries still present from last run)

TLB reach: = TLB coverage
amount of memory accessible with TLB hits
TLB reach = (TLB size) × (Page size)

→ ideally working set fits in TLB
⇝ > increase Page size ⊕ fewer TLB entries ⊖ internal fragment.
> multiple page sizes ⊕ map larger memory areas to increase TLB, yet less internal fragment.
> Increase TLB size ⊖ expensive

Effective Access Time (EAT)
τ = time for associative lookup
μ = memory cycle time
α = TLB hit ratio

$EAT = \tau + 2\mu - \mu\alpha$
for linear PT without cache

Caching

Ideal memory: large, fast, nonvolatile, cheap
Real memory can't get all

Tape/Disk: large, slow, cheap, nonvolatile
SSD: not too large, not too cheap, nonvolatile
RAM: Fast, volatile, expensive
SRAM/Registers: very fast, volatile, very expensive

CPU-Cache:
- buffer memory for temporal + spatial locality
- low latency, high bandwidth
- reduction of main memory accesses, bus traffic
- asynchronous prefetch operation buffer

Cache Miss:
> compulsory miss: first reference, not cached before
> capacity miss: no free space -> evict victim to make room
> conflict miss: collision due to cache organization (doesn't occur in fully ass. cache

Harvard Architecture: seperate buffer for data and instructions
<> seperate bus + cache

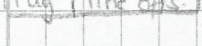
Write + replacement policies:

> Cache hit: write-through: main mem. always up-to-date, writes are slow
write-back: data written in cache only, update main mem. on eviction -> main mem. temp. inconsistent

> Cache miss: write-allocate: to-be-written data item is read from main mem. to cache -> then write according to policy
write-to-memory: modification performed only in memory

Cache Organization

Fully Associative: Cache-lines with fixed length, data identified by tag field -> search all tags for addr. tag



Direct Mapped: lines with fixed length, mapping from addr. to cache lines via hash-function, data identified by tag field

Set Associative: lines with fixed length, n lines = set -> map to set, inside set fully associative

Virtually Indexed, Virtually Tagged
index in cache, tag in cache in virtual addr. without translation

-> AMBIGUITY: identical VAs point to different PAs at different points in time

ALIAS: different VAs point to same physical memory location -> shared memory not allowed

=> context switch: invalidate cache
fork: copy whole AS of parent
exec: invalidat cache
exit: flush cache
brk/sbrk: no action on grow, cache invalid. on shrinking

Virtually Indexed, Physically Tagged (used as first-level cache)
index in cache from VA, tag in cache from PA
- no ambiguities
- no flush on context switch
- shared mem. / mem. mapped files: virtual starting adr. mapped to same cache line
- I/O: cache flush (same as VIVT)

Conflicts: data structures with adr. distance multiple of cache size mapped to same cache line

Properties:
- cache flush mostly avoided → fast context switch, interrupt-handling, syscalls
- delayed write-back after context-switch: performance gain

Physically Indexed, Physically Tagged
index from PA, tag from PA
⊕ transparent to processor, no performance-critical support needed
⊖ Page conflicts caused by random allocation of phys. mem.

Page Faults

caused by access to page, that is not present in main memory

Page Fault Handling:
(1) OS checks validity of access
(2) get empty frame
(3) load contents of page from disk into frame
(4) adapt page table
(5) set present bit
(6) restart faulting instruction

Latency:
$0 \leq p \leq 1{,}0$ page fault rate (every reference faults; no page faults)

Effective Access Time (EAT):

EAT = (1-p) × memory access + p × (page fault overhead + p.f. service time + restart overhead)

example: mem. access = 200 ns, p.f. service time = 8 ms } 1 of 1000 accesses causes p.f. EAT = 8,2 µs → slowdown by factor of 40

Challenges:
- pre-fetch surrounding pages? → seek times dominates ⇒ reading two blocks approx. as fast as reading one
- pre-zero pages? → no info leak btw. processes, need 0-filled pages for stack/heap/bss
- how to resume after fault? → OS needs context of fault (read/write, ...)

idempotent instructions: re-do load/store instructions
re-execute instructions that only access one addr.

