There are 3 types of notes here:

1. Live Tome Lecture Notes
If you see "Week X Lec Y", then those notes were taken live during 2025F lectures.
Tome live lectures were supplementary to the Eyolfson lecture video notes (below).

2. Eyolfson Lecture Video Notes
If you see "Eyolfson Lec X", then those notes I took when watching this playlist.
The Eyolfson lectures were my main source of learning.

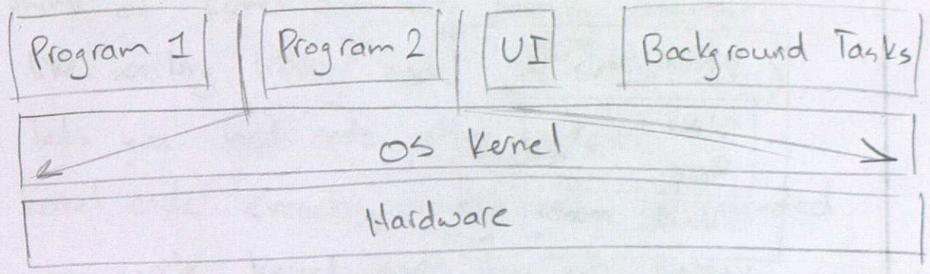
3. Lab Notes

Note: Eyolfson notes and Tome notes are interweaved in this notebook. Please flip over to different topics as needed. Also, all practice pages (for midterm and final prep) have been removed. Also, I didn't write anything about slab allocators.

Week 1 Lec 2

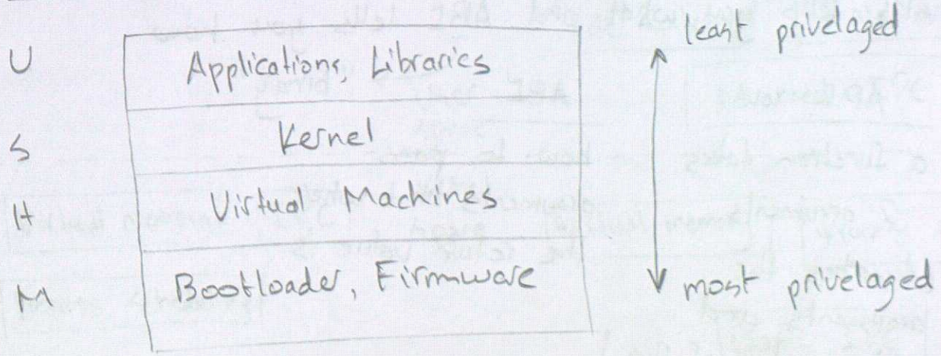
Kernel

- OS manages h/w resources
  - ↳ allows running many programs at the same time
  - ↳ allows programs to access h/w



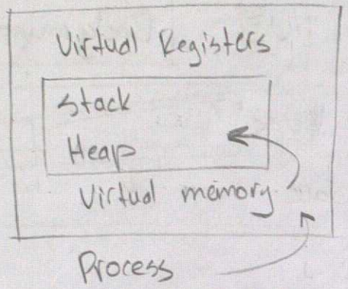
- OS provides illusion that each program is running on its own machine
- Kernel can do privelaged things like access privelaged instructions and access to hardware etc.
  - ↳ eg: only the kernel can manage virtual memory

Diff. CPU Modes



Programs and Processes

- program: a file containing all instructions and data to run
- process: an instance of a running program



- System Calls: transition b/n usermode and kernel mode
  - ↳ operating system application programming interface
- OS API:
  - create/destroy threads or processing
  - allocate/deallocate memory
  - open/close files
- API tells you what and ABI tells you how

