CSC369

①

→ Virtualization

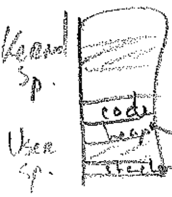
- CPU → time sharing
- Memory (RAM) → space sharing

OS implements virt → high level: policy] — keeping separate
→ low level: mechanism] improves modularity

→ Scheduling and threads

→ Issues: Concurrency (coordination) + Persistence (main mem is lost during crash ⇒ need to keep crit. data)

→ Limited Direct execution (no overhead): set up cpu s.t. it just executes the prog. somewhere



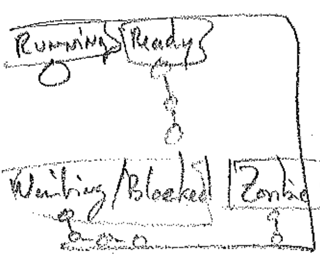
malloc'd DS
local vars, ret. vals

Limitations ⇒ { · have to restrict operation
· need to regain control → interrupts

→ process → has virt. CPU & virt mem
address space

→ multi-threaded process: · multiple pts of exec.
· all threads share the same
shared objs
have ≠ PC !
T2: stack (not shared) → local vars
T1: stack
shared stack → global vars, static obj.

→ process control block (: OS's DS for processes)



→ PID + process state + process priority
→ add. spc. + PC + SP
→ Register save area (to store info in cpu regs for proc while context switch)
→ resource use info
list of open files ...

→ IDTR : int. descrip. table reg.
→ syscall → int handler handler

HW | CPU | SW
IRQ | excep/trap | Syscall
int. req. (from I/O device)
Save regs etc...

OS is event driven

→ Interrupts

→ Mechanism! (vs policy)

- HW int: signal I/O event
- SW int: errors and syscalls
  ⇐ traps = exceptions

→ Mode Bit

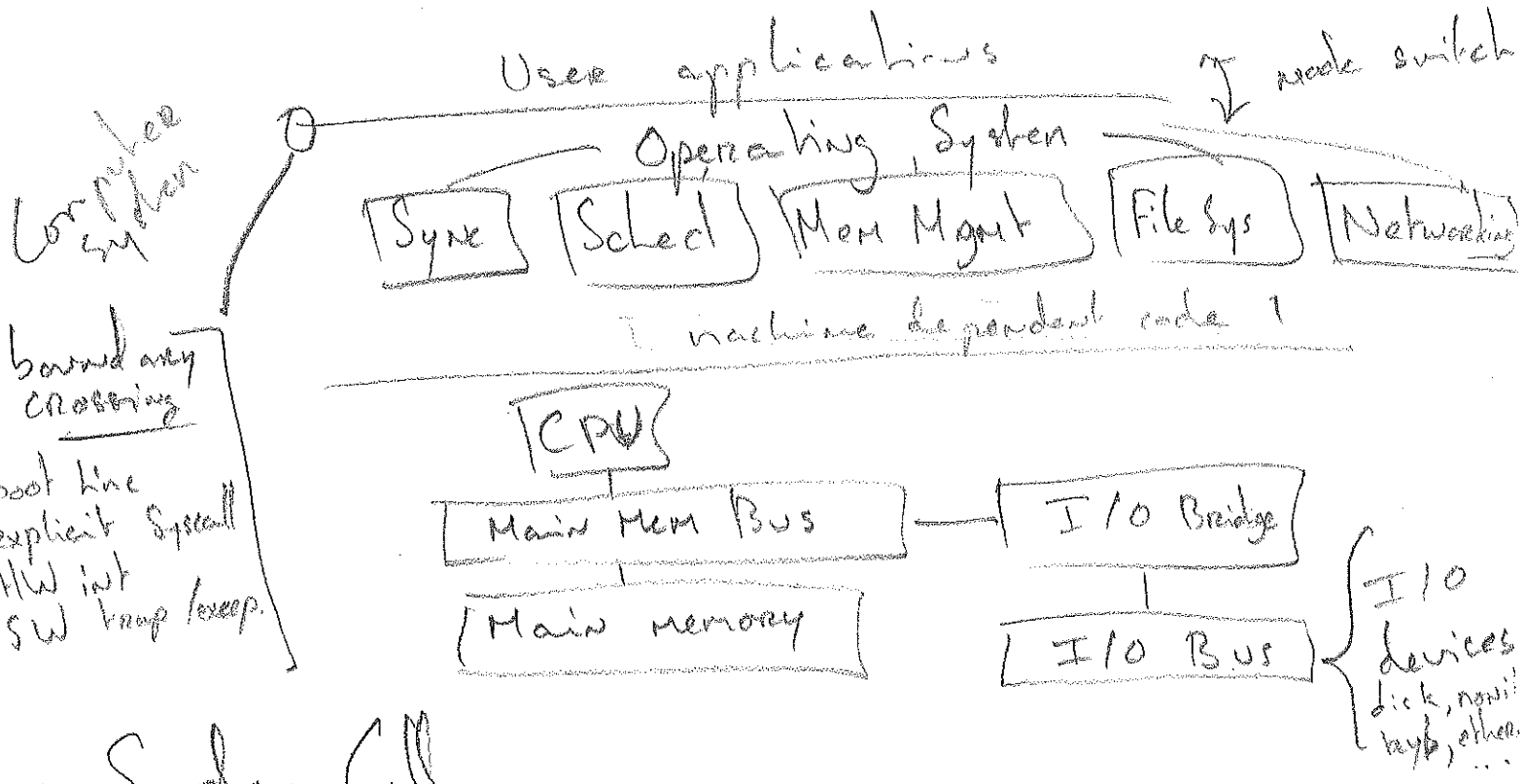
hardware support for protec (required by scheduling)

