Virtualization:

- CPU - no line sharing
- Memory (RAM) - no space sharing

OS implements with a high level policy - keeping separate
- low level mechanism improves modularity

Scheduling and Threads

Issues: Concurrency (cooperative) + Persistence (main ram is lost during)

- Limited direct exec. (no overhead): set up cpu s.t. it just executes the prog.
- Limitations: no have to restrict open...
- need to keep track of progs.
- no use of LBA to register control
- - - interrupts

- a process has virt. CPU & virt mem

- multi-threaded process: multiple jobs of exec.
- shared by all threads, take the same
- TI stack, not local vars, no global vars, static obj.

- process control block (OS's OS for processes)
- PID + process state + process priority
- add. esp. + PC + SP
- Register save area
- resource usage info
- list of open files...

I Interupts

- Mechanism (vs policy)
  - HW int: signals I/O event
  - SW int: alarms

- Exception handler
  - OS is event driver

Mode Bit

- Hardware support for protect (required by scheduling)
- Platform
  - RAM: volatile, ROM: non-volatile
  - BIOS (basic I/O sys): 1st program to run after power is supplied
  - HW is in sys node
  - initialize OS OS
  - create 1st proc. (init)
  - node switch - user

- User applications
  - Operating System
    - Sync
    - Select
    - Mem Manage
    - File Sys
    - Networking
    - machine, dependent code

- System Call
  - When user program does a syscall, it passes args to kernel by giving it pointers to buffers, which have to be copied on kernel stack.

- Context Switch
  - Switch CPU to new program by saving state of old, loading that of new
  - More time to pure overhead! - policies should be paramount.
D. Synchronisation

- critical section prob: two threads accessing same shared
  resource must be controlled
- order of thread execution
- mutual exclusion: avoid
- progress: if CS is available, don't hang
- bounded waiting: avoid starvation; by setting lim on CS
  all threads can enter CS, so eventually everyone
evitably does.

Solution: Peterson's alg
- 2 threads share

(Canoni's) Bakery algo give tickets, cut lines by PID.

Abstract of CS
- locks
- condition variables
- semaphores
- monitors
- messages

HW help: disable interrupts (semaphore switches) during CS access.

spin lock: acquire lock by busy waiting: while(test_and_wait(lock))

release lock

semaphore: ADT w/ first var.
wait() (aka decrement) var - block
signal() (aka V, increment) var - unblock

monitors

atomic instruction: lock: if !lock, lock true
unlock: if lock, unlock false
Binary semaphore: single access to CS
Counting: multiple threads (bound by condition) have access to CS

Make sure wait and signal don't happen together

- low-level primitives
- 1 processor, disable interrupts
- multi-hw instruction, e.g., spinlocks

Readers/Writers: w/semaphores: block reads w/

mutex, block writes w/ warn: only 1 reader blocks
on W-on R, rest block on mutex.

- limited wait & signal: only single condition
- condition var: 

  - wait channel
  - cv wait (lock, sleep, acquire)
  - cv signal (wake waiting thread)
  - cv broadcast (wake all threads)

- Monitor: ADT of data only accessible through ops

  - a process enters the monitor through op
  - a single procedure monitors (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.

  - Mesa monitors: when P1 sent to
    Ready Q and P2 continues.

  - need extra checks P2's wait cond.

→ Deadlock & Starvation

Headset of threads S s.t. V(CS) | BCS is waiting for an event

Thread that can only be performed by a TCS.

Some thread that is postponed indefinitely.
Scheduling

Review: Process is instance of program in execution, synchronized to coordinate.

Threads are part of a process, resource sharing.

Threads alternate between CPU (CPU) and I/O.

- CPU: Ready: CPU → I/O → CPU
- I/O: I/O → CPU

Lifecycle:
- NEW
- Ready
- Running
- Exit

Event occurs:
- I/O Interrupt
- Signal

Blocked CPU burst
- I/O burst
- Last CPU burst (call to exit)

Now pre-emptive sched: CPU is kept until thread blocks/exits

Pre-emptive sched: CPU can be given to other thread before

Algs
- FCFS (first come first served)
- SJF (shortest job first)
- Shortest remaining expected run time

Round Robin (RR)
- Very simple implementation concept
- Circular list of processes, with associated
  (identical, i.e. no priority) time slice (quanta)

Prioritize scheduling

Policy: jobs with priority run

Issues:
- Prioritization inverted: low priority tasks get
  high priority
- starvation: low priority tasks

Multi-level queue scheduling (MLQ)
- extra level of scheduling: multiple ready queues with own
  scheduling algorithm

Feedback scheduling
- Decision depends on history: e.g., aging
- combine with MLQ to have threads of the same type
  MLFQ
- Fair Share Scheduling
  - give threads by owners, give appropriate CPU sharing
  - priority of a thread depends on its own priority and past history of gaps.