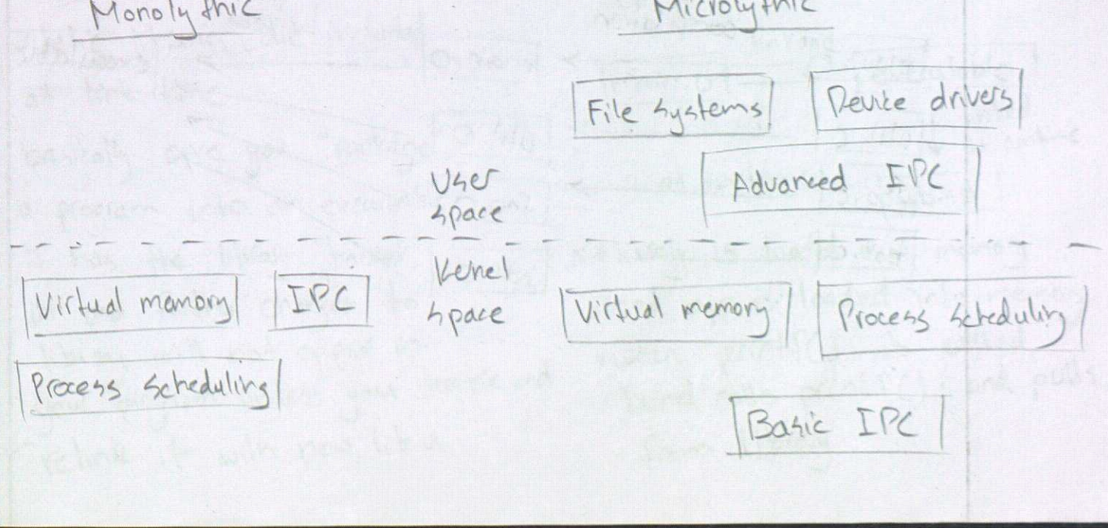
API	ABI
a function takes 2 arguments (describes the arguments and return value of fun.)	how to pass arguments and where the return value is

ABI → "binary"

- Example: programs calls read() system call to read file:
  1. execution goes via library that issues "trap" instr.
  2. trap instr. invokes kernel, which accesses disk
  3. kernel returns to program
- can think of kernel as "long running process"
- more like writing library code (no main func.)
- kernel lets you load code called modules
- your kernel code executes on-demand, when needed
- when you write kernel code, you can execute priveleged instructions.

⇒ • monolythic kernel runs OS services in kernel mode

⇒ • microlythic kernel runs the minimum (smallest) amount of services in kernel mode

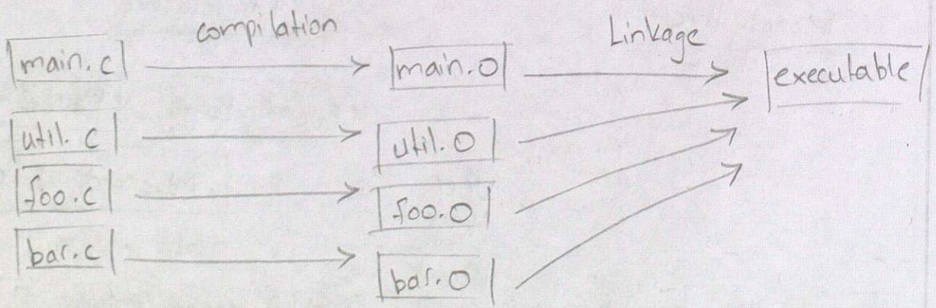


- hybrid kernels, nanokernels, picokernels etc...
- Instruction set Architecture (ISA)
  - ↳ x86-64 (aka amd64): desktops, servers
  - ↳ aarch64 (aka arm64): mobiles, Apple devices
  - ↳ riscV (aka rv64gc): open source ISA

File Descriptor

- IPC: transfer data b/n processes
- file descriptor: can represent a file, terminal, etc.
  - ↳ 0 – standard input
  - ↳ 1 – " output
  - ↳ 2 – error