→ Bootstrapping

RAM = volatile, ROM = non-volatile
(memory)

→ ⇒ firmware → software that ctrl some h w features. doesn't require (req. updat.

BIOS (basic I/O sys): 1st prog to RUN after power is supplied
→ HW is in sys mode
→ initialize OS DS
→ create 1st proc. (init)
→ mode switch → user



→ System Call

cf 209: when user prog does a syscall, it passes args to the kernel by giving its ptrs to buffers, which have to be copied on kernel stack.

→ Context Switch

→ switch CPU to new prog by saving state of old, loading that of new
→ time = pure overhead! → policies should be parsimonious.

5. Synchronisation

- critical section pb: two threads accessing same shared resource must be controlled
- order of thread execution

→ race condi°: outcome depends on which does smthg 1st.

→ mutual exclusion: avoid
→ progress: if CS is available, do smthg
→ bounded waiting: avoid starva° by setting lim. on # times all threads can enter CS, so everyone eventually does.

Soln° Peterson's alg → 2 threads share turn, flag[2]={F,F}
→ interested thread sets its flag
→ spin wait while turn==me && flag[i]==T
→ if 2 t. enter at once and chg turn to their ids, one of them will overwrite the other.

(Lamport's) Bakery alg → give tickets, cut ties by PID.

Abstract° of CS → locks
→ condition vars
→ semaphores
→ monitors
→ messages
} same power

HW help: disable interrupts (context switches) during CS access

HW sup.
disable int during

- spinlock: acquire lock by busy waiting: while(testAndSet(lock)) → consumes CPU cycles until lock is available
  atomic instruction: blocked if l=T, ret true. unblocked if l=F, l=T, ret false
- release lock

- semaphore: ADT w/ { int var, wait (aka P, decrement) → var-- & block, signal (aka V, increment) → var++ & unblock, wait channel }

- binary semaphore : single access to CS
- counting ——— : multiple threads (bounded by counter) have access to CS

⚠ make sure wait and signal don't happen together
- ↳ low-level primitives
- ↳ uniprocessor → disable interrupts
- ↳ multi ——— → hw instruc. e.g. spinlocks.

Readers/Writers pb w/ semaphores: lock reads w/ mutex, lock writes w/ w_or_r, only 1st reader blocks on W_or_R, next block on mutex.

→ limits of wait & signal: only single condition (if thrown me)

condition var: { wait channel
- cv_wait ——— rel. lock, sleep, acquire lock
- cv_signal ——— wake waiting thread
- cv_broadcast ——— wake all threads

- monitor: ADT { data only accessible through ops
  - → a process enters the monitor through op.
  - → single process in monitor (active) } ▷ e.g. use semaphores

⚠ How to ensure single active proc?

(▷ signal policy)
- → Hoare monitors: when P2 signals P1, context switch to P1, cond. P1 was waiting for always T. ⇒ need Q for signaler
- → Mesa monitors: when ———, P1 sent to ready Q and P2 continues. ⇒ need to recheck P2's wait cond.

→ Deadlock & Starvation

Deadlock: set of threads S s.t. ∀t∈S t is waiting for an event that can only be performed by a t'∈S.

