Katrina Sharonin

OS Basics

CPL(0) - greatest ⟷ CPL(3) - lowest ☆ i++ → ret original value ++i → ret new value ☆ cond ? -1 : 1; if true → -1

OS: layer of SW that interfaces btw HW rsrcs & applications running on machine, implement VA

Referee: protection, isolation, sharing + Evaluation: performance, reliability, security, portability

Process: CPU + Mem (stack, data, heap, code) + registers (SP, PC) + IO → illusion of priv. CPU

↳ PCB: PC, esp, mem limit, regs, PTBR, PID, UID, FDT, state ☆ **Maskable** == can be disabled/ignored

Kernel: lowest level of OS, full access: Applications (untrusted) ⟷ Kernel ⟷ HW (untrusted) — PT base ptr, halt processor

Dual mode: 2 bit exec. **User mode**: processor checks each instr, limited set (Cannot change: exec. priv. level, addr space, disable interrupts, I/O)

Kernel mode: OS execute w/ protection off, execute any instr. Need safe transfer & control again ↓

Syscall: ☆ Similar to interrupts. user → kernel at specified locations by OS. User req. OS service, transfer at kernel
① Trap instr (int 0x80) → EV 128, eax = sys#,
② Sys. Dispatch table ③ Ret vals in user registers
☆ iret: eip++, intr. (usually) not disabled — Trapping, non-mask, sync

Exceptions: user mode attempts priv operation, unexpected internal event ☆ Use IDT to jump
ex: div/0, stack space explosion

VS — ☆ reuse addr, saves old register vals

Interrupts: asynch signal to processor, external event needs attention
Stop process → enter kernel at interrupt handler ☆ Resetting timer is priv. op.
① Processor detect IRQ → APIC ② Suspend user, switch to kernel → (save before esp = base kern stack; esp, PSW, eip, errcode) Index → IVT, control unit sets eip = handler
③ Interrupt type → invoke handler ④ Restore user prog. ☆ disable intr. when running
☆ Atomic switch ☆ IVT in kernel ☆ Use interrupt stack

Process API: 1 parent, 0+ child, all start from 'init' process

exit(): ☆ syscalls, doesn't wait for thr — terminate process ⤴

fork(): copy addr space (all), code/data seg, reg. (PC+SP (eip)), stack, FDT pointers, new PID, READY
☆ arbitrary ☆ cpid = fork() =0 child, >0 parent

kill(pid): SIGKILL (interrupt like)

exec("ls", args): ☆ pintos "fails" — ① replace code/data seg (+ new stack/heap)
② set eip to start of prog / reinit sp/fp
③ push args to program on stack
wait(): suspend until one child exits
Sigaction(): set handler for sig. ☆ signal: notify pid/pgid of event → sighandler; ☆ User can override sighandler/default action
☆ 0, 1, 2 (stdin, stdout, stderr)

File I/O: ① Uniformity: same open/r/w ② Open before use ③ Byte oriented (access via byte arrs) ④ Kernel buffered R/W (decouple internals) ⑤ Explicit close resource

File descriptor: unique ID for gen IO in rsrc → per-process file desc. table (Global) — access mode, status flag, offset, inode (phys), ☆ for opened

open(filename): ☆ Byte level operation, visible immediately — ret new FD (lowest #)
close(fd): close fd to any file, reuse fd
read(fd, buffer, maxsize), ☆ move offset **write(fd, buff, size)** ☆ move offset
fd3 = dup(f2): new fd, pts to same file; fd2 = dup2(fd1, fd2); fd2 = fd1 sets close old fd2

lseek (fd, offset, whence) 0: off, 1: loc+off, 2: fsiz + offset — VS

FILE* file = **fopen**(char* filename), ☆ operate on streams, unformatted see bytes w/ position
↳ chunk of mem for buffer + file desc (calls open())
☆ "Prefetch" via filter thru buffer, less syscalls
☆ Pro - high: lots of read w/ small bytes ☆ Pro: low, big chunks

☆ high-level is dangerous: since fwrite may not fflush(fd) immediately
☆ fread may move offset 0 - 1000+ bytes → do not mix w/ fd

Interprocess: **Pipe**: one-way, read-only end (pipefd[0]) write only (pipefd[1]),
☆ lim. kernel buffer
Blocks if W called on full, blocks R if empty pipe, close after last W fd closed
☆ queue ☆ ret EOF ☆ after last R fd closes → SIGPIPE signals, if process ignores, W fails w/ EPIPE err.

Socket: 2 queues, R/W either end, fd obtained via socket/bind ☆ 1 close per process, since both see

Stack / Heap / Data / code — Thr switch: reg, PC/SP saved onto TCB stack. Process switch: + PT, PCB, FDT

Copy-on-write: fork copies parent PT → change page → fault/new page fix

Threads: single execution sequence, separately schedulable task ☆ mask I/O latency, split up work, responsiveness to high priority work
☆ No protection (share addr space + FDs) indiv stack/reg. state (eip, esp, ebp)
☆ Virtualizes processor, int processes, sequential code run concurrently (for self)
User thread: 1 PCB for process, 1 thr = 1 TCB in heap, user-space only
Kernel thread: 1 thr = 1 TCB, indiv. schedulable, requires mode switch 2 switch — VS — Linux: everything is thrd = task struct } schedulable

PCB: code, data, FDT, Heap — TCB 1: regs, stack, Metad.

pthread_create(*thread, routine, arg) **pthread_exit**(val_ptr): terminate & make val_ptr available to any successful join → signal ! exit(NULL)
pthread_yield(): yield CPU to other threads, not guaranteed **pthread_join**(thread, **val ptr): suspend until target terminates ☆ join(thr, NULL)

Synchronization: **Multiprocessing** (2+ CPUs) **Mult. programming** (2+ processes) **Multithreading** (2+ thr. per process) **Atomic**: 1 or 0
lock_init(&mylock) or mylock = MUTEX_INIT | **aquire**(&mylock); wait until free, then mark | **release**(&mylock) only by holder

test&set implementation:
```
acquire(int* lock) {
  while (test&set(lock)) {};
} or (!CAS(lock, 0, 1))
release(int* lock) {
  *the lock = 0 }
```

futex lock implementation:
```
acq(lock) { while (test&set(lock))
  futex(&lock, WAIT, 1) }
release(lock) { lock = 0;
  futex (&lock, WAKE, 1) }
```
☆ avoid spin, like int guard e.g. T&S

futex semaphore impl. ☆ CAS(addr, v1, v2): T if *addr = v1, *addr = v2 else F
```
down(*sema) { int cur;
do { while (!(cur = *sema)) {
  futex(sema, WAIT, 0); }
} while (!CAS(sema, cur, cur-1))
```
```
up(*sema) { int cur;
do { cur = *sema }
while (!CAS(sema, cur, cur+1))
} if (cur == 0) {
  futex (sema, WAKE, 1)
```
☆ a=b returns value of b!

int futex (int* uaddr, ☆ SYSCALL!!! ☆ in usr space
FUTEX-WAIT:
if uaddr == val
↳ sleep, else cont.
FUTEX-WAKE:
wake up to val thr

Condvar futex implementation ☆ pointer within user space (uaddr)
```
if (op == WAIT) { enum old_level;
old_level = intr_disable();
if (*uaddr == val) { lock aquire();
  cond_wait (cond, lock); lock release();
  intr_set_level (old_level); ret 0; }
else { intr_set_level(old_level) }
```
```
if (op == WAKE) {
  int num_woken = 0
  lock aquire ()
  while (num_woke < val) {
    cond signal (cond, lock)
    num woken ++
  } lock release
```