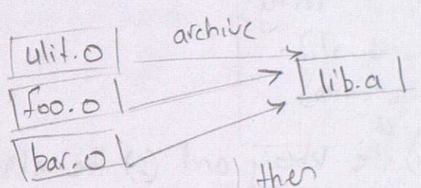
Compilation in C



Libraries

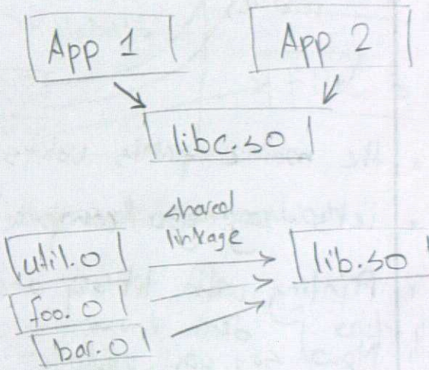
- OS's have kernels and a set of libraries
- Android runs linux kernel, but different Android devices can have different sets of libraries for different purposes
- libraries are needed for doing things alongside a kernel
  - ↳ eg: C Standard Library (libc)
- can have static libraries and dynamic libraries

Static Libraries



- static libraries are included at link time
- basically, once you "package" a program into an executable, it has the library "backed" in and further changes to library will not appear in your program unless you recompile and relink it with new lib.a

Dynamic Libraries

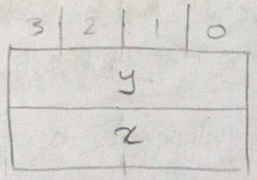


- // at runtime
- library is loaded into memory and app is loaded into memory.
- when "printf()" is called, kernel calls printf(); and pulls from library

☆ go to Eydlson's Lecture

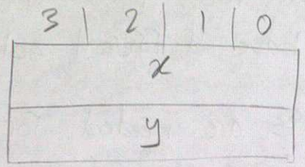
Struct Example

Library V.1




```
struct point {
    int y;
    int x;
}
```

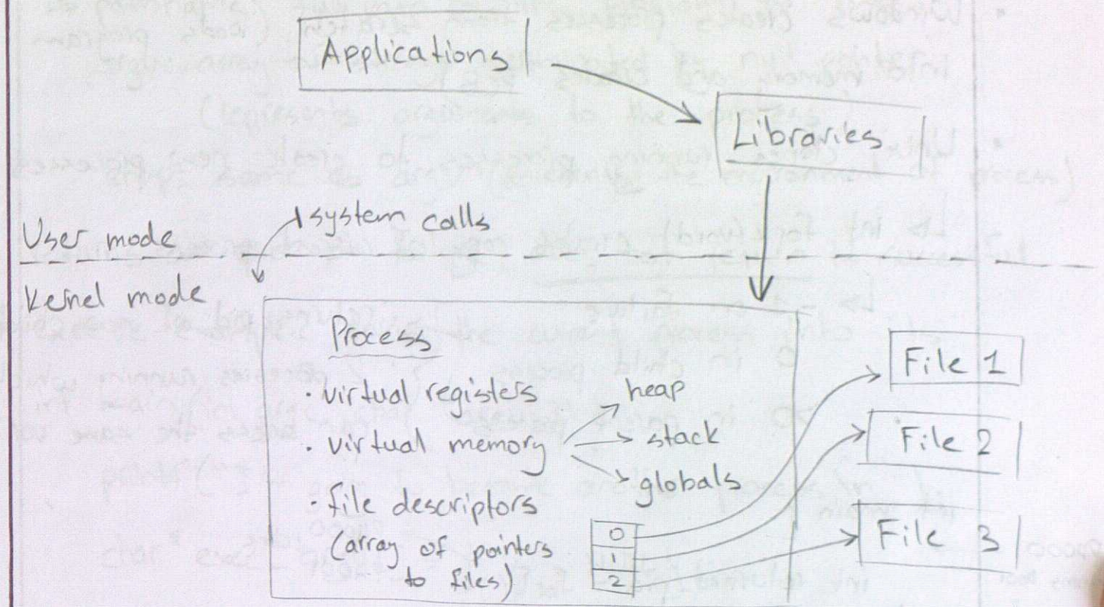
Library V.2




```
struct point {
    int x;
    int y;
}
```