Starv: thread that is postponed indefinitely.

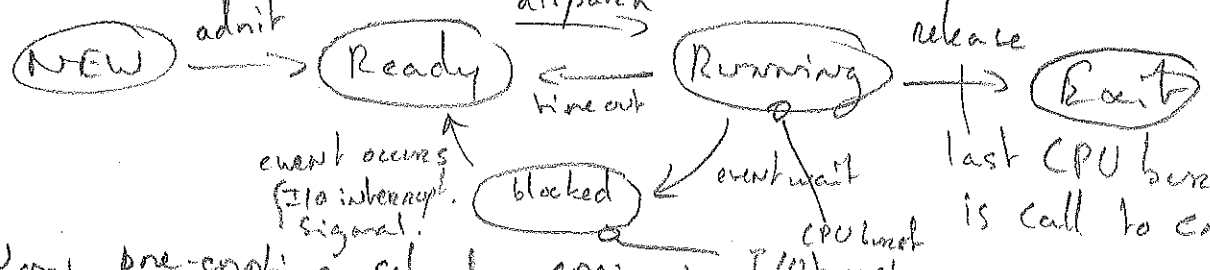
Scheduling

Review: process is instance of (prog in execu°) - synch to coordinate resource sharing.
- thread(s) are part of a process

Threads alternate between computa° (CPU) and I/O.
- CPU-bound: CPU >> I/O
- I/O ——: I/O >> CPU

CPU bursts
I/O ——

Lifecycle:

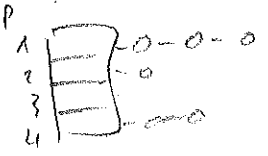


- Non pre-emptive sched: CPU is kept until thread blocks/exits
- Pre-emptive sched: CPU can be given to other thread before

Algs
- FCFS (1st come 1st serv.) ← non-preempting
- SJF/SPN (Shortest Job 1st / Proc Next)
  - choose thread w/ shortest expected runtime
- Round Robin (RR)
  - very simple implementa° / concept
  - circular list of processes with associated (identical, i.e. no priority) time slice (quantum)
- Priority Scheduling

policy: job w/ ↑ priority runs

issues: priority inversion: low p task holds resource high p needs
- starvation: low p never runs



- Multilevel Queue Scheduling MLQ
  - extra level of scheduling: multiple ready qs w/ own sched alg. for each
- Feedback Scheduling
  - decision depends on history: e.g. aging
    - combine w/ MLQ to move threads between qs => MLFQ

→ Fair Share Scheduling
- → grp threads by owners, give appropriate CPU sharing
  - → priority of a thread depends on its own priority and past history of grp.

→ Unix Scheduling
- → MLFQ w/ RR on each Q
- → priority of job j: $P_j(i) = base_j + \frac{CPU_j(i-1)}{2} + nice_j$

$P_j(i)$: pty of j at beg. of interval i; $base_j$: base pty of j; $CPU_j(i-1)$: CPU Utilizn of j in interval i; nice: user controlled adjust factor

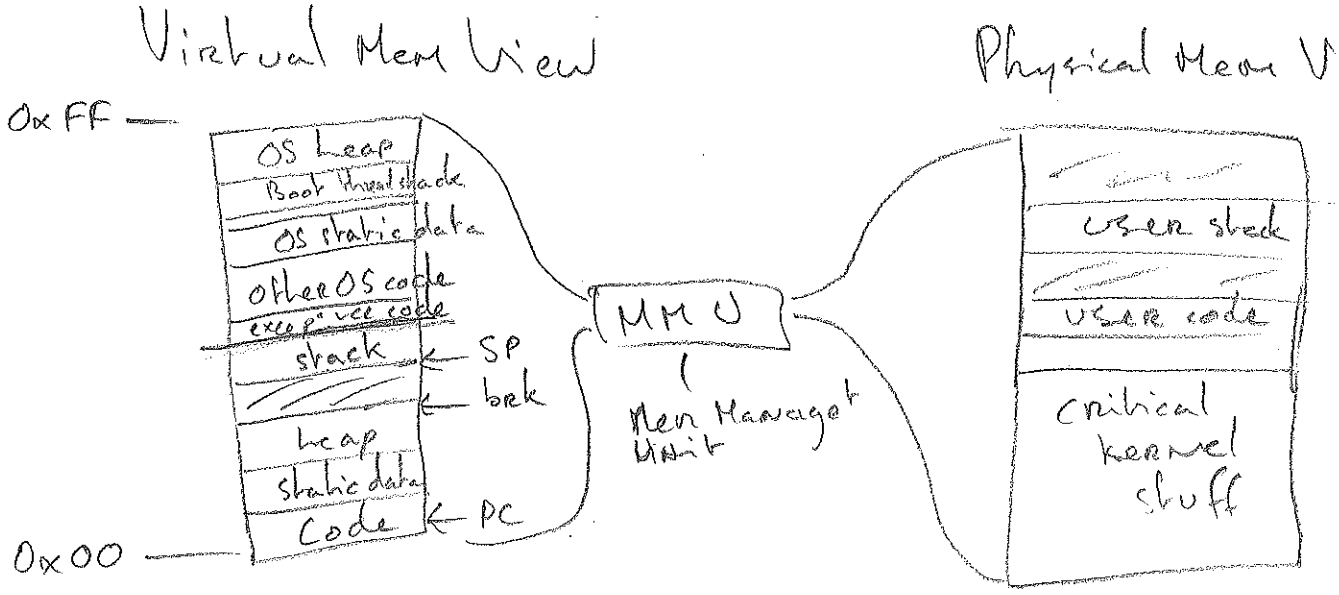
$CPU_j(i) = \frac{U_j(i)}{2} + \frac{CPU_j(i-1)}{2}$