Cond Var: allow sleeping in critical section (unlike semaphores) ☆ must hold lock when performing cond op.
lock, cond_op, unlock ☆ queue of threads, efficiently wait for change
cond wait(cv, lock): atomically release lock & suspend calling thr. (inside, reqs)
cond_signal(cv, lock): wake next **cond broadcast**(cv, lock): wake all
Monitor: lock & 0+ cond var ☆ Each CV has own waiting for sleep thr.
Hoare: assume woke immediately schd. if (list empty) { cond wait } ☆ expensive unsafe
Mesa: assume not immediately schd. while (...) { cond_wait } ☆ easier safer

Semaphore: sema_init(≥0), ☆ Atomic **up() / V()** ++ (wake waiting thr), ☆ Atomic **Down() / P()** (wait for val to be >0, dec, sleep if now 0)
☆ init(0) → mutex/lock ex: A start B, A down() → wait, B() up done ☆ init(1) → resource ex: Before consuming, down(), done then up()

Sema lock implementation
```
aquire (lock) {
  sema_down (sema)
  set holder of this lock
}
release (lock) {
  set lock holder = NULL;
  sema_up (sema) }
```

Sema cond var implementation ☆ MUST aquire lock before calling these opps
```
cond_wait(cv, lock) {
  sema init (sema, 0)
  push sema into queue
  lock_release (lock)
  sema_down (sema)
  lock aquire (lock)
}
```
```
cond signal (cv) {
  if (waiters queue not empty) {
    pop waiter off & exec
    sema_up() on their sem }
```
```
cond_brodcast (cv) {
  while (waiters queue not empty) {
    pop waiter off & exec
    sema_up() on their sem }
```
☆ spin locks good if multiprocessor where less active threads than cores → ↑ to release lock & signal spin thr. would be fast

prog	Shared prog
High level API	lock, sema, monitors, send/recieve
HW	Load/St, disable ints, T&S, CAS

☆ old lock
```
aquire () {
  int disable()
  if (val == BUSY) {
    put on wait queue;
    sleep()
  } else { val = BUSY }
  intr enable
```
☆ sleep should tell scheduler to enable intr on next sched. thr.

PintOS

```
struct list_elem {
  struct list_elem *prev;
  struct list_elem *next; }
struct list {
  struct list_elem head;
  struct list_elem tail; }
```

```
struct list * list;
struct list_elem * e;
for(e = list_begin(&list); e != list_end(&list); e = list_remove(e)) {
  word_count_t * cur = list_entry(e, word_count_t, elem) }
```

★ iterate & remove or e = list_next(e)

search for this elem embed | struct type | field name

Base & Bound: Context switch == change base & bound, each has VA 0 → PHYS_BASE (VA illusion)
Base + offset ≤ Base + Bound checked

Processes: ★ child can exit & state freed by parent in wait(), parent doesn't need to exit ★ PCB malloc by kernel
Memory: ★ Kernel: 0xC000 + maps to same PA, kernel always mapped, panic if deref NULL, all user VA: 0 → 0xC000
★ User & kernel stacks separate ★ Syscall can use "kernel only" PTE w/o updating PT
Threads: Shared scheduler for usr & kernel ★ Normal thr: enter kernel & save info on paired kernel stack before scheduling
Kernel only: exclusively on kernel stack
Sys call: thr blocked any time / depth → save kernel stack on wait queue, restore state from another kernel stack
linked on ready queue

Pipe cont.

```
int pipefd[2], ★init fds
int status = pipe(pipefd);
// fork()
if (pid == 0) {  // child
    close(pipefd[1]); // close write
    read(pipefd[0], buf, sizeof(buf))
    close(pipefd[0]); // close read }
else {           // parent
    close(pipefd[0]) // close read
    write(pipefd[1], msg)
    close(pipefd[1]) // close write }
```

Sockets cont.

★ all syscalls

① Create server socket via socket()
② Bind to specific addr via bind()
③ Listen for conns via listen()
★ Unique conn: (srcIP, dstIP, src port, dst port, TCP/UDP)

```
void * serve client(arg) {
  while ((n = read(sock, buf, bufsize)) > 0 {
    //...
    if (write(sock, buf, n) == -1) {
      close(sock)
      //... exit } }
  close(sock); thr_exit; // close up
```

```
int main(argc, argv) {
  server_socket = socket(family, type, protocol)
  if (s.s == -1) ret 1;
  if (bind(serv_sock, addr, addr_len) == -1) ret
  if (listen(socket, 1) == -1) ret 1
  while (1) {
    int com_sock = accept(sock, NULL, NULL)
    // etc.
}
```

File OS cont.

★ guarantee they will read/write same specified amount, unlike low level

- **fopen**(char * path, * mode): ret * FILE, open file assoc. file
- **fclose**(FILE * stream): flush stream & close FD, 0 on success or EOF on err
- **fread**(* ptr, size, n, * stream): read n items of data, each size bytes long, from stream, storing to ptr, ret # items read
- **fwrite**(* ptr, size, n, * stream): write " " " " ret # items written
- **fflush**(FILE * stream): forces all data in user space buffer to stream
- **fprintf**(FILE * stream, * format): print stream according to format
- **fscanf**(FILE * stream, const char * format): scans stream according to format

ssize_t read(int fd, void * buf, size_t count): attempts to read up to count bytes from FD into buffer, ret # bytes read

MT2 Cheat Sheet

Katrina Sharonin

(ex) | 1 | 2 | 3 | 4 | 1 | 3 | 4 | 1 | 3 | 3 |
0 20 28 48 68 88 108 112 125 145 153

Waiting time: $P_1 = 0 + (68-20) + (112-88) = 72$, $P_2 = (20-0) = 20$, (88-48)-

Avg completion time = $(125+28+153+112)/4 = 104.25$ ($P_3 = (28-0) + (125-108)$ -

Scheduling

: process of deciding which thread given access to rsrcs for moments (CPU, disk access, etc)

① Min (response time / latency), user-perceived time to do task ② Max (thruput): tasks/time ③ Min overhead (t to switch) ④ Predictability

Wait time = total time in ready queue **Response time** = t until process first scheduled **Completion t** = W + runtime, total t

Policies

☆ Convoy effect: any non-premptable, long blocks shorts

First Come First Serve (FCFS): FIFO arrival

Pros: simple, good throughput, bc minimizes overhead of context sw.

Cons: avg completion time highly variable, convoy, ☆ can starve since job takes forever

Shortest Job First (SJF): shortest task first ⤵ avoid convoy

Shortest Remaining time first (SRTF): premptive SJF

Pros: optimal for min avg (≤ FCFS) completion time for nonpre & prem.

Cons: psychic / impossible, starvation possible (SJF)

O(n) scheduler: at each switch scan full list of processes in ready q → compute p. score → pick best

cons: bad scalibility w/ more threads / multicore CPUs

O(1) scheduler: like MLFQ, choose next in const. time.

Real time/kernel: 0–99 (0 highest), User: 100–139 ← set w/ nice

Per p. level 2 ready qs: active q (not past Δt) expired q (past Δt); job finish, calc. pri.