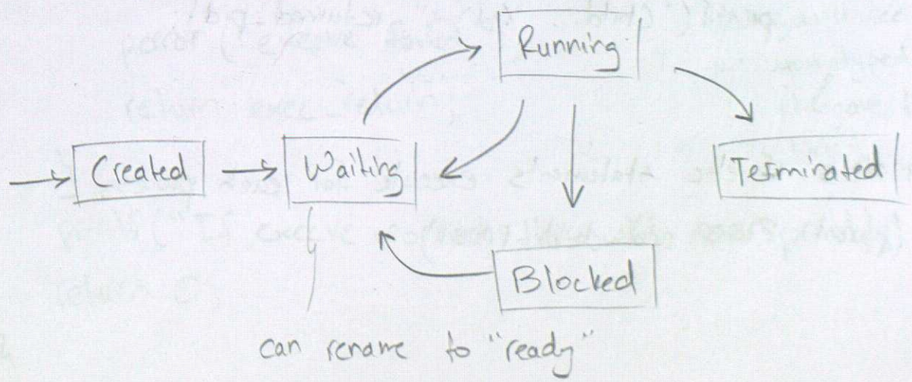
- the main.c prints values using (1) the library and (2) the struct
- Let's say you compile main.c with V.1 of dynamic library
- Printing with library and struct yield same output
- Now, say you update library to V.2 (so you swap x and y) and run main.c → the library printing yields correct value, but struct printing yields the old order printing since it was originally compiled with library V.1
  - ↳ library printing correct b.c it's looked up at runtime

Processes



- A process control block (PCB) contains all information
  - ↳ in linux, task_struct contains
    - process state
    - cpu registers
    - scheduling inforation
    - memory mngmt info
    - I/O status info
    - any other accounting info

  } each process gets an ID "pid"



* you can read process state using "proc" Filesystem
* Windows creates processes from scratch (loads program into memory and creates PCB)
* Unix "clones" running processes to "create" new processes
  - int fork(void) creates copy of current process
    - -1 on failure
    - 0 in child process
    - >0 in parent process

    → returns pid of new child
    → 2 processes running which can access the same var's

PH000 returns 4001 / PH001 returns 0 / (returned_pid btw)

PH000 ↓clone PH001

```
int main {
    int returned_pid = fork();
    if (returned_pid >0) {
        printf ("Parent... %d\n", returned_pid);
        :
    }
    else if (returned_pid ==0){
        printf("Child... %d\n", returned_pid);
    }
}
```

* main's if-else statements execute for each process (parent PH000 and child PH001)

* execve replaces the process with another program
  - pathname: full path of the program to load
    argv: array of strings terminated by null pointer (represents arguments to the process)
    envp: same as argv (represents the environment of process)
* returns an error on failure, does not return if successful
* execve-example.c turns the current process into "ls"

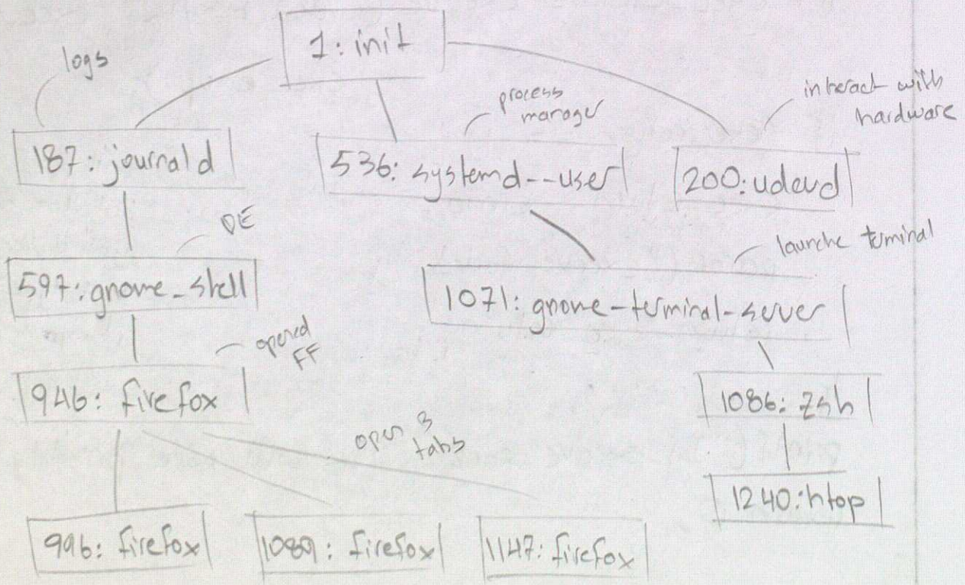
```
int main (int argc, char *argv[]) {
    printf ("I'm going to become another process\n");
    char* exec_argv[] = {"ls", NULL};
    char* exec_envp[] = {NULL};
    int exec_return = execve ("/usr/bin/ls", exec_argv,
                              exec_envp);
    if (exec_return == -1) {
        exec_return = errno;
        perror ("execve failed");
        return exec_return;
    }
    printf ("If execve worked, this will never print");
    return 0;
}
```