→ Memory Manag't

Requirements:
- Relocation
- protection → requires HW support (node bit)
- sharing
- logical org → mapping between ϕ mem and code modules
- physical org → memory vs disk ⇒ hierarchy ⇒ manage flow between levels

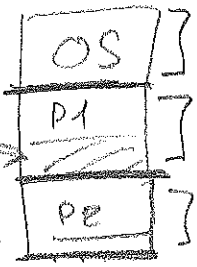
→ Address binding
- data in code (symbolic) needs to be binded to ϕ add.
  - → compile-time binding: absolute code (binary contains real add.) ⇒ no possible relocn
  - → load-time ——— : static relocation (add.)
  - → execution: ○ compiler maps symbolic add. to logical, relocatable
    - ○ linker take the whole collec'n of obj. files (.o) to map to logical, absolute add.

→ RUN-time binding: dynamic relocation



→ Fixed partitioning of Ø Mem
- → internal fragmenta° — unused
- → overlays (progs larger than partition → programmer has to deal @).

solvo: swapping: swap out often waiting processes

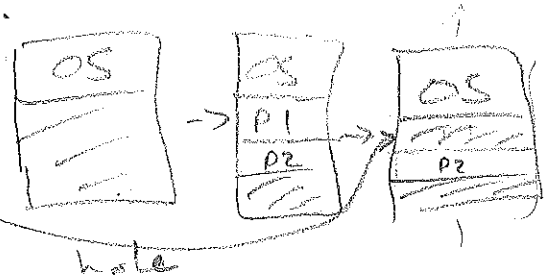


→ Dynamic
- → external fragmenta°

defragmenta°
- → compac° : requires process to be relocatable.

sol'n:
- post frag
- pre frag

→ placement alg:
- 1st fit: 1st block large enough from start.
- next fit: from prev. search
- worst fit: choose largest block
- best fit: choose block w/ size closest to needed.
- quick fit: keep multiple lists of free blocks for common sizes



o with run-time binding:
- → process instruc° refer to relative addresses.
  - ⇒ hw support: base reg + limit reg are loaded at runtime
    - on memory ref. instruction, add base to relative addr to get proper Ø addr. and compare to check for illegal addr. excep° ⇒ if(A<B || A≥B+L), then trap

→ Paging

— divide { phys mem into equal fixed size page frames.
virt mem of proc into virt pages
⇒ no ext. frag, int frag is limited

OS support: page table as OS DS
: per process
: map page to frame
— virt addr. is page# + page offset.
page# = vadd / pagesize
— offset = vadd mod pagesize

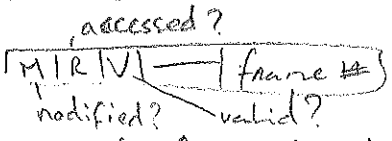
HW support: page table base register PTBR
on each mem ref MMU translates page# → frame#
and adds offset

e.g. 16bit & 1024 bytes page size ⇒ 

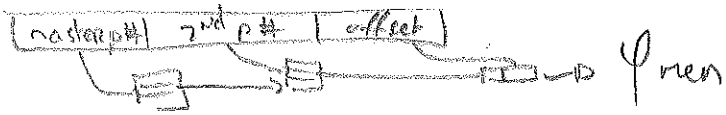
→ need 10 bits to address 1024 bytes.
→ leftover bits determine max # of addressable pages