- Unix Scheduling
  - MLFQ w/ RR on each Q
  - priority of job \( j \): 
    \[
    P_j(i) = \text{base}_j + \frac{\text{CPU}_j(i-1) + \text{nice}}{2} \cdot \text{user}_j \cdot \text{adjunct}_{i\text{th\, interval}} \cdot \text{base}_{i\text{th\, interval}} \cdot \frac{\text{CPU}_j(i) - \text{CPU}_j(i - 1)}{2} \cdot \text{CPU\, utilization of\, } j \text{\, in\, interval } i
    \]

- Memory Management

  Requirements:
  - Relocation
  - protection
  - sharing
  - logical addr -> mapping between physical addr and code modules
  - physical mem -> memory vs disk hierarchy
    - manage flow between levels.

- Address Binding
  - data in code needs to be bound to \( \phi \) add.
  - compile-time binding: absolute code (binary contains real add.) no possible relocation
  - load-time: static relocation (addr

  compiler maps symbolic add. to logical, relocate
  linker takes the whole collection of obj. files (.o) to
Virtual Mem View

0x00

0xFF

- Fixed partitioning of Mem
  - Internal fragmentation
    - Overlays (progs larger than stack)
    - Swapping (a programmer has to deal)
  - Dynamic
    - External fragmentation
      - De-fragment
      - Requires process
        - Post Fragment
        - Place mem alg: 1st fit, 1st block large enough
        - Next fit: from prev. search
        - Worst fit: choose largest block
        - Best fit: choose block w/size closest to needed
        - Quick fit: keep prewrite lists of free blocks for common sizes

- With run-time binding:
  - Process instr. refer to relative addresses.
    - Hard support: base reg + limit reg are loaded at runtime
      - In memory def. instruction, add base to relative address to get proper addr. and compare to checks for illegal addr. except: if A(4B+11A) > B+C, then trap
Paging

- divide 1GB into equal fixed size page frames.

\[ \text{virt mem of proc} \rightarrow \text{virt pages} \rightarrow \text{no ext. frag, int frag is limited} \]

- OS support: page table as OS data

\[ \text{per process} \rightarrow \text{virt addr is page# + page offset} \]

\[ \text{page#} = \text{vaddr} \mod \text{pagesize} \]

\[ \text{offset} = \text{vaddr} \mod \text{pagesize} \]

- HW support: page table base register PTBR

- each mem ref MMU translates page# \rightarrow frame# and adds offset

- e.g. 16-bit \& 1024-byte page size \rightarrow 15 10 9 8 7 6 5 4 3 2 1 0 \text{ need 18 bits to address 1TB}

- 32-bit \& 4096-byte page size \rightarrow \text{max # of addressable pages}

- Page table: array of page table entries PTE

\[ n \gg 13 \rightarrow \text{page size} (12) + \text{only half of mem is user} \rightarrow +1 = 13 \]

- PTE: one per page

\[ \text{valid?} \]

\[ \text{modified?} \]

- limits: exponential size required for PT!

\[ 64 \text{bit} \Rightarrow 16 \text { PTBs} ! \]

- Hierarchical paging

- 2-level paging: virt add

\[ \text{real add} \rightarrow \text{frame#} \text{ offset} \rightarrow \text{mem} \]
32 bit - 4k pages, 4 bytes PTE => we want Hashed Page Table to fit in frame (64 -> 4k bit = 1k = 1024 bits)

64 bit - still a lot of overhead

- Hashed Page Tables
  - virtual address
    - page offset
    - page hash table
    - page

- Paging limits
  - nen ref overhead => huge cache!
  - TLB hits > 99%!
  - TLB misses => need policy to choose which PTE to evict
  - OS maintains PTE in main mem, HW accesses it directly
    - SW loaded
    - OS synchs TLB with PTE

- Paged virtual mem
  - swapping pages mem to disk - demand paging

- locality
  - temporal loc - recently accessed ors are likely to be accessed again
Policies for Mem Manag:

- Fetch policy: when to fetch
- Placement: where to put pages
- Replacement: what to evict
fork returns twice => copy of add. space of parent
exec returns => pcb
exit => zombie while parent waits
context switch => save old, load new

