Memory

- MMU: Translate vaddr → paddr on every load/store/instruction fetch
  - ↳ Read PTE, check valid, merge
  - ↳ Set "accessed" bit; "dirty" on write

Base & Bound

Set registers w/base and limit

- ⊕ Simple
- ⊖ Internal + external fragmentation (between blocks)
- ⊖ No sparse addr. space support for interprocess sharing, e.g. chunks for code/data/stack/heap

Both resolved w/segmentation

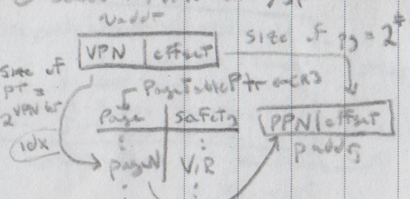
- ⊖ May need to swap on context switch (i.e. store process mem to disk)

\# paddr bits = \# PPN + \# off; \# p. pgs = $2^{\#PPN}$
\# vaddr bits = \# VPN + \# off; \# v. pgs = $2^{\#VPN}$
vmem (B) = \# v. pgs × pg_size; pg_size (B) = $2^{\#off}$
PT size (B) = \# v. pgs (\# PTEs) × (\# PPN + \# meta)/8 ← PTE (B)

Paging

Each proc. has Page Table in p. mem. maps VPN → PPN + safety bits

- ⊕ Fixed size chunks (pages), no external frag.
  - ↳ Prevent internal frag. by limiting to ~1-16 KiB; use numerous pages per segment
- ⊕ Sharable mem: diff. VPNs map to same PPN (proc.)
  - ↳ Ex. Kernel region has same PTE, shared exec-only user libs
- ⊖ Scales w/VPN bits; sparse (or programs unfound)



PTE = \# PPN bits + safety
\# paddr bits = \# PPN + \# offset

Demand Paging: Treat DRAM as cache on disk; PT tracks pages in memory

not in mem ⇒ page fault, OS loads to mem

Page Fault: vaddr not mapped

(1) MMU traps kernel, saving reg/state
(2) Kernel stores needed VPN in CR2
(3) Check validity, move page to disk and load + update PTE
(4) Restore saved state, resume exec.

Inverted Page Table

- Other "Forward" PTs have size of PT ≥ vmem alloc to process
- Pmem. may be less than vmem, i.e. on disk
- Use hash table, mapping VPN → PPN
- ⊕ Size indep of vaddr space, good for 64B addr.
- ⊕ Directly related to amount of physical mem.
- ⊖ Poor cache locality
- ⊖ Hash chains complexity

Multilevel: Chain PT in "tree" structure

- ⊕ Much smaller: alloc PTEs when needed
- ⊖ ≥ 2 lookups per reference, diminishing returns
- ⊕ Unused PT can be stored in disk (e.g. during context switch)
- ⊕ Share granularity at PT/page scales
- ⊖ PT must be contiguous
  - ↳ 10-10-12 keeps table to exactly 1 pg in size

Replacement Policies

1. FIFO (queue). Evict oldest
   - ⊖ Evicts heavily used entries
   - ⊖ Does suffer from Belady's anomaly
2. LRU.
   - ⊖ Overhead tracking (+ possibly comparing over sets) (n! compares)
3. Random.
   - ⊕ Empirically on par w/LRU if temporal locality is low
4. MIN. Replace entry not used for longest time
   - ⊖ Non-causal; ⊕ Theoretically opt

Types of Misses:

1. Compulsory. First access to block
   - Avoid w/prefetching (sequential disk reads)
   - Insignificant in large scheme [could also tackle working set too]
2. Capacity. Too many blocks; finite size
   - Increase cache size
   - Reduce associativity, e.x. access pattern 0, 32, 64, … 32N, 0 …. An N-line direct map cache just misses at 0 & 32N conflict while fully-assoc. LRU misses all
3. Conflict/collision. Multiple mem. locations mapped to same cache location
   - Increase cache size/assoc.
4. Coherence. Other processes (I/O) update memory
   - Due to parallelism