Cons: very complex to reason & tune

Round Robin (RR): each run with time quantum (q)

☆ Large q → ☆ higher thruput resembles FCFS ☆ Small q → lots of interleaves

Pros: fair w/ n process & q, max waiting time (n−1)q, no starve

Cons: small q increases completion time / suffers from low thruput + context switching, bad for I/O responsiveness

Multi-level feedback Q (MLFQ): ☆ RR + resolve ties queues for ready jobs per p. level

☆ Approximates SRTF; placed at highest, bump if past Δt given vs. stays same

Cons: gamed with I/O req → no matter how many times given up, reduce if > Δt

Subject to starvation → after time period S, move all jobs to topmost q

Proportional Fair sharing +CFS, d(log n): bad lag for interactive tasks bc of fair share CPU proportional to weights

Job share according to priority, low run less but no starvation

(ex) 1/n ← # users (ex) max-min, user needs less of share (ex) proportional w/ p.

Lottery scheduling: type of CFS; give tix, pick win each Δt

☆ on avg. each job's CPU time proportional to # tickets it has

Cons: could choose long jobs / low prior, unfair for less jobs

☆ Any policy that favors fixed property for scheduling causes starvation

Stride scheduling: deterministic proportional fair sharing; stride = #W / Ni → # tickets → more = smaller stride, run more

① look for lowest pass counter ② run ③ add stride; ☆ Con: long delays until job is rescheduled if big stride vs. small

Earliest Eligible Virtual Deadline First (EEVDF): schedule tasks for Δt proportional to weight, Goal: proportional sharing, Minimize lag

Each task has eligible time to run & deadline to finish by ☆ r_i = flow arrival rate = rate at which flow i transmits across link

☆ Generalized processor sharing (GPS): fluid / weighted fine grain switching $f = \text{Capacity}/(w_1+w_2+w_3)$ → $\min(r_i, f \times w)$ O(1) q

Eligible time = t when req. can be scheduled ⇒ ① At each t only consider eligible process ② pick with earliest deadline!

Lag = ideal service time − real service time ☆ ⊕ Lag = underallocated → reward & schedule earlier vs. ⊖ punish

$f_i(t) = w_i / \sum^n w_i$ → $S_i(t_0, t_1)$ ideal $= f(t_0) \cdot (t_1 - t_0)$] assume f const in $t_0 \to t_1$

(ex) $S(0,1) = 0.5 * (1-0) = 0.5$ s, transmits 0.5 bits (1/2 packet) in t = 0 to 1, full access to CPU for 0.5s

Virtual time V(t): ← at physical time t: round we are on at real time t, rate inversely proportional to Σw active tasks

Moves slower w/ more active ☆ more competition tasks since time / round incr. Round duration = Σw active thr

$V(t_1) = \int_0^{t_1} 1 / \sum w_j \text{active} \cdot dt$ ⇒ If # active thr. const $t_1 \to t$ → $V(t) - V(t_1) = (1/\sum w_j \text{active})(t - t_1)$

Virtual eligible deadline: earliest V(t) (≈ completion t in fluid flow), first eligible time (V_e) is when (VE) thread first arrives ↙ not dictated by time slice

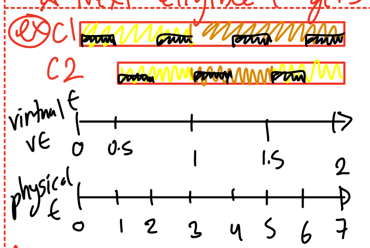
Virtual deadline: earliest V(t) thread can be fully serviced $V_d = V_e + R(\text{request interval}) / w_i$ (client weight)

Algorithm ① every quantum q Ⓐ get $V_e \le$ curr. VE Ⓑ select w/ earliest virtual deadline V_d

② Compute next based on lag Ⓐ New $V_e^{k+1} = V_d^k - (\text{lag})(w_i)$ Ⓑ New $V_d^{k+1} = V_e^{k+1} + R/w_i$

☆ Next eligible t gets impacted by lag → ⊕ lag pulls closer eligible time vs ⊖ postpones it, ← flows w/ ⊕ lag always eligible

(ex)



☆ C1: $w_1 = 2$, $r_1 = 2$ quanta

☆ C2: $w_2 = 2$, $r_2 = 1$ quanta

q = 1 second

☆ no lag → $V_e^{k+1} = V_d^k$

t	v(t)	Active thr	dV/dt	C1	C2
0 +q	0	C1	0.5 $1/w_1$	$V_e = 0$ 0+2/2 $V_d = V_e + R/w_1 = 1$	X ↙ client 2 arrival in phys.
1 +q	0.5 = V_e C2	C1, C2	0.25 $1/(w_1+w_2)$	$V_e = 0$, $V_d = 1$ ☆ Both eligible since $V_e \le t$	$t_0^2 = 1$ (C2 arrival) $V_e = v(1) = 1/w_1 * 1 = 1/2$ $V_d = V_e + r/w_2 = 1$
2	0.75	C1, C2	0.25	$V_e = 0$ $V_d = 1$	$V_e = 1$, $V_d = 1.5$
3	1	C1, C2	0.25	$V_e = 1$, $V_d = 2$	$V_e = 1$, $V_d = 1.5$ earlier dead!
4	1.25	C1, C2	0.25	$V_e = 1$, $V_d = 2$	$V_e = 1.5$, $V_d = 2$
5	1.5	C1, C2	0.25	$V_e = 1$, $V_d = 2$	$V_e = 1.5$, $V_d = 2$
6	1.75	C1, C2	0.25	$V_e = 1$, $V_d = 2$	$V_e = 2$, $V_d = 2.5$

$V_e = V(t_0^2) = V(1) = 1/w_1 * (1-0) = 0.5$ ← inclusive

$V_d = V_e + r/w_2 = 0.5 + 0.5 = 1$ ← exclusive

Next $V_e = V_d = 1$ (no lag)

☆ Let C2 run on the break

V_e = prev V_d (no lag)

$V_d = V_e + r/w = 1 + 1/2$

C1 next since $V_e \le t$

V_e = prev(V_d), $V_d = 1 + 2/2 = 2$

V_e = prev(V_d), $V_d = 1.5 + 1/2 = 2$

Both eligible & same dead, choose C1

Affinity: thr. on CPU, OS tries to reschedule ($ reuse)

Weighted Fair Queing (WFQ): O(n)·q lag, n = # clients approximate GPS, select packet that finishes first in GPS assuming no future arrivals

☆ Every client recieves at most q delay compared to GPS ☆ CONS finish time dynamically changes / recompute if new arrive

Solve with virtual time ☆ Virtual finish time doesn't change w/ new arrivals, always (len packet / w_i)

☆ **System virtual time**: index of current Round in weighted bit by bit RR ☆ $dV/dt = C / N(t)$ ← total weight clients

WFQ: F_i^{k+1} (Round by which packet k+1 send) $= \max(V(a_i^{k+1}), F_i^k) + L_i^{k+1}/w_i$

(arrival round; round when last packet i finishes; # rounds it takes + serve packet k+1)

F_i^k: V finish time of packet k of client i

L_i^k: len of packet k of client i

leave after 3x requests of length ↳ 3 rows no chge

☆ Vs. EEVDF schedules only iff eligible (when it can start) WFQ uses finish time

"Takes up exactly requested service time" = no lag

Deadlock

cyclic waiting for resources, implies starvation (but starvation does not), can't end w/o intervention

① Mutual exclusion & bounded rsrcs: 1 thr. at a time
② Hold & wait: thr. holding at least one rsrc waiting for more rsrc
③ No pre-empt: rsrcs only voluntarily released, after thr. finish
④ Circular wait: {T1...Tn} waiting threads T1→T2...Tn→T1 (★ vs. mul. rsrcs)

① Provide sufficient resources: use virtual memory
② Abort requests or acquire all rsrcs atomically (both)
③ Pre-empt threads (force give up (ex) transaction abort
④ Order rsrcs & always acquire in same order

Max − curr = Need
Total − Current = Avail
★ Donate current to avail

Avoidance

```
[Avail] = [Free Rsrcs]
Add all thr to UNFINISHED
do {
  DONE = true
  for each NODE in UNFINISHED {
    [Request] = [Max_NODE] − [Alloc_NODE]
    if ([request] <= [Avail]) {
      remove NODE from UNFINISHED
      [Avail] += [Alloc NODE]
      DONE = false
    }
  }
} until (DONE)
```

★ if UNFINISHED not empty, deadlock possible

Safe state vs. unsafe (will unavoidably lead to)

(ex) Deadlock Init A=0, B=4, 12 max

```
void threadA() {
  rqust(3) } executes
  rqust(6)   ↳ 12-4-3 = 5 left → blocks
  free() }
void threadB {
  if rand_bool {
    rqust(6) }  ← blocks
  free()
}
```

never free in A or B!