<> complex instructions must be re-started too
-> problem: some CISC instructions difficult to restart
solution: touch all relevant pages before operation starts
keep modified data in registers -> no page faults
design ISA accordingly: complex operations execute partially, leave consistent state on p.f.

Memory-mapped files
- mapping disk block to a page in memory
- read: demand paging -> subsequent reads/writes treated as ordinary mem. access
- allows mapping for several processes -> allows sharing

Copy-on-write
parent and child process initially share same mem. pages
<> copy only if one process tries to modify it
<> more efficient process creation

Page Frame Allocation

> Global Allocation: all pages considered for replacement
- process can get anothers frames -> no protection from process that hogs all memory

> Local Allocation: only frames of faulting process considered
- isolates processes, how many frames does each process get?

↳ Equal Allocation: all processes get same amount of frames

Proportional Allocation: allocate according to process size

> Priority Allocation: proportional alloc., but using priority instead of size
-> on page fault: consider own ~~pages~~ or frames or frames of lower priority processes

Memory Locality: background storage slow -> goal: run near mem. speed, not slow storage speed

Pareto principle: 10% of memory gets 90% of references

Trashing: system is busy swapping pages in and out
<> on each swap, a page with soon referenced content is swapped out
<> low CPU utilization, processes wait for page fetching
=> OS thinks it needs higher degree of multiprogramming <>

caused by:
- no temporal locality in access pattern
- too many processes -> don't fit all in memory
<> degree of multiprogr. too high
- memory to small for a single process
- page replacement policy doesn't work

Working set (WS)
number of pages referenced in time Δ (Δ = ∞ : entire program)

D = Σ WS_i : total frame demand, D > m : Trashing

Ideally: replace page with reference furthest in the future (oracle)

Idea: predict future from past
or: sacrifice precision for speed (reference bit + timer)
↳ check history on every timer interrupt, if ≠ 0 page is in WS

› page fault rate
- too low: give frames to other processes
- too high: allocate more frames

Demand-Paging: transfer only faulting pages
⊕ less mem. needed per process → higher degree of multipr. possible
only transfer what is needed
⊖ initial page faults on task start
more I/O operations → overhead

Pre-Paging: speculative transfer at every page fault
⊕ improves disk I/O throughput
⊖ wastes I/O bandwidth if page never used
can destroy WS of other process in case of page stealing

Replacement Policies

Page Buffering: keep pool of free page frames (pre-cleaning) instead of demand-cleaning
→ daemon in background cleans, reclaims and scrubs pages for free pool
write back changes unmap zero out

FIFO: evict oldest fetched page

Belady's Anomaly: for every N ∈ ℕ of page frames constructing a reference string, that performs worse with N+1 frames possible

Oracle: optimal: replace page referenced furthest in the future
problem: cannot predict future :-

LRU = Least Recently Used: easy to understand, hard to implement
› cycle counter implementation
MMU writes CPU's time stamp to PTE on access
→ scan all PTEs on page fault to find oldest
⊕ cheap access if done in hardware
⊖ memory traffic for scanning

› stack implementation
doubly linked list of all page frames, referenced page to tail
⊕ replacement victim found in O(1)
⊖ needs to change 6 pointers on each access

Clock: = second chance replacement

- HMU sets reference bit in PTE
- keep all pages in circular FIFO list
- on page fault: scan pages in FIFO's order
  → ref. bit = 0 → victim found (hardware then resets ref. bit)
  ref. bit = 1 → set to 0, continue scanning
- Large memory: 2 arms, one clears ref. bit, other selects victim

Random: pick random victim → not that horrible in reality

Memory

Allocation → Dynamic Memory Allocation Problem: fragmentation

Allocate / free memory chunks of different sizes at random points in time
↳ allows data grow
- kernel uses it

↳ has huge impact on performance

Dynamic Memory Allocator
- keeps track of free memory parts
- no compaction → causes fragmentation
- cannot control request order / number