Set-Associative: N-entries per cache idx

- Index refers to set of rows now
  - ↳ Check all tags; size scales × \# sets, n
- ⊕ Reduces "paging" of conflict misses
- ⊖ Comparators scale w/n

10-10-12 on 4KiB pages

- Space of one page mapped = 2 × 4KiB
- 2^{10} entries of {ptrs/data = 4 Bytes} ⇒ PT = 2^{12} B = 4 KiB = pg size (wow)
- Top level page references to (1 + 1024) [pages] · 4096 [bytes/page] = 4096 + 2^{22} B [max size of PT]

Avg. Mem. Access Time = Hit_p × Hit_time + Miss_p × Miss_time

Demand Paging cont.

- Working set: addr space used during execution (div by PG size for TLB entries)
- Resident set: subset held in mem
  - ↳ Thrashes when too small

Caching

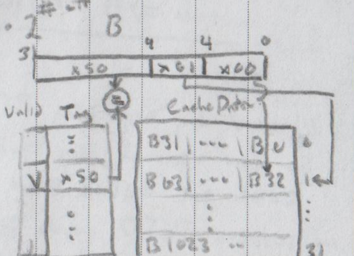
- MMU translation occur on every access, taking ~ PT height
  - ↳ Worse if PT on disk
  - ↳ Motivates TLB (translation cache) State (VPN\#, PTE) + process ID
- On context switch, vaddr changes: Flush TLB, or (in hardware) include ProcessID
  - ↳ Aside: Virt. Index/Tag cache would need to be flushed

- Block: minimum quantum of caching
  - ↳ offset used to identify data inside block, sometimes not needed
- Index: Identifies set
- Tag: Identifies actual copy

- ☐ Write-Through: write to cache + mem
  - ⊕ Read misses ≠ writes; ⊖ I/O slow
- ☐ Write-Back: write to block; mem on evict
  - ⊖ Read miss may need write of dirty data; ⊕ Less I/O

Direct: Single block per set/index

- 2^N byte cache has (32 − N) MSB tag bits
- LS M bits for byte select (block size 2^M B)
- Remaining bits for index (height, \# blocks)
  - ↳ Size: $2^{\#idx} \cdot 2^{\#off}$ B
- No eviction policy needed



Fully Associative

Any block can hold any line ⇒ no idx

- ↳ Must compare w/all cache tags, done in hardware, parallel, but difficult
- TLB of ~128-512 entries
  - ↳ Conflict misses expensive

Aside:

- TLB lookup & cache access can occur in parallel if cache idx fits in page offset bits (const. in p. & v. addr space)

Scheduling | Asm. single thread, indep. programs

Goals/Criteria:

1. Min. Avg. completion time
   - waiting + run time (response)
   - crucial for time-sensitive (I/O)
2. Max throughput (ops/sec.)
   - Min. overhead (context switches)
3. Fair
   - Is not min. avg. response time
   - ↳ Priority schedule very unfair

Throughput ⇒ FCFS
Avg. response ⇒ SRTF approximation
I/O throughput ⇒ 〃
Fair (CPU time) ⇒ CFS
Fair (wait time) ⇒ RR
Meeting Deadlines ⇒ EDF
Favoring Importance ⇒ Priority/Lottery

Meltdown

- Resolved by separating PT for user + kernel (e.g. Flush twice per syscall, 800% overhead)
- ↳ Hardware PCID avoids flushing when changing addr space

FIFO/FCFS:

- One program runs til done
- ⊕ Least overhead, simple
- ⊖ Convoy effect; response time depends on arrival ordering

Round Robin (RR):

- Each proc gets fixed CPU time q
- q ≥ longest task ≡ FCFS
- ⊕ Curbs convoy w/preemption; max wait time of (n−1)q (excluding context swaps)
- ⊖ Lots of context swaps; need q >> context switch to ensure overhead isn't too great
- Small q ~ SJF; lower response time, higher completion 〃

SJF/SRTF:

- Run job w/least amount of work to do; non-causal
- ⊕ Optimal min avg. completion
- ⊖ Impossible to know proc length; starvation if many short jobs block long job

Lottery:

- Give jobs tickets, randomly choose
- Give all one ticket ⇒ no starvation
- ⊕ On avg., CPU time ~ # tickets
- ⊖ Unfair for less jobs
- Gracefully handles add/rm jobs

MLFQ:

- Early RR; later FCFS] Each gets set amount of time
- ⊕ Approximates SRTF
- ⊖ Overly punished for occasional I/O

2nd Chance: Split into Active (RW, FIFO) and SC (invalid, LRU)

- Search direct-map Active first
- PF: Pop active, mark invalid push to SC
  - ↳ In SC: push head of active, mark RW
  - ↳ New page in H; page out LRU

EDF: D_i C_i

- Given tasks w/ deadlines + compute time
- ⊕ Predictable; provide guarantees
  - ↳ Feasible if $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1$
- by choosing closest D_i

Strided

- Proportional share time w/o randomness
- Let $\text{Stride}_i = W/N_i$ for N_i tickets, const. W ≈ 10,000
- Pick job w/lowest "pass", schedule & add stride to its pass
- ↳ More tickets ⇒ lower stride, scheduled more often

CFS:

- Illusion of 1/N CPU time
- Track CPU time per thread, schedule lowest
- Sleeping threads don't inflate CPU time, get interactivity (boost when waking)
- Quanta_i = max(min_granularity, target_latency/N)
  - ▫ Target latency: period in which all threads serviced; no starvation
  - ▫ min_gran prevents excessive overhead
- Weighted quanta:
  - Q_i = target_latency × $(w_i / \sum_p w_p)$
  - ↳ Less weight ⇒ more time (urgent); nice value, stride

Clock: Approximates LRU/mm

- Circle of in-mem PTEs w/ use bit that's set to 1 on reference
- On PF, while hand ptrs to an in-use PTE, it zeroes and advances; then evicts candidate for replacement and advances
  - ↳ If all in use ≡ FIFO
- Partitions into new/old (2 sets)
- ↳ Extend w/N^{th} chance: evict candidate increases counter, replaced if reaches N

Deadlock | ⇒ Starvation, but starvation ⇏ deadlock (starvation can end, deadlock can't end)

Necessary conditions:

1. Mutual Exclusion & Bounded Resources
   - Finite # threads (i.e. 1) can simultaneously use a resource
   - ↳ Infinite illusion, ex VMEM
2. Hold & Wait
   - Thread holding one resource continue to do so when attempting to acquire another held resource
   - ↳ Abort request; acquire atomically; indep. threads
3. No preemption
   - Resources released voluntarily
   - ↳ Fail if waiting too long (leaves inconsistent state) / don't allow waiting (inefficient)
4. Circular Waiting
   - Set of threads waiting on eachother
   - ↳ Order resource usage

Coping:

- Design specifically s.t. deadlocks impossible (restrictive)
- Recovery: Roll back actions of deadlocked threads; prone to same execution order
- Avoidance: Dynamically delay resource requests.
  - Bankers: Stay in safe state (opposite means possible ordering leading to deadlock)
  - ▫ Threads state resource needs prior.
  - ▫ Pretend request granted, run deadlock detection algo

Let max_i be required resource of thread i, $alloc_i$ the current possessions of i

```
unFin ⇐ all threads; done = False
while not done:
    done = True
    for t in unFin:
        if max_t − alloc_t ≤ avail:    ← request
            unFin.rm(t)
            avail += alloc_t
            done = False
rt len(unFin) == 0   // True if ∃ safe ordering
```

Heuristic: K-handed lawyers:
- Last: don't acq. if no one would have K
- 2^{nd} last: no one K−1
- 3^{rd} last: no one K−2 …

deadlock free (safe state): Not if ∀: $(alloc_i + (\text{free} - 1)) < K$

- N ~ large ≡ LRU approx
- N less ⇒ more efficient

Can be used to manage Free pgs list } • Dirty pgs write when added to list; • Allows faster PF handling