32bit & 4096 bytes page size ⇒ 

→ Page table: array of page table entries PTE
~ 0 >> 13 → page size (12) + only half of mem is user ⇒ +1 = 13

→ PTE: one per page 

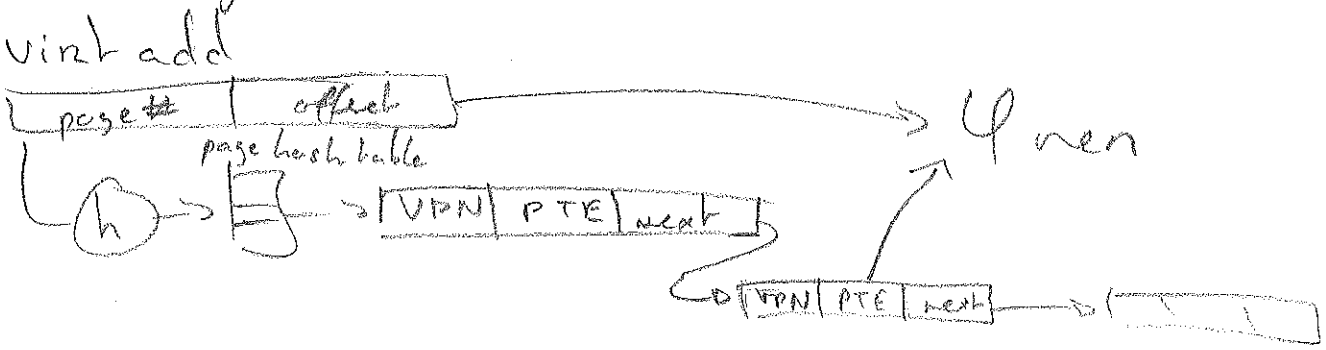
— limits: exponential size required for PT!
64bit ⇒ 16 PB!

— Hierarchical paging
2-level paging: virt add


32 bit → 4k pages, 4bytes PTE ⇒ we want Masters PageTable to fit in frame (4k) →
4k/4 = 1k ⇒ 10bits
since 12 for offset ⇒ 10 bits for 2nd PT (nice coincidence)

64 bit → still a lot of overhead

→ Hashed Page Tables



→ Paging limits

mem ref overhead ⇒ hw cache! ← part of MMU
→ fully associative cache of recent
→ Translation Lookaside Buffer TLB

TLB hits > 99% !

TLB misses (faults) → need policy to choose which PTE to evict from TLB to accomodate new

TLB manage → HW loaded (MMU)
OS maintains PTBR in main mem, HW accesses it directly
→ SW loaded
TLB miss → trap to OS
→ OS syncs TLB ⇔ PTBR (chg protect bit, etc..)

→ Paged virtual mem

swapping pages mem ⟷ disk ← swap file — demand paging

→ Locality

temporal loc — recently accessed locs are likely to be accessed again
spatial loc — locs near

→ Policies for Mem Manag°t

- Fetch policy : when to fetch page → demand paging vs prepaging
- Placement ——— where to put pages
- Replacement ——— what to evict

UNIX

fork returns twice → copy of add. space of parent

exec returns → pb Δ

exit ⟶ zombie while parent waits

context switch → add space changed

→ save old, load new

PCB: process { add space: code + data, stack (execu° space)
PC / registers
OS resources: files network ports
kernel thread and stack – OS ds
PID

I/O, resource use (for sched.), proc state, PC
CPU regs (for saving during switch), proc priority, page tables

multi process < multithread (kernel) < multithread (user)