Bitmap divide mem. in units of fixed size
bitmap keeps track of units, 1 = allocated 0 = free
→ needs additional data structure to store allocation length (for free)

List
(1) use one list-node for each allocated area
↳ needs extra space for list, length already stored
or (2) one list-node for each unallocated area
↳ keep list in unallocated area, needs extra data structure for lengths
↳ search for free space with low overhead

Fragmentation: three factors needed
(1) different lifetimes
(2) different sizes
(3) no relocation (no compaction)
} all present in dynamic mem. allocators

Strategies

- **Best-Fit**: allocate smallest free block large enough
  - ↪ must searche entire list, unless ordered by size
  - ↪ free: coalesce adjacent blocks

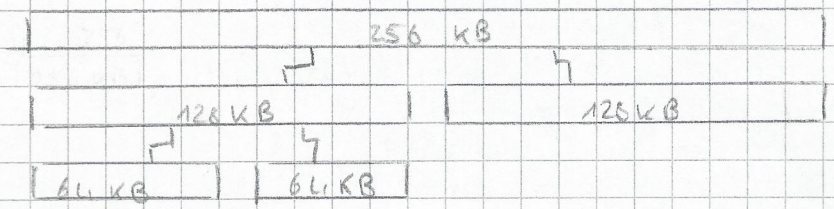
  Problem: SAWDUST
  small remainders, unusable

Worst-Fit: allocate largest free block
- ↪ also must search entire list
- ↪ less sawdust, but in reality worse fragm. than best-fit

- **First-Fit**: fragm. in both ↗ → at least minimize duration
  allocate first free space big enough
  - ↪ fast, but leftover holes of variable size

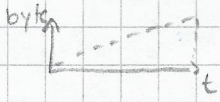
Next Fit: First Fit, but start searche from last allocated space

Buddy Allocator: Allocates memory in powers of 2 — used in Linux kernel to alloc phys. mem
- → round up request to next-higher power of 2
- → all chunks naturally aligned (starting adr. multiple of size)
- → split blocks until block of correct size available (buddies)
  - ↪ merge buddies if both free

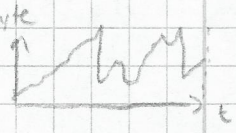


Allocation patterns most programs exhibit 1/2/3 of these patterns of alloc/free

Ramps: accumulate data monotonically over time
practical sense: no free



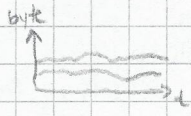
Peaks: allocate many objects, use briefly, then free all
- ↪ fragmentation!



Peak phase: alloc a lot, then free everything
- ↪ "arena allocation": only support free of everything

Arena = linked list of large memory chunks

Plateau: allocate many objects for a long time

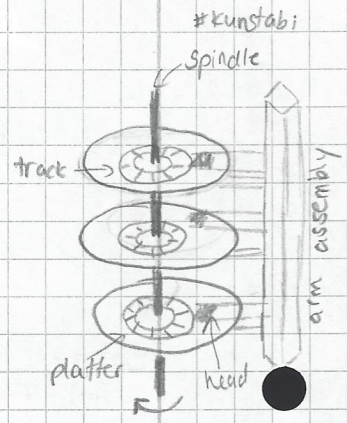


> **know patterns of real programs**:
- segregation = reduced fragmentation:
  allocated at same time ~ freed at same time
  different type ~ freed at different time

implementation observations: progr. allocate small number of diff. sizes
most allocations small (<10 words)

SLAB Allocator used in Linux kernel
Kernel often allocates / frees memory for few specific data objects of fixed size
- slab made up of multiple pages of contiguous physic. mem.
- cache consists of one / multiple slabs
  ↳ each cache stores only one kind of object (fixed size)

Secondary - Storage



Hard Disk Drive

- Stack of magnetic platters, rotate together on central spindle
- disk arms rotate around pivot, all move together
  ↳ contain heads for each recording surface
    ↳ heads read and write data

platters
- divided into concentric tracks
- stack of tracks of fixed radius: cylinder
- only one head active at a time
- access time: seektime (time to move heads to cylinder containing the sector)
  rotational delay (time waiting for disk to rotate sector to head)

seek:
(1) speedup (accelerate arm to max speed / halfway point) → dominatse short seeks
(2) coast (max speed, for long seeks)
(3) slowdown (stop arm near destination)
(4) settle (adjust head to actual desired track) → dominates very short seeks

settle longer for write than for read → if read wrong → reread
if write wrong → wrote on wrong track