---

ms = 10^{-3} s
μs = 10^{-6} s
ns = 10^{-9}
ps = 10^{-12}

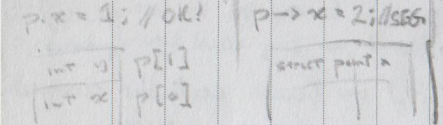
1KiB = 1024 B = 2^{10} B
1MiB = 2^{20} B ≈ 10^6
1GiB = 2^{30} B ≈ 10^9
1TiB = 2^{40} B = $(1024)^4$ B ≈ 10^{12}

n	2^n	n	2^n
0	1	7	128
1	2	8	256
2	4	9	512
3	8	10	1024
4	16	11	2048
5	32	12	4096
6	64	13	8192
		14	16384
		15	32768

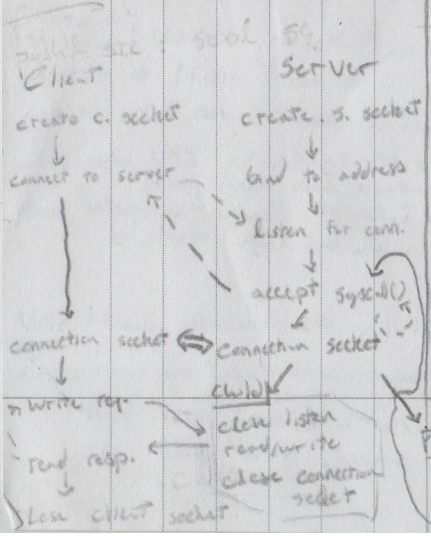
C

- 32-bit ⇒ sizeof(xptr) = 32 bits = 4 bytes
- char *q; in read-only static
- char b[1]; in stack

```
struct point {int x; int y;}
struct point* p
p->x = 1; // OK!
p->x = 2; //SEGV
```




```
struct list_data {
  char * name;
  struct list pl_list; }
struct pl_node {
  int value;
  struct list_elem elem; }
int pl_sum(struct list_data* data){
  struct list *x = &data->pl_list;
  for(struct list_elem *k = list_begin(list);
      k = list_end(list);
      k = list_next(k)) {
    struct pl_node* temp =
      list_entry(k, struct pl_node, elem)
    total += temp->value
  } return total; }
```



x86

Syntax: %register, $const
offset(base, index, scale)
↔ base + idx*scale + offset

Calling Conventions:

Callee Prologue:
1. Save needed GPRs to stack
2. Push args reverse-order (first = lowest address); + padding prior for 16b-align
3. Push rt address (next instruction) and jump to callee ('call')

Epilogue:
1. Pop return value
2. Restore GPR

Caller Prologue:
1. New Frame: push ebp, set ebp to new esp
2. Allocate space for local variables (decrease esp)

Epilogue:
1. store return val in eax
2. De-allocate local variables: set esp to ebp
3. Restore caller's ebp from stack
4. Jump to caller's rt address

pushl src	subl $4 %esp movl %src %esp
popl dst	movl %esp dst addl $4 esp
Call addr	pushl %eip jump addr
push rt + jump	
leave	movl %ebp %esp
restore caller's frame	popl %ebp
ret	popl %eip
pop + jump to rt on stack	

	Process	Thread
Creation	Fork	p_thread_create
page table	distinct	same
registers	distinct	distinct
Stack	separate; inaccessible	separate; accessible
Heap + static	separate	shared
File Descriptors		

- Switching threads faster as they share address space

Thread States:
1. Running. Currently in CPU
   - Spin lock (busy waiting)
2. Ready. Eligible, not running
   - Just Finished I/O
3. Waiting/Blocked. Ineligible to run
   - Linked on wait que; sleeping on CV

Sockets two-way

Pipes one-way
Fd pts to buffer in k. mem

Dual Mode: Enforced in Hardware
- Swap via syscalls, interrupts, & exceptions (external, sync) (internal, sync)
- User can't change table ptr, disable interrupts, access k. mem, or interact w/hardware directly