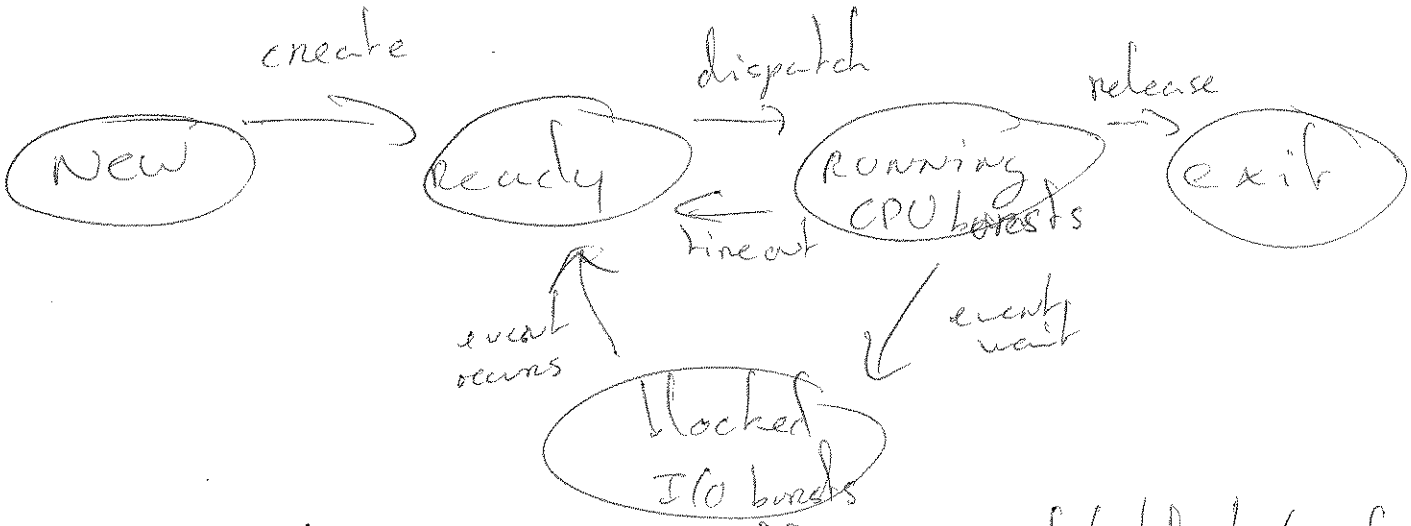
race cond.: 2 proc accessing same resource with possibility of multiple outcomes depending on "scheduling"

↳ "critical section"

↳ mutual exclus
↳ progress: only threads involved in CS can influence who gets it
↳ bounded waiting (no starvation): given a waiting proc, should be a bound on # of other procs able to access CS

pbs w/ machine instruc° for CS

→ busy waiting
→ starvaᵢ°
→ deadlock — priority inversion



Now pre-emptive → proc yields cpu until block/exit
→ can be (context) switched out

batch
interactive

---

priority inversion (vs) starvaᵢ°

↘ priority yields
makes ↑ priority wait

↗ priority always executed
⇒ ↓ priority status

MLFQ → multiple queues, each proc goes on
Q dep. on b, feedback allows chg.

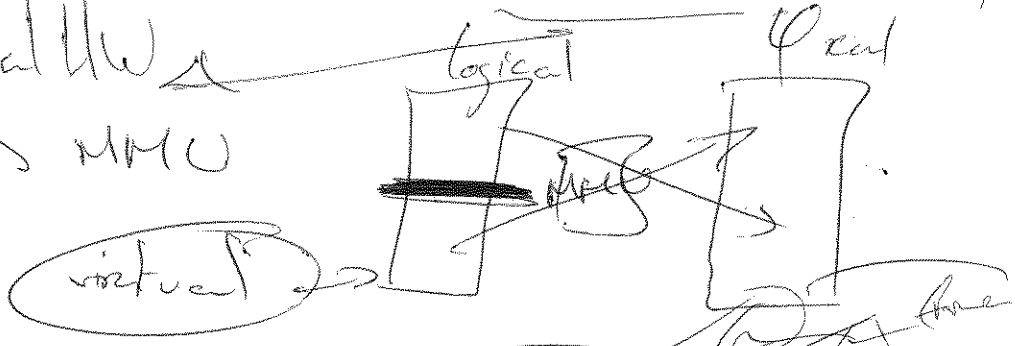
5 reqs of MM

8

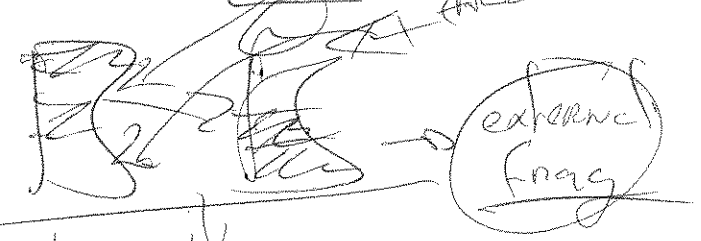
Virtual mem
- reloca° — modularity
- protec° — CS
- sharing — control
- logical org — transla°
- physical org — Regs / Caches / Ren / Disk