will print first line and then on terminal you'll see as if you typed "ls" and it gave the files listed

Eyolfson Lec 5

Process Management

- can read process' state by reading /proc/<PID>/status
  - R: running and runnable [running and waiting]
  - S: Interruptible sleep [Blocked] ← opted out of using cpu
  - D: Uninterruptible sleep [Blocked] ← even if process wanted to run, it can't run (I/O wait eg)
  - T: Stopped
  - Z: Zombie
- On Unix, the kernel launches a single user process
  - ↳ kernel looks for "init" process (or "systemd" in Linux)
  - ↳ must always be active (if it exits, kernel thinks that you are shutting down)



- Use "htop" to see process tree (F5 to toggle list/tree)
- processes are assigned a pid on creation and does not change
  - ↳ eventually, kernel will reuse pid once the original process dies
- the parent process is responsible for its child
  - ↳ parent at minimum must read child's exit status
  - ↳ child exits first (zombie process)
  - ↳ parent exits first (orphan process)
- you need to call "wait" on child processes
- wait_example.c Blocks until the child process exits, and cleans up

```
int main(                ) {
  pid_t pid = fork();
  :
  if (pid == 0) { sleep(2); }
  else {
    printf("Calling...");
    int wstatus;
    pid_t wait_pid = wait(&wstatus);
    if (WIFEXITED(wstatus)) {
      printf("Wait ret...
    }
  }
  return 0;
}
```

because child process waits for 2 seconds b4 returning and exiting...

... parent process must "wait" for child to exit b4 updating info at the address of wstatus and printing

- a zombie process waits for its parent to read its exit status
- an orphan process needs a new parent
  - ↳ child process loses its parent, but it still needs to be assigned some parent process to acknowledge it when it exits
  - ↳ OS reparents orphan process to "init"
- orphan-example.c the parent exits b4 child, init cleans up

```
int main(        ) {
  pid_t pid = fork();
  if (pid == -1) {
    :
  }
  if (pid == 0) {
    printf("child parent id: ...");
    sleep(2);
    printF("child parent id (after sleeping) ...");
  }
  else {
    sleep(1)
  }
  return
}
```

you fork, child exists for 2s while parent exits after 1 s, child gets reassigned to systemd as parent