Uniprogramming: app always in same PA, 1 at a time, any PA / no protect

Mem transl. thru Relocation: use loader/linker to adjust addr., no protect

Base & Bound: registers, Base + addr offset < Bound (code, data, heap, stack)

★ Kernel executes w/o Base/Bound reg., Loader rewrites addr to wanted offset

Limits ① No expandable mem: static | W/ relocation: wr. & compiled as if loaded at addr 0

② No mem sharing b/w processes | PA = VA + base } translated by MMU in CPU

③ Non-relative mem addr.: location of code/data det. at runtime | ① relocate dynamically

④ External Frag.: cannot relocate programs | ③ Relative to adjustable reg, can relocate process

⑤ Internal Frag.: addr space must be cont. even if unused | ④ Move processes around & consolidate

Segmentation:

★ shared-mem seg. b/w processes ★ seg. selector registers − which seg. curr. running

cont. portion of addr space of particular len, place each seg indep at diff locations, base/bounds per logical seg.

★ Minimize internal frag. ★ Seg map in processor, Seg # ↔ Base/limit pair; Base + offset = PA | (ex) 0x8020 → 100 0000 0010 0000

| seg # | offset | (ex) 4 seg → $\log_2(4)$ = 2 bits (ex) 16 bit addr → 16−2 = 14 bit offset → seg size = 2^{14} bytes | seg 2 offset 0x20 → 0xF000 + 0x20 = 0xF020

★ HW adds protection bits + limit. Pros: support sparse addr spaces, avoid internal frag, min. HW req/efficient transl. Cons: external frag (var. spaced), moves processes

Paging:

divide logical space of processes into fixed size chunks (pages); PA = array of fixed size slots (page frames), aka PPN, each can contain VM pg.

| VPN | offset | + pid → VPN index into PT → | PPN | offset | (PTE + reference bit for eviction) ★ PT resides in mem. ★ each process has PTBR (base), saved/restored on context switch

- # VA pages = $2^{\#VPN\ bits}$ (★ same for pa)
- Page size (bytes) = $2^{\#offset\ bits}$
- VM size (bytes) = # VA pages × pg size (★ same for pa)
- # PT entries = # V pages
- PT size = # PT entries × PTE size (bytes)
- PTE size (bytes) = # PTE bits / 8 (PPN + meta)

Demand paging: active pages in mem, others on disk + mark PTE invalid

Copy-on-write: copy parent addr space, page sharing + marking R only

Zero fill on demand: New data pages carry no info, mark PTEs invalid, page fault on 0 page

Data break pts: debug, mark page R only, fault if exec.

★ Processes share page, map VA ↔ same frame, protection bits

★ Kernel region of each proc. has same PTE, running same binary → same code seg.

Cons: huge PT size, time overhead access, internal fragment, sparse

Multilevel paging: | VPN1 (base of PT2) | VPN2 (next PT) | offset |

★ If region unused → mark entire inner region invalid

Cons: still big, 2 mem accesses. Pros: good if spaces sparse

TLB: recent addr translations, fully assoc. $, NOT actual data

Tag	Data	
VPN	PPN	Metad

Context switch: store PID, avoid killing all; Invalidate entry if page moved

Paged segmentation: | virtual seg # | VPN # | offset | ★ seg # → Base, check limit → Get PT addr → VPN → check permission bits → PPN + offset

Inverted Page Table: each entry: physical page + pid + which VA maps to phys. ★ size proportional to physical mem (# pages in), << virtual mem

Anchor Hash Table: Hash(VA + PID) → mul. physical frames ★ Relies on good hash! Otherwise VAs all map to same PP & kickout

Caches

AMAT = (HR × HT) + (MR × MT) ★ $AMAT_{L1} = (HT_{L1} \times HR_{L1}) + MR_{L1} \times MP_{L1}$, $MP_{L1} = AMAT_{L2} = HT_2 HR_2 + MR_2 MP_2$

Replacements: FIFO: evict longest in mem, fair due to equal time. MIN: replace pages used farthest in future; optimal but needs future $

LRU: temporal eviction, approx MIN, expensive in HW ★ Belady's anomaly: ↑ $ capacity ↑ MR (yes FIFO, no MIN/LRU since stack) (Stack == set pages of $ size n ⊆ pages n+1 size)

Clock algorithm: approx LRU, old & new pages. page empty, Use: 0 → bring in A:1, next. Hit: set use bit = 1, don't advance. Miss: 1→0, if 0 evict & swap, advance

Demand paging cont: working set: subset in addr space for exec. resident set: subset addr space in curr RAM memory. TLB needs 64 entries

Thrashing: mem. too small for working set, const replacements (ex) 256KB WSS, 4KB pages → $2^8 \cdot 2^{10} / 2^2 \cdot 2^{10} = 2^6 = 64$

Page Fault handling: VA ↔ PA fails (invalid PTE, access violation, DNE) → Trap → engage OS (alloc more stack pages, make page accessible (copy on w), bring page from 2nd-ry mem (demand paging)) ★ Need to execute SW to allow HW to proceed

★ On-chip $ ↔ SRAM ↔ DRAM (main mem, $ for disk) ↔ Secondary Storage (SSD) ↔ TAPE tertiary

Invalid PTE (not in mem, find in disk): ① MMU traps into OS ② OS (A) chooses old page to replace (B) If old page modified, W to disk (C) Change PTE + TLB → invalid (D) Load new page into mem from disk (E) Update PTE, invalid TLB for new entry (F) Cont. thr. from original faulting location (★ replay) ③ TLB for new page loaded when thr cont. ④ while pulling pages from disk, OS runs another process from ready Q, place other on wait

(ex) PA: 8GiB, Page 8KiB, PTE 4 bytes, Levels PT to map 46 bit VA if every PT fits in 1 page?

bytes mapped using PT = # pages pointed by table × page size = (size PT / size PTE) × pg size

= (2^{13} (since each PT = size of page) / 2^2) × 2^{13} = (2^{11}) × 2^{13} = 2^{24} bytes

↳ 2^{11} entries per table → need 2^{46} mapped; find n where

$(2^{11 \times n}) \times 2^{13} = 2^{46}$ → 46−13 = $2^{33} = 2^{11 \times n}$ → n = 3 levels

(ex) w/o TLB: 1 PA access = 3 PT lookups + 1 data access

w/ TLB: Best: 1 data access. Worst: 3 PT + 1 data

AMAT TLB: (10+50) × 0.5 + (1−0.5) × (10+50+50)