→ add. bindings → load-time → inherent frag
→ exec-time → dynamic

special HW
⇒ MMU



Dynamic Partitioning



→ compaction → free requires relocatability

Paging — ren divided into page frames,
proc addressed logically with pages
→ PTBR: array of PTEs indexed by VPN (per proc)

→ Page tables
→ Hierarchical PT
→ Hashed PT
→ Inverted PT → use hashing to reduce search time

Translation lookaside buffer

→ TLB ∈ MMU

→ 99% hits in practice → locality
→ TLB miss
↳ HW or SW inpf.

TLB manager: consistency of prof bits PTBR ↔ TLB

Page schemes → locality (temporal, spatial)

HW: needs → MMU, TLB, PT...

SW: policies → fetch, placement, replacement

- Demand paging: evict when full (no need to evict clean but need to know loc on disk)
- Prepaging: fetch more exploiting locality

Replacement algs

Belady — ~~beg~~ evicting page that won't be used for the longest period is opt. "
↳ Belady's alg: opt — for benchmark

→ FIFO: suffers from Belady's anomaly:
more space ⇒ more hits

→ LRU: last recently used — good guess
- exact: $!
- approximate: counter, shift down instead

→ 2nd chance: if ref bit == 0, evict otherwise move to next page frame

Page Buffering

- pool of free pages
  - frames on free list hold prev. cont and can be rescued if referenced before reallocation

→ local repl. every proc has fixed # pages

→ global repl. dynamically grows per proc

- Working Set = 122 {refs} Δ t
- Page Fault Freq. → ~~approximation~~ ad-hoc
  → ↑ or ↓ according to faults

Overcommitted to paging ⇒ thrashing == more time fetch/relo
than making CPU progress

Redux

CPU instr: read → TLB lookup → MMU translates to φ

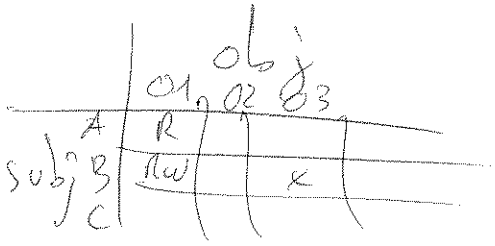
page faults { → PTE bit
→ No page → ↓ page not allocated → seg fault
→ ↓ page not in φ → allocate + fetch

Segment BS ?!

File Sys

Hard links → Acyclic Graphs (keep)

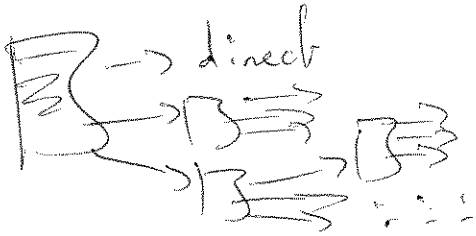
Access Control Lists



→ use groups (UNIX) to avoid ACL ↗↗

- FS implementation : 4k blocks, root in master block, free bitmap
- dir ⟵ : linear lists, hash table
- disk layout : contiguous alloc, linked (ptr to next block), indexed alloc

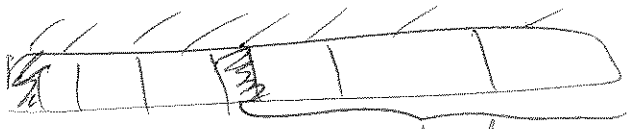
↳ UNIX inodes : stores metadata
: indexes blocks used by file/dir

is 

Exploit locality ⇒ File Buffer Cache (for writes, flush periodically)
competes w/ VM
↳ read ahead for reads

Disk I/O ⇒ messy (bad blocks, missed seeks...)

Original UNIX 

cylinder grps 

Deadlock

Nec + Suf:
- Nec: Mutex, Hold and wait, Non-preemption, circular wait

- prevention of these is hard
- avoidance: Banker's alg

- Recovery
  - → Ostrich alg
    - → finite res → node OS == virt.
    - ⇒ So only deadlocks are on locks

- write ahead logging: log transaction, checkpoint
  - writes log → data
  - updates → log
- set of noncon flicting ~~sets of~~ series of transac°
  - → conflict serializability: can reschedule w/out conflict
    - always
    - in c-s set
  - timeout for lost messages, protocols for duplicates