calling "wait" when you don't have any children throws an error immediately

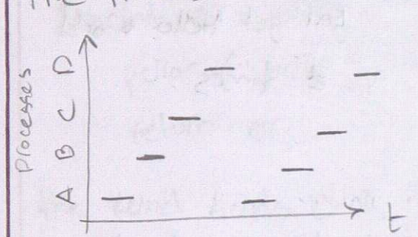
Basic IPC

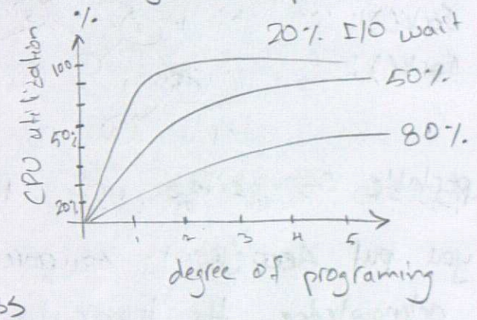
- IPC is transferring bytes b/n 2 or more processes
- assert(bytes_read == bytes_written); ← sanity check
- "read" just reads data from a file descriptor
  - ↳ ∃ no EOF character, kernel returns 0 on a closed file descriptor (so "read" returns 0 bytes read)
- "write" just writes data to a file descriptor
  - ↳ returns # of bytes written (can't always assume successful, so should save errno)
- Standard File descriptors: can close standard input (file descriptor 0) and open a file instead
  - ↳ Linux uses lowest available file desc. for new ones
- your shell will let you redirect standard file descriptors (use <).
- signals are a form of IPC that causes interrupts
- you can set your own signal handlers with "sigaction"
  - ↳ declare a func. without ret. val and takes in signal #
    - 2: SIGINT
    - 9: SIGKILL
    - 11: SIGSEGV
    - 15: SIGTERM

* a signal pauses your process and runs signal handler
* you need to account for interrupted system calls
  ↳ eg: read(); syscall returns -1 if it's interrupted (by a signal for instance as in Lec example)
* you can send signals to processes using their PID
  ↳ use command "kill <PID>"
  ↳ can use "pidof < >" to find pid
* most operations are non-blocking (ie: it returns immediately, and you can use it to check if something occurred)
  ↳ "wait" is currently a blocking call, but you can turn it into a non-blocking call
  ↳ can monitor for changes in non-blocking calls using eigther polling or interupts.

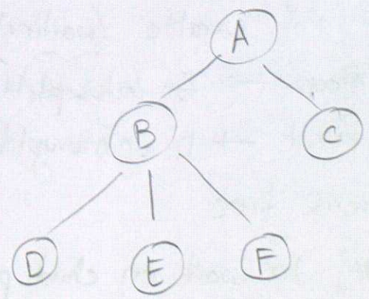
Processes

* if a process is interrupted in the RUNNING state, then we stop it after the currently executing instruction is completed
* Process termination:
  - normal exit (voluntary)
  - error exit (voluntary)
  - fatal error (involuntary)
  - killed by another process (involuntary)
* The Process Model: say we have just 1 processor



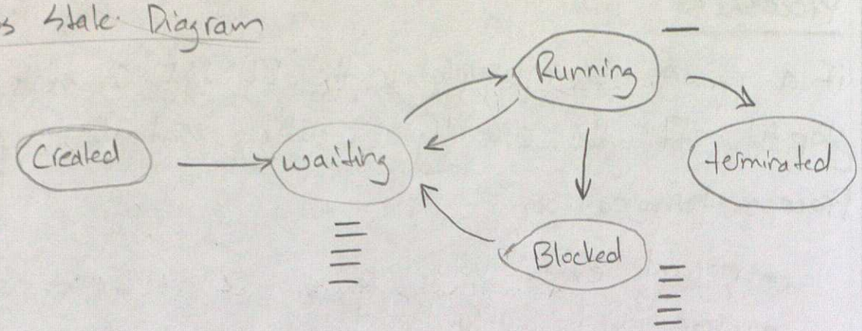


* max cpu usage with fairness
* Process Tree: a process duplicates an instance of itself

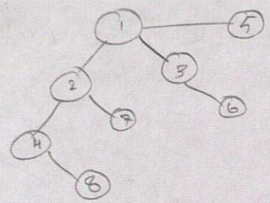


(copy on write means that you only update (or make a full real copy) once the variable is accessed/changed) (otherwise, you just have a pointer to the var in the parent)

Process State Diagram



- parent gets pid of child from fork();
- child gets 0 from fork();
- fork();
  fork();
  fork();



calling fork() 3 times, you get Hello World 8 times

- portable OS interface unix: POSIX
- you put sleep(1000); so parent exists long enough to acknowledge the termination of the child.