AMAT No TLB: 50 + 50 = 100 ★ 50 = main mem, TLB = 10 (fetch PTE, read data)

Digits								
0000 0	0101 5	1010 A	2^2 4	2^8 256	Ki 10	1 ms = 10^{-3}
0001 1	0110 6	1011 B	2^3 8	2^9 512	Mi 20	1 µs = 10^{-6}
0010 2	0111 7	1100 C	2^4 16	2^{10} 1024	Gi 30	1 ns = 10^{-9}
0011 3	1000 8	1101 D	2^5 32	2^{11} 2048	Ti 40	1 ps = 10^{-12}
0100 4	1001 9	1110 E	2^6 64	2^{12} 4096	Pi 50	
		1111 F	2^7 128	2^{13} 8192	Ei 60	

Type	Pros	Cons
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in virtual memory	Multiple memory references per page access
Multi-Level Paging	Fast and easy allocation	
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

★ Little endian (LSB at smallest)
FS F4 — bottom 4 permission
20 07 → top 12 → x2000 0x20A4
0x52 → 16 × 2 bytes → 0x A4 VPN
★ PA: Base + VPN × (PTE size bytes)

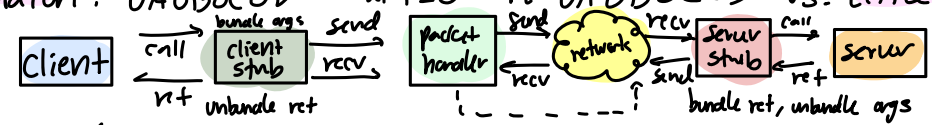
★ Donation: keep highest until release
★ Next runner is waiter w/ highest eff pri, then preempted by any donors, then cont
B C B D B A ★ C = donor 100, D = donor 80 > B 50 > A 1 (e.p. 100, 100, 80, 50, 1)

WSS ≤ RSS ≤ PHYS-MEM (stabilizing process = RSS fits PHYS-MEM)

Distributed Systems Client/server: 1+ server, client makes remote procedure call, server responds
Peer-to-peer: no hierarchy, gossiping ☆ Availability (proportion of time sys in func. cond, use backups)
☆ Fault-tolerance (well defined behavior if fault) ☆ Scalability (can add more +srvrs) ☆ Transparency (easy UI, hide location, migration, replication, concurrency (how many users), parallelism (split job), fault tolerance

Protocol: agreement (syntax: how structured) (semantics: what a com means) ☆ Marshalling = serializ.
☆ Big endian: 0A0B0C0D → at i=0→n 0A0B0C0D vs. Little: 0D0C0B0A (IDL)

RPC:

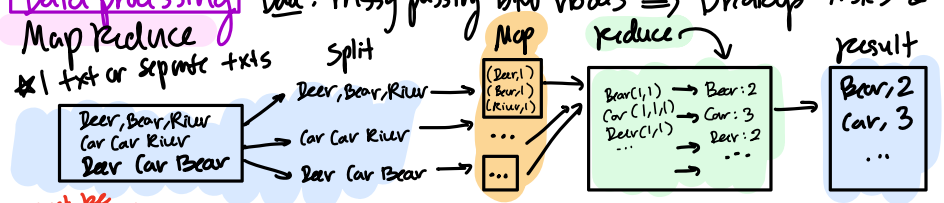


☆ Which mbox/dest to send to? **Binding**: convert user-visible name ⟺ network endpoint (dynamic w/ name service)
☆ CONS: RPC not performance transparent (cost high bc marshalling, stubs etc. same-machine RPC ≪ net RPC)

Dist File Systems: Mount files (local can refer to remote), global name space or (host name, local name)
Virtual File system Switch (VFS): allow same sys call API to be used on diff file systems in same name hierarchy
↳ superblock (mounted file sys) inode (specific file) dentry (dir) file (open f, assoc w/ proc) might take time
☆ CONS: performance (Network >> local mem, server bottleneck) ⇒ caching ⇒ consistency problems
↳ Dealing w/ failures? (1) server file w/o server ack, (2) HTTP changes in buffcache, shared state across RPCs
↳ Stateless protocol: all info to service is included Idempotent: 3x = 1x = nx execution
Network File System (NFS): stateless + idempt., RPC for file ops, write-thru $ (modify disk before ret to client) !!! ↑ poll traffic
↳ Transparent failure model to client ☆ Weak $ consistency: client polls server for changes, may use old ver, arbitrary want sequential order

Internet: IP (can use any network) ⇒ Layering cons (may duplicate N-1 functionality, may need same info, hurt performance (hide details), not cleanly separated (dependencies) e2e: hosts implement themselves, check & try
☆ E2E == increase network complexity, delay/overhead, host sanctions reduces complexity, Conservative (all) moderate (lower)

Data processing Bad: msg passing btw nodes ⇒ Break up tasks & distribute to workers (1 coord)
Map Reduce
☆ 1 text or separate txts

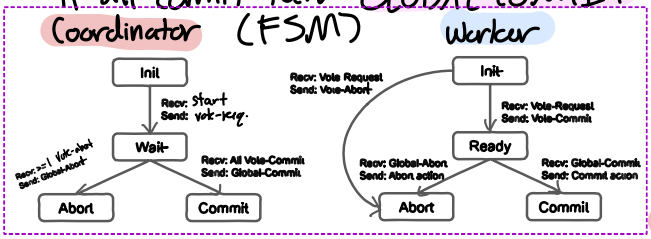


☆ Must be idem. tasks (Kin, Vin) list(K_inter, V_inter) (K_inter, list(v_inter)) list(K_out, V_out)

☆ Fault tolerance: does not handle coord crash
① Map/reduce task crash (map has no dependencies, reduce uses map parts on disk)
↳ retry on another node
↳ if keeps failing ⇒ fail job or ignore input block
② Worker node crash: relaunch tasks on other nodes, relaunch map since need outputs
③ slow task (straggler): launch 2nd copy on another node, take first output, kill other
☆ CONS MR stores intermediate state on disk ⇒ slow

Spark: efficient handling of intermediate datasets, high level ops (SQL, stream etc), reliable virt. mem
↳ Resilient Distributed Datasets (RDDs): immutable collection of objects stored mem/disk, auto rebuilt on failure
↳ Transformations: apply f(RDD) → new RDD Actions: return val to prog or w to disk
Lineage: logged seq. of transformations to create RDD ☆ Failures: recompute RDD partition w/ lineage
Hadoop: single namespace for whole cluster, replicate data 3x, MR framework, Colocate w/ file sys, append only

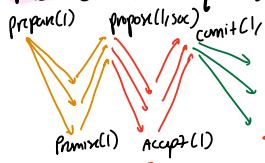
Coordination General's Paradox: if network is unreliable, impossible to guarantee 2 entities do smthg simult.
☆ If malicious, impossible if less than 3f+1 parties (f misbehav)
2 phase Commit (2PC): atomically agree, no constraints on time, either commit or abort transaction eventually choose to commit or abort
☆ Agreement (same decision) ☆ Finality (cannot reverse decision) ☆ Consistency (if no fails + yes, commit) ☆ Termination (no fails/repairs)
① Spray VOTE-REQ ② Send VOTE-COMMIT or VOTE-ABORT (if any ABORT, abort) ③ collect votes, if all commit then GLOBAL-COMMIT else GLOBAL-ABORT ④ GLOBAL dictates local commit or abort