Solid State Drive (no moving parts) FLASH MEMORY

- store data by storing charge
  ↳ lower power consumption + heat, no mechanical seek times

- limited number of overwrites
  ↳ flash translation layer (FLT) provides wear leveling
    ↳ repeated writes to logical block won't wear out phys. block
    ↳ impacts performance

- random writes expensive
- limited durability: charge wears out over time → data loss

NAND Flash
- higher density, fast erase + write, needs error correction (more internal errors)
- Block of 64/128 pages, divided into 2-4 planes
- must erase whole block before programming → write back all pages if something changed

Package → Die → Plane → Block → Page

Performance Optimization

spare block: keep a set of erased spare blocks
↳ choose block, erase old block later, write data to new block

data rate of disk << data rate of CPU/RAM

=> use multiple disks to parallelize disk I/O
↳ RAID

SLED = single large expensive disk

RAID redundant array of inexpensive disks
multiple disks
↳ more storage, parallelism, more reliable by storing redundant data

RAID 0: no redundancy
- decreased availability compared to SLED
- increased bandwidth to/from logical disk

Disk 0	1	2	3
0	1	2	3
4	5	6	7
8	9	10	11

RAID 1: mirrored

Disk 0	1	2	3
0	0	1	1
2	2	3	3
4	4	5	5

RAID 2: redundancy through Hamming code
overkill, rarely implemented

RAID 3: byte-interleaved parity

RAID 4: block-interleaved parity
$P(0,\dots,3) = block_0 \otimes block_1 \otimes block_2 \otimes block_3$
↳ updates need 2 reads + 2 writes!
=> parity disk is bottleneck

0	1	2	3	4
0	1	2	3	P(0-3)
4	5	6	7	P(4-7)
8	9	10	11	P(8-11)

RAID 5: block-level distributed parity
=> no bottleneck problem

0	1	2	3	P_0
4	5	6	P_1	7
8	9	P_2	10	11

Tertiary Storage Devices: defined by low cost
generally: removable media

Optical Disks (DVD/CD)
- rw: modify over and over
- worm = write once, read many times

Magnetic Tapes
compared to disk: ⊕ less expensive, holds more data
⊖ random access much slower

File Systems

Job of OS: low-level device control (initiate disk reads, ...)
high-level abstractions (read file)

File = collection of related info
- has set of attributes = meta data (name, type, size, protection, ...)

Attributes:

Protection	Hidden flag	Lock flags
Password	System flag	Record length
Creator	Archive flag	Key position
Owner	ASCII / binary flag	Key length
Read-only-Flag	Random access flag	Time of last access
Creation time	Temporary flag	-"- change
Current size	Maximum size	

File Management
- provide convenient naming scheme, uniform I/O support for different storage devices, standardized set of I/O interface functions, I/O support/access for multiple user
- minimize loss/corruption of data
- provide acceptable performance

File Access

Open files: meta data needed:
- » file pointer: points to last read/write location
- » access rights: access mode info
- » file-open count: counter of number of times file is opened
- » disk location: cache of data access info

» today: random access: read bytes/records in any order
↳ essential for db systems

Plain file: sequence of bytes, typically located on a disk
↳ possible to access random byte within unstructured file, if file pointer is positioned appropriatly

Directories: = node in FS owned by an subject (eg root) containing info about files of the FS

. = current dir
.. = parent dir

↳ hierarchical file system structure

Unix: single, uniform FS tree, different FSs mounted into one tree

Two-level Directory: seperate dirs for each user

bound to partition, each new HL increases inode's link counter, only really removed if = 0 (decrease link counter on rm)

Hard Links: map a name directly to an inode
↳ impossible to distinguish btw. file and hard link to it