Process Management

- Linux lumps "running" and "runnable" (waiting) to say "R"
- process put itself into sleep → S: interruptible sleep
- process waiting for I/O event → D: uninteruptible sleep
- use htop to see process tree
- use wait() or waitpid() to wait on child process

Basic IPC

- Reading to standard input (fd 0)

```
char buff[4096];
ssize_t bytes_read;
while ((bytes_read = read(0, buff, sizeof(buff))>0)
  ssize_t bytes_written = write(1, buffer, bytes_read);
  ⋮
```

read() returns >0 as long as there is something to read

filename < something — something becomes st.in for filename

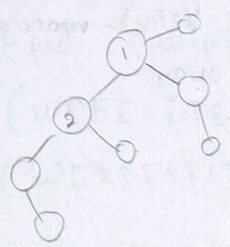
- error number appears in "errno"

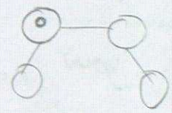
```
int err= errno;
perror ("write");
return err;
```

- the kernel sends 0 to read() ... ∄ a real "EOF" character.
  → man 2 read

Signals:

- kernel's default handler returns exit code: (128 + signal #)





Eyolfson Lec 7 — Sep 13, 2025

Process Practice

* uniprogramming is for old batch-processing OSs
  ↳ uniprogramming: only 1 proc. at a time
  ↳ multiprograming: multiple " " " "
* the scheduler decides when to switch
* the core scheduling loop changes running processes
  1) pause currently running process
  2) save its state so you can restore it later
  3) get the next process to run from the scheduler
  4) load the next process' state and let that run
* cooperative multitasking: processes use syscalls so OS stops it.
* true " : OS retains control and pauses processes
* swapping processes is called a context switch (faster = better)

→ `int pipe(int pipefd[2])`

pipefd: [read end | write end] = [0 | 1]

- one way comm channel b/n 2 file descrip.
- pipe: kernel-managed buffer or array

* can use & in your shell to run a process

* check(); function for errors.
* printf("Child reads: %.*s \n" bytes_read, buffer)
* read() syscall is blocking (waits for input) → blocked state
* do "pidof <name>" to get pid if running.
* close fd as soon as you are done with them
  ↳ put "close(fd[1]);" in child since I know I won't be writing as a child (likewise fd[0] in parent)
  ↳ this way, kernel knows there is nothing coming in the pipe so read() actually returns and child ends after running its course.
* must create pipe before fork so both processes can use/access it.

Waiting for child

* must close write end of pipe or else infinite wait's

```
int wstatus = 0;
pid_t pid = waitpid(child_pid, &wstatus, 0);
check(pid, "waitpid");
assert(WIFEXITED(wstatus));
assert(WEXITSTATUS(wstatus) == 0);
```

* don't mess with standard error (fd 2)
  ↳ that's why you use perror(" ") instead of printf("");

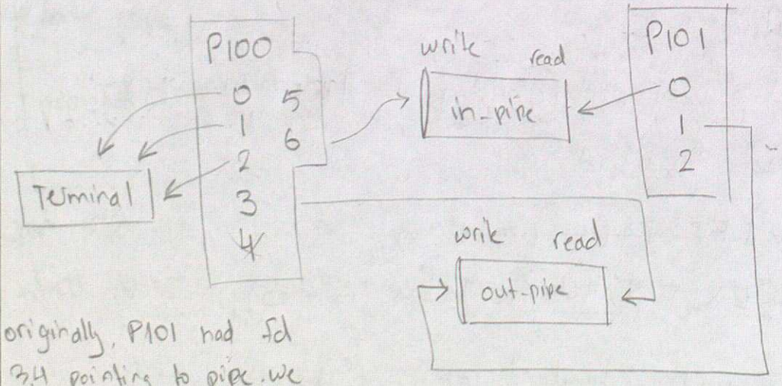
Subprocesses

- execve(); starts running a new program
  - ↳ needed to know exactly the full path of program

→ int execlp(const char *file, const char *arg ...);
- ↳ just give it the name and it finds path itself
- ↳ can give C-strings directly

→ int dup(); ← gives you new lowest fd
int dup2(); ← lets you choose fd

- make sure to initialize things (like pipes)