☆ Coordinator can self abort if waiting too long → GLOBAL ABORT
☆ Worker waiting after voting → timeout & terminate
↳ Coordinator crash if waiting too long → termination protocol:
① Block & wait until coord recovers (get GLOBAL-*)
② Ask other worker "p" (optimization) Ⓒ p voted COMMIT, no help
Ⓐ p decided Commit/abort → find decision Ⓑ p not decided, vote ABORT → help decide
☆ Stable storage (assume logs & state restored) Ⓚ VOTE-REQ but no decision → Abort, twice & no vote → vote abort
☆ 2PC not subject to paradox bc all nodes eventually come to same decision (not at same time)
☆ 2PC CONS: blocking VS. 3PC (allow progress if blocks), PAXOS ⇓

Consensus: Termination (eventually decide) Agreement (same val) Validity (proposed by some process)
↳ Impossible on asynch system ⇒ we still do it lol

PAXOS: Safety (consensus not violated) & Eventual liveness (high chance of consensus, no guarantee)
☆ NO COORDINATOR, NO upper bound on msg delay (eventually will arrive) 2f+1 nodes, f fail
Rounds: unique ballot id, asynch, j vs. j+1 rnd → take j+1 → 3 phases/round



Election: majority on ballot_id ⇒ once elect cannot pick smaller id
Bill: if participant already got b > ballot_id, ignore propose, else accept
Law: majority ACCEPT (ballot_id, v), send COMMIT (ballot_id, v)

☆ version 1 failed consensus (multi val)
ex Commit(1, soccer)
Commit(2, football)

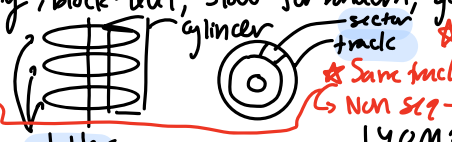
☆ Version 2 election: Prepare(id) → if b > id seen → send PROMISE(id) BUT if already sent ACCEPT(old_ballot id, val) → send PROMISE(id, (old-id, val)) (else do nothing, log
↳ If majority promise(id) → elect, if majority promise(id, (old, old)) → pick v w/ highest old-ballot id val (old must propose v)

Core safety theorem: If 2f+1 participants accepted v in round r, then for all rounds r' > r, proposer gets at least one PROMISE(r', (r, v)) {0 0 0 0 0} ☆ Always 1 overlap ☆ Cannot prove liveness

General I/O ☆ Device types: Block (raw I/O ex disk/ram), Character (single char/time, get()put(), keyboard, mouse, USB) Network (unique I/O, socket interface) I/O layer: device driver. Bus: high BW but slow with close
Single bus: O(n²) relationships w/ 1 set of wires, but blocks all except 1 **Parallel:** hard to synch
Device controller: registers (R/W) ⇒ **Programmed I/O**, CPU involved in every data transfer, heavy polling
① **Port-mapped I/O**: separate addr space (in/out instr to comm. w/ device registers)
② **Mem-mapped I/O**: looks like mem access, consumes local addr space (ld/str instr)
VS

Direct Mem Access (DMA): ① CPU gives DMA rq. to device controller ② device + DMA controller transfer data ③ when done DMA controller interrupts CPU ☆ DMA has direct control of mem. bus ☆ No bottom half if NO DMA or interr. driven I/O
↳ If uses DMA, **device driver** (device specific code in kernel that interacts w/ HW) can split into 2 pts
↳ **Top half** (sync): accessed in call paths via syscalls, kernel interface, **Bottom** (async): interrupt routine, get in/transfers, wake sleeping I/O
☆ **Blocking**: wait until data ready ☆ **Non-blocking**: immediate, might return nothing ☆ **Asynch**: ptr to buf, immediate ret, notify fill

Storage Devices HDDs (Magnetic): rarely corrupted, large capacity, block-level, slow for random, good sequential
Latency = Q time + controller + seek + rotational + transfer
☆ **Seek**: position head on track vs. Ø if same cylinder
☆ **Rot**: wait for desired sector (none if same track + sequential) else estimate 1/2 rot.
☆ **Transfer**: R/W min time, transfer sector under head



☆ Same cyl. → No seek
☆ Same track → Seq → No rot
↳ Non seq → assume rot
140MB/s transfer

ex 4096 sector, 3×10^6 sector/track, 100 track/platter, 2 platter, 5400 rpm, 5.6 ms seek, 1 ms cont + Q,
① Size? $((3\times10^6 \times 4096) \times 100) \times 2$ ② Thruput 64KB read? 1 ms + 5.6 ms + ($\frac{1}{5400}$ min = 11.11/2 rot = 5.55 avg) + 64KB/140MB/s (transfer) = 0.457 ⇒ total ≈ 12.6 ⇒ thruput = 64KB/12.6 ☆ Block size ≥ sector size, BOTH use block API

Disk scheduling: FIFO (fair but random), SSTF (shortest seek) (may starve), SCAN (closest rq. in dir of travel, no starvation) ≥ C-SCAN (one-dir, skips otw back, fairer than SCAN since not biased towards middle)

SSDs (storage state): no moving parts, limited W, low power ☆ R/W pages, erase blocks (page < block)
Bank (access in parallel) > Blocks (64–256 pages) > Pages few KB > Cells (1–4 bits) ☆ 2–3 (MLC) wears faster

☆ No seeks, expensive/smaller
Latency = Q + controller + transfer

Flash translation layer (FTL): translate logical block to flash blocks, reduce↓ amplification (control/gc W traffic), avoid wear, indirection (DRAM mappings) (copy on W (w new page then update ptr)

File Systems ☆ Most files small ☆ Most bytes in large files ☆ Fixed size blocks
FFS: File # → inode array → inode (metadata + DP & IP) → DP or array of pointers (ptr size)
ex 12 dir + 1 Dbl + 1 Indir (4 KiB block) == $12\times4KB + 1024\times4KB + 1024^2\times4KB$ ☆ $1024 = 2^{12}/2^2$
☆ New file → find free inode + free DB ☆ Disk optimizations: dist inodes on (same cylinder + diff tracks) (reduce seek), contiguous alloc ☆ Dir + files in same **block group** + free map ☆ 10% free inode ≠ dir entry / 1:1 driven.
Pros: seq., random, no external frag, big files, locality for file + meta, ok for small
Cons: tiny (read inode + whole block) → internal frag, but DP do ok, bad encoding, 10% free

easy to implement for firmware
bad security/perf.
locality for metadata, stored separately

FAT (File alloc table): file # → index into table → LL of blocks, free list (LL), append to grow
Dir: < file name : file # > ☆ file attributes stored in dir not file itself → Linear search
Pros: sequential, no external frag, big files Cons: random (LL traverse), internal frag (if 2 many small)

NTFS (new tech): file # → index MFT → Record: file meta + data OR ptr to extents (variable len chunks of data) OR if massive, ptr to lists of extents (like dbl); MFT always has file name
Hard link: < name : file#>, link(), part of 2+ dirs, cannot be removed until all HL gone ☆ NOT supported by FAT
Soft link: < name: path>, symlink(), no ref cnt, if path DNE, fail ☆ Since no inode, cannot make mul direntries

Buffer cache temporal locality, LRU good unless scanning which kicks, delayed W/write back periodically ☆ Paging uses clock
↳ CONS: interrupted → loss of shared data → ① careful order ② Copy on W: make new file, then pt ptr to there
⇓