Lock #2 w/o int. ds. short busy wait

```
int guard = 0;
int lock = FREE;
Acquire(int* lock){
  while(test&set(guard));
  if (*lock == BUSY) {
    sleep & guard=0;
    // guard=0 on wake rec
  } else{
    *lock = BUSY
    guard = 0
  }
}
Release(int* lock){
  while(test&set(guard));
  if (non-empty wait que){
    wakeup
  } else {
    *lock = FREE;
  }; guard = 0;
}
```

#4

```
typedef enum{UNL, LOC, CONT} Lock;
Acquire(Lock *lock){
  if (cmp&swp(lock, UNL, LOC))
    return;
  while (swap(lock, CONT) != UNL){
    Futex(lock, WAIT, CONT)
  }
}
Release(...){
  if (swap(lock, UNL) == CONT){
    Futex(lock, WAKE, 1);
  }
```

unnecessary syscalls
→ No overhead if uncontested!

#3 bool maybe=false → syscall Free when uncontested

```
int lock = 0;
Acquire(int *lock, bool* maybe){
  while (test&set(lock)){
    *maybe = true
    Futex(lock, WAIT, 1);
    *maybe = true;
  }
}
release(..., ...) {
  *lock = 0
  if (*maybe) {
    *maybe = false;
    Futex(lock, WAIT, 2)
  }
}
```

Sync. Lock-less

```
Leave note A
while (noteB){ pass }
if (noMilk) { buy MILK }
rm noteA
```

Thread A busy-waits

```
leave noteB
if (no noteA) {
  if (no Milk) { buyMilk() }
}
rm noteB
```

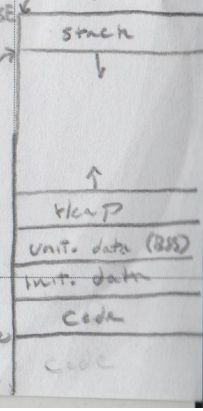
Thread 2

Lock #1
Non-busy: doesn't hog interrupts

```
Acquire() {
  disable interrupts
  if (value == BUSY){ sleep() }
  else { value = BUSY; }
  enable interrupts
}
Release(){
  disable interrupts
  if (non-empty wait que){
    pop; give to ready que
  } else { value = FREE; }
  enable interrupts
```

add to wait que and enable int. will re-enable on return

- Can't be used by users
- Not great w/multiprocessors; disable int scales O(n)



AMAT $:= p_H \cdot t_H + ((1-p_H)(t_H + t_m))$ ($1-p_H = p_m$)

$= p_H t_H + (p_m t_H + p_m t_m)$

$= (p_H + p_m) t_H + p_m t_m$ ($p_H + p_m = 1$)

$= t_H + (1-p_H) t_m$

asm working set fits in DRAM
e.g. no disk ops. required

$\{Max(t_1, t_{TLB}) + p_1(t_2 + p_2 t_m)\}$ (check in parallel)
$+\{p_{TLB}(t_{TLB} + 2(t_1 + (\text{"})))\}$

Get VPN:PPN for TLB
Then retrieve actual data
(possibly cached)

c/w,

$\{t_{TLB} + (1 - p_{TLB} p_F)[t_1 + p_1(t_2 + p_2 t_m)]\}$ (in TLB ⇒ no fault possible)
$+\{p_{TLB}(2[\text{"}] + p_F(t_1 + t_2 + t_m + t_D))\}$

FS

- inode_disk; stored in buffer cache (not inode)
  - ↳ Holds ptrs to data
- inode, in memory, holds sync primitives

For doubly indirect:

max_file = blk_size (bytes) × [#indir. + $\frac{blk\_size}{4}$ + $(\text{"})^2$]

Distributed Sys

As opposed to central server for many clients, we desire no central coord (all-way communication)

- Scalability - Add/rm resources dynamically
- Transparency - mask complexity behind interface

General's Paradox: Impossible to confirm last msg went through (confirm ack. simultaneously)