Symbolic links: files, that contain path to other file, points nowhere on rm

Access Rights:
- none: user might not know of existence of file, not allowed to read dir containing the file
- knowledge: user can determine files existence + ownership
- execution: user can load + execute program, no copy allowed
- Reading: user can read file, incl. copying, execution
- Appending: user can only add data to file, can't modify / delete
- Updating: user can modify, delete, add, creating, rewriting, removing all
- changing protection: user can change access rights for other users
- Deletion: user can delete file
- Owner: all previous rights, can create / modify user group rights

UNIX Files: 1 = execute, 2 = write, 4 = read
Directories 1 = dir. open, 2 = create / read / rename files
4 = list files

Disk Structure
- partition: ~~subdir~~ part of subdivided disk
- volume = disk / partition with File System
  ↳ tracks FS's info in device directory / volume table of contents

- Sector 0 = MBR: boot info, disk partition info
- sector 0 of partition: volume boot record

File Control Block
permissions, dates (create, access, write), owner, group, ACL, size, file data blocks or pointers to file data blocks

Virtual File System
provides same syscall interface to be used for different types of FSs

Implementing Files
- FS tracks meta data: belonging blocks, order of blocks, free blocks
- FS must find corresponding blocks given a logical region of a file
  ↳ metadata from FAT, directory, inode

Allocation Policies

- Preallocation: - max. size must be known at time of creation
  ↳ difficult to reliably estimate size
  ↳ users tend to overestimate
- Dynamic allocation: allocate in pieces as needed

Contiguous Allocation:

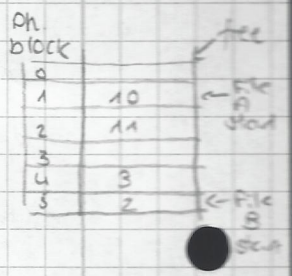
Array of N contiguous logical blocks reserved per file (to be created)
↳ minimum meta data per entry (starting block addr. + N)
→ good N? what about needing more than N?

→ Scattered disk ⇝ periodic compaction

Chained Allocation:

per file linked list of logical file blocks
↳ each block contains pointer to next block, last block contains NIL pointer
FAT/directory contains addr. of first
→ no external fragmentation
→ sequential read good, random very bad (blocks scattered across disk)

Linked list with RAM
blocks only store file data, FAT in RAM keeps track of list



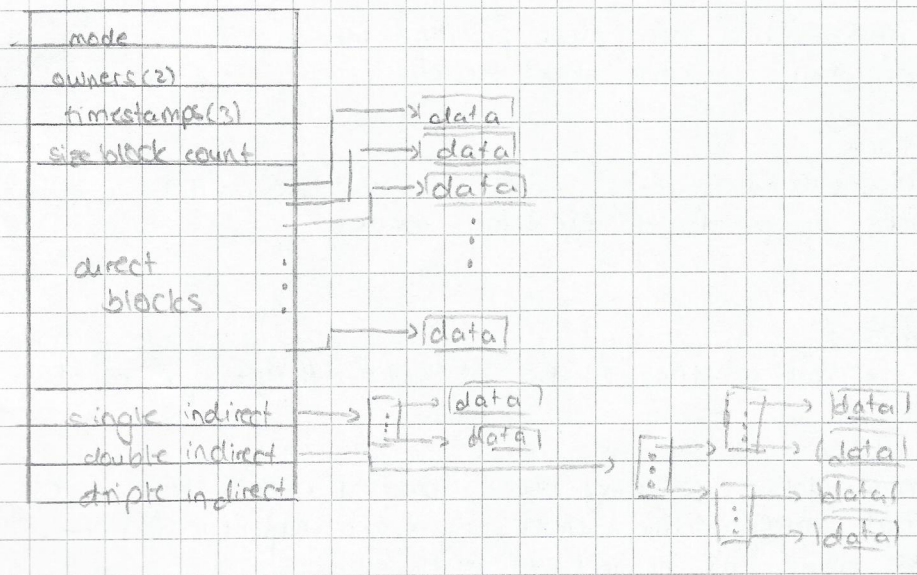
Indexed Allocation:

- FAT contains one-level index table per file
- index has one entry for each allocated file block
- FAT contains block number for the index

Block sized: only block numbers stored in index
⊕ eliminates external fragmentation

Variable sized: store block start number + length
⊕ improves contiguity
reduces index size

Multi-level index: use more than one index block for larger files



Implementing Directories
UNIX: entry in dir only refers to i-node

Linear Dir. Lookup
- for big directories not efficient
- space efficient with compaction
- variable file names: fragmentation

Hashing a Dir. Lookup
- hashing file name to inode
- filename + meta data variable sized

⊕ fast lookup ⊖ not as efficient as trees for very large dirs

Tree structure
- sort files by name
- store dir entries as tree

⊕ efficient for large # of files per dir
⊖ complex, not efficient for small # of files, more space

UNIX File
- opening file creates file descriptor, used as index in per-process table of open files
  -> entry points to system-wide file table
  -> points to buffered inode table
  -> points to data

BSD Fast File System
- block size: 4KB / 8KB, can be chopped into 2, 4, 8 fragments
- bitmap instead of free list
- data + meta data in same cylinder group
- items of same dir. in same / nearby cylinder groups

Directory
- dir. entry contains at least: length of entry, file name, inode number
- dir. contains at least: .. = parent dir, . = link to itself

UNIX Directories
- multiple dir. entries may point to same inode (hard link) (within same FS)
- pathnames to identify files

Mounting:
files from currently inaccessible FS's get accessible
↳ create mount dir in existing FS and set this as root for the other FS

UNIX Inode
- Mode = file type, protection bits, setuid, setgid bits
- NLinks = Number of dir. entries pointing to this inode
- UID = id of file owner
- GID = id of group of file owner
- size
- addr = addr. of first 10 blocks, then 3 indirect blocks
- Gen = generation number (incr. on reuse of inode)
- Atime = last access
- Mtime = last modify
- ctime = last change

Buffering:
- buffer disk blocks in main mem., access via hash table
- blocks with same hash value chained together
- replacement policy: LRU
- free buffer: double linked list

UNIX Buffer Cache
modern systems: unified buffer cache

⊕ reduces disk traffic
hit rates up to 90%.

⊖ - write-behind policy might lead to
- data losses in case of crash, incosistend state of FS
↳ rebooting system requires checking all dirs and files
- always two copies involved (disk → buffer → AS)

Linux Ext2fs FS ≈ BSD FFS
main difference: disk alloc policies
↳ FFS: 8kB blocks + subdivision
Ext2fs: default block size 1kB
- places logically adjacent blocks into phys. adj. blocks on disk
↳ I/O request for several blocks = single operation

Journaling FS
- record each update as transaction → written to log (once written: committed)
- on crash: remaining transactions in log must be performed

Log-structured FS
- disk as circular buffer
- all updates (incl. inodes, metadata, data) to log

I/O Systems

Device Management Objectives
- Abstraction from details of phys. devices
- Uniform Naming, independent from HW details
- Serialization of I/O ops by concurrent apps
- protection agains unauthorized access
- buffering, if data cannot be stored in final destination
- error handling of device errors
- virtualizing by multiplexing

Block devices (disk drives)
- commands incl. read, write, seek

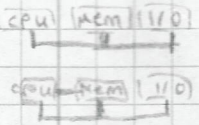
Character devices (keyboards, mice, serial ports)
- commands incl. get, put

Network devices
UNIX and Windows include socket interface
↳ seperates network protocol from network operation
↳ includes 'select' functionality

I/O Hardware
- Common components: Controller, Port, Bus
- Devices have addr. used by: direct I/O instructions, mem.-mapped I/O
- Device addr. typically point to: status-, control-, data-in-, data-out-register

Memory-mapped I/O different options:
(a) seperate I/O AS and memory AS
(b) memory-mapped I/O → 1 common AS
(c) Hybrid (pentium) → part of I/O space in mem., part in extra AS

two bus options:
(a) single bus architecture: one bus for mem. and I/O requests
(b) dual-bus memory architecture: seperate bus for memory