Durability: being able to recover from disk failure **Reliability**: after recover want consistent state
Transaction: unit, atomic seq of R/W that goes from consistent state 1 → state 2 on failure
Atomicity **Consistency** (valid state to another) **Isolation** (concurrent exec. ok) **Durability** (once committed, not erased)
Log (data stays in log form), write commit to log; ⇓ **Journaling** expensive (writing 2x data) → modern journal metadata from log
Journaling (log for recovery) write data to log; once transaction written, disk apply changes, then remove transaction
↳ Case 1: sys crashes when wr to log → incomplete transaction → no changes applied ✱Idempotent
↳ Case 2: sys crashes applying changes to disk → re-apply transaction by looking at log (ex) ext3 FFS+ jlogs

Discussion Problems

Disc 7

Corrupted sectors: spread backup evenly, checksum, incr. percieved wr speed by signaling from buff$, not platter (in background)
Thruput (ex) seek 9ms, 7200 RPM, transfer rate 50 MiB/s

① random accss, 4 KiB thruput?
↳ on avg ½ revolution
↳ ½ × 1/7200 = 4.17 ms rot. t
Transfer t: 4KiB × 1/50 MiB = 0.07812 5
t = 8 + 4.17 + 0.07 = 12.248
Total thruput = 4 KiB / 12.2 = 326.6 KiB/s

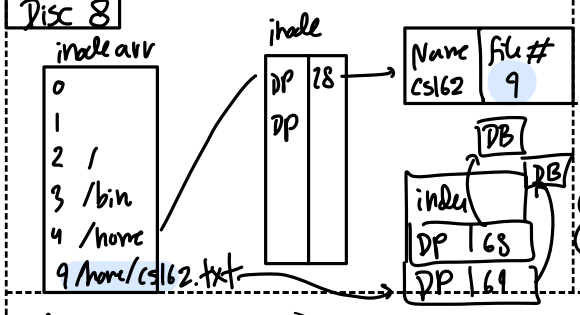
② 4GiB same track?
↳ No seek time
↳ 4.17 + 0.078...

③ Next sector?
↳ No rot/seek
↳ just transfer rate

```
inode_resize(id, size) {
  for(i=0; i<12; i++) {
    if (size <= 512 * i && id->direct[i]!=0) {
      free(id->dir[i])
      id->dir[i] = 0          ✱Shrink
    } else if (size > 512 * i && id->dir[i]==0)
      id->dir[i] = block_alloc() }
  if (id->indir == 0 && size <= 12 * 512)
    id->len = size
    ret true
```

```
block_sect_t buff[128]
memset(buf, 0, 512)
if (id->indir = 0)
  id->indir = block alloc
else
  block_read(id->indir, buffer)
```

Disc 8



Quick format: mark all blocks free Full: quick + zero out DB
FAT vol size / block size = 2^{19} entries × 4 bytes size = 2^{21} bytes
✱FAT32 → 4 GiB size limit

FAT steps D:\MyFiles\video.mp4
① Read in blocks of / until MyFiles found <name: file #>
② Read in blocks of /MyFiles until video.mp4 found <name: file #>
③ File # = block 0 → read all blocks until EOF

```
for (int i=0; i<128, i++)
  if (size <= (12+i) * 512 && buff[i]!=0)
    block_free(buff[i])          Shrink
    buff[i] = 0
  else if (size > (12+i) * 512 && buff(i)==0)
    buff[i] = block alloc()      grow
}
```

```
if (size <= 12 x 512)
  block_free(id->indir)
  id->indir = 0                  shrink sm that indir gone
} else
  block_wr(id->indir, buff)
id->len = size;
```

```
Sector alloc failures
sector = alloc_block()
if (sector == 0)
  inode_resize(id, id->len)
  ret false
}
```

Disc 9

Availibity: probability sys. can accpt & process req (99.9% = 3 nines of availbity)
Durability ≠ Avail. ✱Durability ⇒ reed-solomon ECC (short term) RAID (long term)
✱ even if commit → nothing applied OR intermediate state OR all updates written to disk

Linux uses ~ & vfat to access files in flash drives formatted w/ FAT32
NFS: may have inconsistant views due to polling/cache vs. local file sys. sees truth
NFS is stateless

Disc 10

Completed map tasks reexec when worker crashes, coord. finds locations of map results
to reduce workers → reduce worker incurs RPC to read data
Lazy eval. transformation → avoid unessary comp., optimize query knowing computation graph

```
map(key, val) {
  Emit(key, val)
}
```

```
reduce(key, list<val>) {
  lowest = INIT_MAX
  for (val : List<val>)
    lowest = min(lowest, val)
  Emit(problem name, lowest)
}
```

2PC: optimize by ignoring about decisions → reduce logging
✱Blocked until bad, may hog rsrcs

worker	Send/Rcv t	log
W1	400	10 ms
W2	300 ↓	70 ms
W3	200 ms	30 ms

C has timeout 3s
log latency 5ms
parallel com

① Min time for 2PC to complete successfully
prep phase = worker recieves + log + send back + coord logs
= max(400+10+400, 300+70+300, 200+30+200) + 5
Comit/abort phase = worker recieves + log result + send back (no coord log)
max(400+10+400, 300+70+300, 200+30+200) ⇒ Total = Prep + comit

② All 3 commit, broadcast com't, W2 crashes & recovers after timeout of C → transactn comit/abort? latency?
Still commits
Prep phase = 815 (like last)
Comit: try 1: 3000ms (timeout)
try 2: 300+70+300
total = 815 + 3000 → 670

Deadlock cyclic waiting for resources, implies starvation (but starvation does not), can't end w/o intervention

① Mutual exclusion & bounded rsrcs: 1 thr. at a time
② Hold & wait: thr. holding at least one rsrc waiting for more rsrc
③ No pre-empt: rsrcs only voluntarily released, after thr. finish
④ Circular wait: {T1…Tn} waiting threads T1→T2…Tn→T1 (☆ vs. mul. rsrcs)

① Provide sufficient resources: use virtual memory
② Abort requests or acquire all rsrcs atomically (both)
③ Pre-empt threads (force give up (ex) transaction abort)
④ Order rsrcs & always acquire in same order

Max − curr = Need
Total − Current = Avail
☆ Donate current to avail

[Avail] = [Free Rsrcs]
Add all thr to UNFINISHED
do {
  DONE = true
  for each NODE in UNFINISHED {
    [Request] = [Max$_{NODE}$] − [Alloc$_{NODE}$]
    if ([request] <= [Avail]) {
      remove NODE from UNFINISHED
      [Avail] += [Alloc$_{NODE}$]
      DONE = false
    }
  }
} until (DONE)

Avoidance
☆ If UNFINISHED not empty, deadlock possible
Safe state vs. unsafe (will unavoidably lead to)

(ex) Deadlock Init A=0, B=4, 12 max
void threadA() {
  request(3) } executes ↳ 12−4−3 = 5 left
  request(6) ← blocks
  free() }
void threadB {
  if rand_bool {
    request(6) } ← blocks
  free()
  ↳ never free in A or B!