Livelock → continuously retrying same ops prevents prog -

---

interrupt

exception → syscall
→ illegal

Security

- confidentiality : ∅ info released
- integrity : ∅ info modified
- availability : provide access to legitimate users
- authentiality : establishing

Trap Doors — entry point exploit
Logic Bombs — legitimate destructive code
Trojan Horses — concealed malware that when run can do sthg like
Viruses — replicating its code in progs
Worms — network exploit

→ Buffer Over flows : fill buffer until can override return addr to point to exploit code

→ Network Attacks : passive → eavesdropping → encrypt
→ traffic analysis → obfuscate patterns

active → replay : capture to reuse
→ modify messages
→ masquerade
→ denial of service : flooding

→ Login Spoofing (similar to fishing)

Sol° ....
Transport Layer Security → Certification Authority
— handshake protocol

Sync : . mutex (avoid race cond) CSC 369
(CS) . progress → do smthg if CS is available
. bounded waiting → avoid starvation by lim # procs.

Peterson's alg: flag [2] = {F,F}, set T if interested
Bakery's alg: give tickets, cut ties by PID
locks → Test and Set atomic ⇒ busy waiting (HW disable interrupts)
cond. vars → var, wait, signal, wc → only if not binary; cv_broadcast
semaphores (same power)
monitors → data only accessible through ops
single proc active → Hoare: P2 → P1 wakes and comes
Mesa: P2 → P1 sent to ready and P2 goes

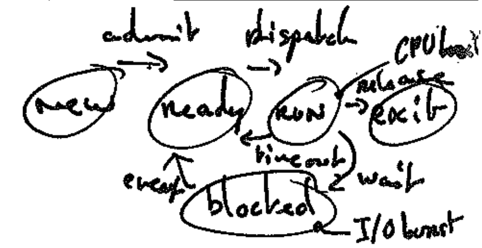
Sched: . pre-emptive = kick out stuff
→ FCFS, SJF, RR, Priority
→ time quant per proc
. Hierarchical + feedback ⇒ MLFQ
. Fair share

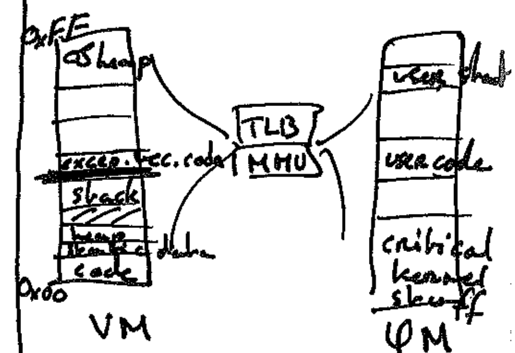
UNIX: RR on each Q of MLFQ

Mem: . Relocation
. protection → mode bit (HW support)
. sharing
. logical org → virt mem ↔ φ mem
. physical org → cache → Mem → Disks.
→ Add. binding: compile-time: absolute code
load-time: static reloc.
run-time: dynamic reloc.
. Fixed partitioning φM: internal frag
. Dynamic partitioning: ext. frag → post frag: compac.
pre frag: First Fit → 1st block fits
Next Fit → FF from prev. search
Worst fit → biggest block
Best fit → closest to size
Quick fit → multiple lists of common size blocks
. with run-time binding:
process instr. refers to relative add.
HW support: base reg + limit reg
. Paging:

— paged out —

Disk I/O: disk sched. FCFS – good for low load
SSTF – shortest seek time first
SCAN / C-SCAN – request in same dir (full disk)
LOOK / C-LOOK – same but until last req.





File Sys: tree vs acyclic g. with hard links
FS impl: free bitmap, root in MB
dirs: linear, hashed lists
disk layout: contig., linked, indexed
metadata → inodes

Exploit locality:
{ write ahead
{ read ahead
File Buffer Cache

Deadlock
{ Mutex
{ Hold & wait
{ Non-preemption
{ circular wait
prevention? no
avoidance? Banker's alg
→ procs declare max res.
→ not given if possibility of unsafe state
→ safe state = at least one deadlock-free schedule

Security
. confidentiality: ∅ info release
. integrity: ∅ info mod
. availability: provide access
. authenticity: legitimate access

conflict serializability – conflicts eq.
set of transac. s.t. ∃ sched. that is serial (by switching non-conflicts)
live lock – processes waiting for others locked in a relentless pointless task

Close

university of toronto
department of
computer science

File name MarkingScheme_A4 Download

File size 6 KB

next time bring earplugs for final exam!