I/O-Management

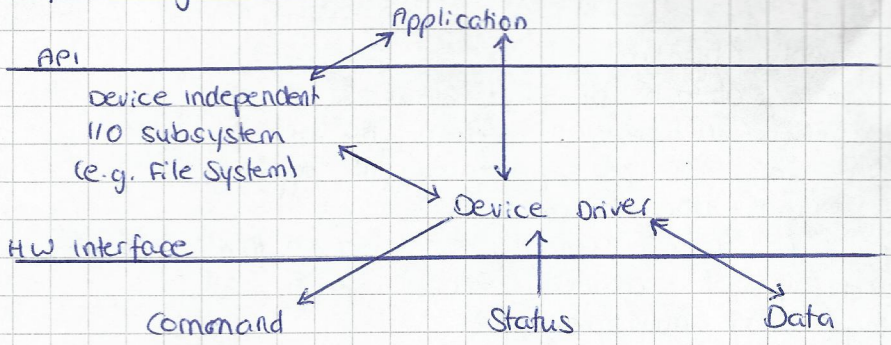
Programmed I/O: thread busy waiting for I/O to complete
↳ processor time wasted
kernel thread polling state of I/O device
→ command-redy / busy / Error

Interrupt-driven I/O: I/O-command issued, processor continues executing instructions, I/O-device sends interrupt once done

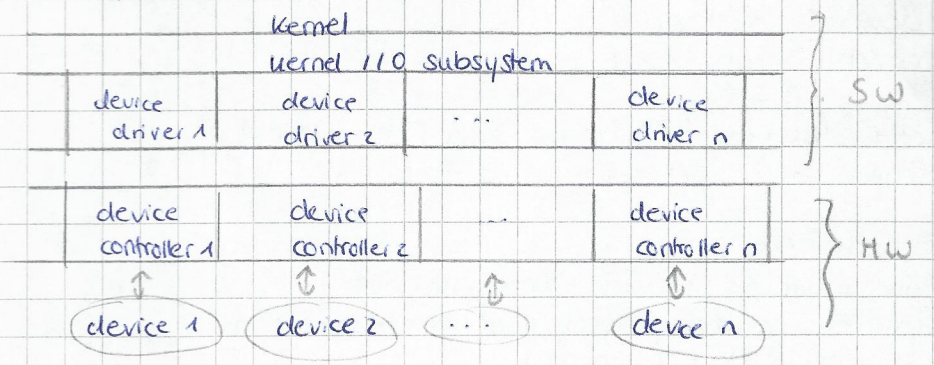
Direct Memory Access: controls exchange of data between main mem. and I/O device, processor interrupted after transfer is complete
bypass CPU to transfer data directly btw. I/O device and mem.

Interrupt handling:
(1) save registers, not saved by HW-interrupt mechanism → (2) set up context (AS) for interrupt service procedure (typically no switch) → (3) set up stack for interrupt procedure (usually runs on kernel stack of current process/KLT, handler cannot block)
→ (4) acknowledge/mask interrupt controller (re-enable other interrupts) → (5) run procedure
→ (6) sometimes: wake up higher priority process/KLT → (7) load new/original registers
→ (8) return from interrupt, start running new/original process

I/O System organization



Kernel I/O structure



Kernel I/O subsystem

- Scheduling
- Buffering - store data in mem. while transfering btw. devices
- Error handling
- Protection (I/O performed via syscalls)
- Spooling - hold output for a device, if device can serve only one request at a time
- Device reservation - provides exclusive access to a device
  - syscalls for alloc/free

Device-Independent I/O-Software

abstract commonalities of drivers of similar classes
- e.g. buffer / buffer (cache) management
  allocating and releasing dedicated devices
  error reporting to upper levels
- uniform device interface for kernel code
  uniform kernel interface for device code

Device Driver

classified into categories:
- block devices
- character devices (stream of data)

-> OS defines standard interface

- reentrant
- after command arrives: device either completes immediatly + returns to caller
  or processes request + caller blocks until interrupt signal