Uniprogramming: app always in same PA, 1 at a time, any PA / no protect
Mem transl. thru Relocation: use loader/linker to adjust addr., no protect
Base & Bound: registers, Base + addr offset < Bound (code, data, heap, stack)
☆ Kernel executes w/o Base/Bound reg., Loader rewrites addr to wanted offset
Limits ① No expandable mem: static | W/ relocation: wr. & compiled as if loaded at addr ∅
② No mem sharing btw processes | PA = VA + base } translated by MMU in CPU
③ Non-relative mem addr.: location of code/data det. at runtime | ① relocate dynamically
④ External Frag.: cannot relocate programs | ③ Relative to adjustable reg, can relocate process
⑤ Internal Frag: addr space must be cont., even if unused | ④ Move process around & consolidate

Segmentation: ☆shared-mem seg. btw processes ☆seg. selector registers – which seg. curr. running. cont. portion of addr space of particular len / place each seg indep at diff locations, base/bounds per logical seg.
☆ Minimize internal frag. ☆ Seg map in processor, Seg # ↔ Base/limit pair; Base + offset = PA | (ex) 0x8020 → 100 | 000 0010 0000
| seg # | offset | (ex) 4 seg → $\log_2(4) = 2$ bits (ex) 16 bit addr → 16−2 = 14 bit offset → seg size = 2^{14} bytes | seg 2 offset 0x20 → 0xF000 + 0x20 = 0xF020
☆ HW adds protection bits + limit. Pros: support sparse addr spaces, avoid internal frag, min. HW req / efficient transl. Cons: external frag (var. spaced), moves processes

Paging: divide logical space of processes into fixed size chunks (pages); PA = array of fixed size slots (page frames) aka PPN, each can contain VM pg.
| VPN | offset | + pid → VPN index into PT → | PPN | offset | PTE + reference bit for eviction ☆ PT resides in mem. ☆ each process has PTBR (base), saved/restored on context switch

$\#VA\ pages = 2^{\#VPN\ bits}$ (☆ same for PA)
$Page\ size\ (bytes) = 2^{\#offset\ bits}$
VM size (bytes) = #VA pages × pg size (☆ same for PA)
#PT entries = # V pages
PT size = #PT entries × PTE size (bytes)
PTE size (bytes) = #PTE bits / 8 (PPN + meta)

Demand paging: active pages in mem, others on disk + mark PTE invalid
Copy-on-write: copy parent addr space, page sharing + marking R only
Zero fill on demand: New data pages carry no info, mark PTEs invalid, page fault on 0 page
Data breakpts: debug, mark page R only, fault if exec.
☆ Processes share page, map VA ↔ same frame, protection bits
☆ Kernel region of each proc. has same PTE, running same binary → same code seg.

Cons: huge PT size, time overhead access, internal fragment, sparse
Multilevel paging: | VPN1 (base of PT2) | VPN2 (next PT) | offset |
☆ If region unused → mark entire inner region invalid
Cons: still big, 2 mem accesses Pros: good if spaces sparse
TLB: recent addr translations, fully assoc. $, NOT actual data
Tag	Data	
VPN	PPN	Metad

Context switch: store PID, avoid killing all; Invalidate entry if page moved

Paged segmentation: | virtual seg # | VPN # | offset | ☆ seg# → Base, check limit → Get PT addr → VPN → check permission bits → PPN + offset
Inverted Page Table: each entry: physical page + pid + which VA maps to phys. ☆ size proportional to # pages in physical mem, << virtual mem
Anchor Hash Table: Hash(VA + PID) → mul. physical frames ☆ Relies on good hash! Otherwise VAs all map to same PP & kickout
Caches: $AMAT = (HR \times HT) + (MR \times MT)$ ☆ $AMAT_{L1} = (HT_{L1} \times HR_{L1}) + MR_{L1} \times MP_{L1}$; $MP_{L1} = AMAT_{L2} = HT_2 HR_2 + MR_2 MP_2$
Replacements: FIFO: evict longest in mem, fair due to equal time MIN: replace pages used farthest in future; optimal but needs future $
LRU: temporal eviction, approx MIN, expensive in HW ☆ Belady's anomaly: ↑ $ capacity ↑ MR (yes FIFO, no MIN/LRU since stack) (stack == set pages of $ size n ⊆ pages n+1 size)
Clock algorithm: approx LRU, old & new pages. page empty, use: 0 → bring in A:1, next. Hit: set use bit = 1, don't advance Miss: 1 → 0, if 0 evict & swap, advance
Demand paging cont: working set: subset in addr space for exec. resident set: subset addr space in curr RAM memory. TLB needs 64 entries
Thrashing: mem. too small for working set, const replacements (ex) 256KB WSS, 4KB pages → $2^8 \cdot 2^{10} / 2^2 \cdot 2^{10} = 2^6 = 64$
Page Fault handling: VA ↔ PA fails (invalid PTE, access violation, DNE) → Trap → engage OS (alloc more stack pages, make page accessible (copy on w), bring page from 2nd-ary mem (demand paging)) ☆ Need to execute SW to allow HW to proceed
☆ On-chip $ ↔ SRAM ↔ DRAM (main mem, $ for disk) ↔ Secondary Storage (SSD) ↔ TAPE tertiary
Invalid PTE (not in mem, find in disk): ① MMU traps into OS ② OS Ⓐ chooses old page to replace Ⓑ If old page modified, W to disk Ⓒ Change PTE + TLB → invalid Ⓓ Load new page into mem from disk Ⓔ Update PTE, invalid TLB for new entry Ⓕ Cont. thr. from original faulting location (☆replay) ③ TLB for new page loaded when thr cont. ④ while pulling pages from disk, OS runs another process from ready Q, place other on wait

(ex) PA: 8GiB, Page 8KiB, PTE 4 bytes, Levels PT to map 46 bit VA if every PT fits in 1 page?
#bytes mapped using PT = # pages pointed by table × page size = (size PT / size PTE) × pg size
$= (2^{13}$ (since each PT = size of page$) / 2^2) \times 2^{13} = (2^{11}) \times 2^{13} = 2^{24}$ bytes
↳ 2^{11} entries per table → need 2^{46} mapped; find n where
$(2^{11 \times n}) \times 2^{13} = 2^{46}$ → $46 - 13 = 33$, $2^{33} = 2^{11 \times n}$ → n = 3 levels

(ex) w/o TLB: 1 PA access = 3 PT lookups + 1 data access
w/ TLB: Best: 1 data access Worst: 3 PT + 1 data
AMAT TLB: $(10+50) \times 0.5 + (1-0.5) \times (10+50+50)$
AMAT No TLB: 50 + 50 = 100 ☆ 50 = main mem, TLB = 10
(fetch PTE, read data)

Digits

0000 0	0101 5	1010 A
0001 1	0110 6	1011 B
0010 2	0111 7	1100 C
0011 3	1000 8	1101 D
0100 4	1001 9	1110 E, 1111 F

2^2	4	2^8	256	Ki 10
2^3	8	2^9	512	Mi 20
2^4	16	2^{10}	1024	Gi 30
2^5	32	2^{11}	2048	Ti 40
2^6	64	2^{12}	4096	Pi 50
2^7	128	2^{13}	8192	Ei 60

$1\ ms = 10^{-3}$
$1\ \mu s = 10^{-6}$
$1\ ns = 10^{-9}$
$1\ ps = 10^{-12}$

Type	Pros	Cons
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in virtual memory	Multiple memory references per page access
Multi-Level Paging	Fast and easy allocation	
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

☆ Little endian (LSB at smallest)
FS F4 → bottom 4 permission
20 07 → top 12 → x2000 0x20A4
0x52 → 16 × 2 bytes → 0x A4 ↑VPN
☆ PA: Base + VPN × (PTE size bytes)

☆ Donation: keep highest until release
☆ Next runner is waiter w/ highest eff pri, run preempted by any donors, then cont
B C B D B A ☆ C = donor 100, D = donor 80 > B 50 > A 1
(eff: 100, 100, 80, 50, 1)

WSS ≤ RSS ≤ PHYS-MEM (swapping process = RSS fits PHYS-MEM)