PCB process
add space: code + data, stack (exec space)
PC / registers
OS resources: files, network ports
kernel thread and stack = OS ds

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

multi process < multi thread (kernel) < multi thread (user)

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

critical section
not actual exit
progress: only threads involved in CS can influence
bounded waiting (no starvation): given a waiting proc should be a bound on # of other proc able to access CS
machine instruction for CS

- busy waiting
- starvation
- deadlock - priority inversion

create

NEW → ready → dispatch → release → exit

event occurs

event wait

new process

resource

now pre-emptive - process yields CPU until block/exit

can be (context) switched out

batch

interactive

priority inversion vs. starvation

priority yields makes priority wait

priority always exceed

→ priority starvation

MLFCQ

multiple queues, for each process goes on

depends on feedback allows change
Renes of MM

- Reloc.
- Protection
- Sharing
- Logical vs. physical

Postal

Disk

Load-time

Exec-time = dynamic

Special HW => MMU

Virtual

Dynamic Pach/ching

Compaction

Frame requires relocation

Paging

Ren divided into page frames, proc addressed logically with pages

$\rightarrow$ PTBR: array of PTEs indexed by VPN

- Page tables
- Hierarchical PT
- Hashed PT
- Inverted PT — use hashing to reduce search time
Translation lookaside buffer

→ TLB-EMMU

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

TLB management: consistency of page bits
PTE → TLB

Page schemes → locality [temporal, spatial]

HW: PTEs → MMU, TLB, PTE...

SW: policies → fetch, placement, replacement

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

Replacement algo

- Evicting page that won't be used
- Belady → for the longest period not opt
- LRU Belady's algo: opt for benchmark
- FIFO: suffers from Belady's anomaly:
  - more space = more hits
- LRU: last recently used → good guess
  - exact: $!
  - approximate: counter shift
- 2nd chance: if ref bit = 0, evict otherwise move to next page frame
Page Bubbling, part of free pages
- frames on free list bold prev. cont and can be rescued if referenced before reallocation
  - local repl. every program has fixed # pages
  - global repl. dynamically grows per proc
- Working Set = \( 128 \times 2^{-i} \) for \( i \geq 0 \)
- Page Fault Freq. ad-hoc
  - \( 3 \) for \( 0 \) according to fault

Overcommitted to paging \( \Rightarrow \) thrashing \( \Rightarrow \) more line fetch/relax
- slow making CPU progress

Redux

CPU instr: read \( \rightarrow \) TLB lookup \( \rightarrow \) MMU translates to \( \) page fault \( \rightarrow \) PTE link
- page not allocated \( \Rightarrow \) seg fault
- Upage not in cache \( \Rightarrow \) allocate + fill

Segment BS ??,
File Sys

Hand Links to Acyclic Graphs (keep
Access Control Lists

- use groups (Unix) to avoid ACL

FS implementation:
- block, root in master block, free bit
- dir: linear lists, hash table
- file layout: contiguous alloc, linked (phy to ex), block indexed alloc

- Unix inodes: stores metadata
  - indices blocks used by file/dir

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

Original Unix
  - cylinder gaps
  - LBW, clock

Exploit locality = File Buffer Cache (for Unix)
  - read ahead (forward)
  - flush period

Compare w/ VM
Deadlock

- Need for mutual exclusion
- Hold and wait
- Non-preemptive circular wait

Prevention of these is hard.

Avoidance:

- Bawden's algo

Recovery

- Osmich algo

- Finite res → no log (OS = VMS)

- So only deadlocks are on locks

Write-ahead logging: log transactions, checkpoint writes log → data updates → log

Set of non-conflicting sets of series of browsers

Conflict serializability: can reschedule without conflict

Liveloop: continuously refreshing some ops prevents prog.

Interrupt

Exception → syscall

→ illegal
Security
- Confidentiality: info released
- Integrity: info modified
- Availability: provide access to legitimate users
- Authenticity: establishing 2

Trojan Horses — entry point exploit
Logic Bombs — legitimate destructive code
Trojan Horses — malware that when run can do seemingly
Viruses — replicating its code in progs
Worms — network exploit
- Buffer Overflows: fill buffer until can overwrite return
  address point to exploit code.

- Network Attacks: passive — eavesdropping, encrypt
  traffic analysis, masquerade
  active — replay, capture to reuse,
  modify messages, denial of service, flooding
- Login Spoofing (similar to phishing)

SSL
- Transport Layer Security — Certification Authority
  handshake protocol