↳ Motivates distributed transaction:
2+ machines agree to (not) do something in an atomic manner

Two-Phase Commit: Delegated coordinator decides if all process should commit or abort a transaction

↳ Log ensures guarantee of atomicity and allows recovery (log on disk)

- Blocking recovery coord, motivates 3PC & PAXOS

Byzantine G.P.: One general, n-1 lieutenants where f may be malicious

Integrity Constraints:

1) All loyal lieutenants obey same order
2) If general is loyal, all follow his order

- Can't solve n = 3, need n > 3f

Queuing

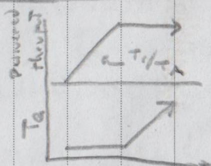
$\mu = 1/T_s$: service

$\lambda = 1/T_A$: arrival

$u = \lambda/\mu$: utilization $= T_s/T_A$

$C = \sigma^2/T_s^2 = Var(x)/E[x]^2$

= 0 if periodic, deterministic
= 1 if memoryless, poisson process
≈ 1.5 in Filesys locality



$T_Q = T_s \times \frac{C+1}{2} \times \frac{u}{1-u}$

↳ Avg time in queue

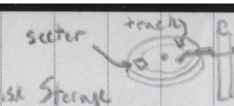
Little's Law: Avg jobs in sys

$L_Q = \lambda \times T_Q$

= arrival rate × response time

↳ Asm stable state

Storage



HDD: Magnetic Disk Storage

- Outside r/w faster
- Unit of transfer is blk (i.e. one sector)
- Overlapping tracks have greater density

Req. Time = $T_Q + T_C + T_D$

T_Q : Device driver software queue

T_C : Hardware controller

T_D : Disk; seek + rot. + transfer

↳ same track ⇒ no seek | $\frac{1}{2} \times 1/RPM$

- Hardware checksum detects bad sectors
- Scheduling FIFO has terrible seek times, ideally shortest seek and rot time (possibly leads to starvation)

↳ E.x. SCAN elevator algo to find closest request in direction of travel

SSD: Flash storage devices

- No moving parts; low power/light
- Operate on page-size chunks, can't address byte-level
- Sequential/random reads are same speed
- Writes handled in-order, must first erase numerous blks

↳ Controller manages free list

↳ Writes ≈ 10x reads; erase ≈ 10x writes

- Each blk has finite life of erases
  wear leveling managed by...
- Flash Translation Layer (FTL): maps OS virt. blk # → phys pg #

↳ Can implement copy-on-write

Effective BW per op: $\frac{\text{transfer size}}{\text{response time}}$

x = # ops
B = time per op
S = overhead

$= x / (S + x/B)$

$= B / (1 + SB/x)$

I/O

Bus: Common set of wires for communication among devices, protocol for data transfer

- $O(n^2)$ wires for n devices, as $n \to \infty$ capacitance $\to \infty$ (slower)
- Communication monopolizes channel
- Ex: PCIE bus - Fast serial channels (parallel limited to slowest device)
  - ↳ Account for vary access times/patterns
- CPU ↔ Device controller communication
  - ↳ Set of registers on device, possibly memory buffer
- Programmed IO, uses cycles ∝ data size
  1) Port-mapped IO: in/out instr
  2) Memory-mapped IO: load/store instr
     - ↳ Appear in physical addr space, can protect w/ translation
     - ↳ Ex frame buffer / setting interrupt vector

- Data Granularity
  1) Block: open/read/write/seek
     ↳ Disk drives, DVD Rom, raw I/O
  2) Character: get/put
     ↳ keyboard, mice, USB
  3) Network: socket/select
     ↳ Ethernet/WiFi, Bluetooth

- Direct Memory Access [no CPU cycles used]
  Give controller access to mem. bus, let it transfer data blks directly

↳ Must signal/notify CPU when done
↳ Controller needs DRAM access

1) Interrupt, handles unexpected events well but high overhead
2) Polling. OS periodically checks status register
   lower overhead, but can waste cycles

↳ Use in tandem: interr. first packet, then poll rest