- originally, P101 had fd 3,4 pointing to pipe. we repointed using dup2() and closed them, just like fd 4 in P100

Basic Scheduling

- preemptible: can be taken away and given to someone else (eg: cpu) through scheduling
- non-preemptible: cannot " " " " " " " " (eg: disk space, memory) through allocations and deallocation
- dispatcher: low-level context switch
- scheduler: high-level decision making, runs whenever a process changes state.
- metrics:
  - minimize waiting time and response time
  - maximize cpu utilization
  - maximum throughput (as many proc's as poss.)
  - fairness

1) First Come First Served (FCFS)
2) Shortest Job First (SJF) → can reduce wait time, but you don't know burst time, and you can have starvation
3) Shortest Remaining Time First (SRTF)
4) Round Robin (RR)

Waiting Time of process = (endtime − starttime) − burst time
(endtime − starttime): how long it was around for; burst time: how long it was executing for

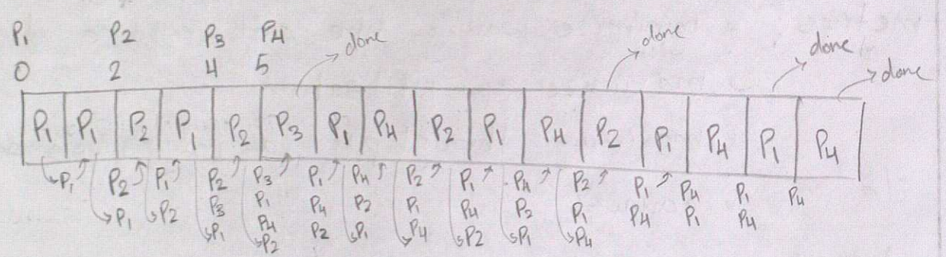
- blocked time ← if ∃ a blocked-queue

Response Time of a process = execution start time − arrival time

\# of context switches = # times you go from one process to another

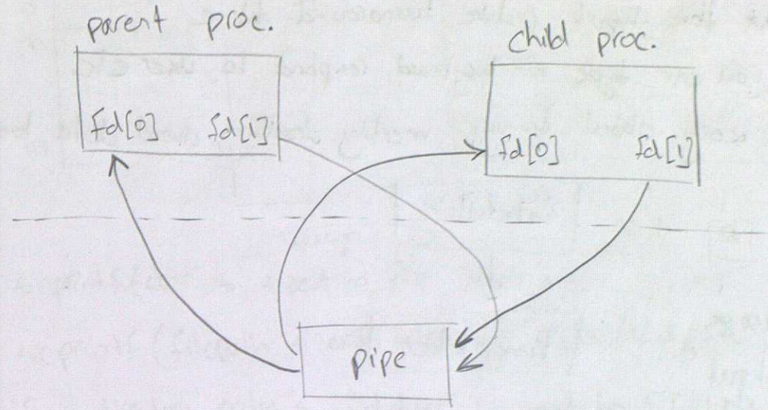
Process	Arrival	Burst
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Time Slice = 1



- RR performance depends on quantum length and job length
  - too low → too many context switches
  - too high → basically FCFS

Pipe:



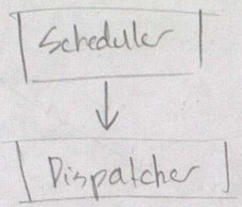
- close fd of pipes once you're done using them
- dup() and dup2() let you change fd's by redirecting what they point.
- "atomically" means it will do those steps no matter what (no intrpts).
- make sure to initialize to zero: `int in_pipefd[2] = {0};`

Week 3 Lecture 3 — Sep 19, 2025

Categories of Scheduling Algorithms

- batch : max throughput, reduce turnaround time
- interactive : you can type on keyboard, respond to user etc.
- real time : u worry about timing, meeting deadlines, avoid data loss

Metrics
- wait time
- max cpu usage
- max throughput
- fairness



- for non-preemtive, scheduler only decides what to run next
- for pre-emtive, scheduler can also decide when to stop running a process in addition to deciding what runs next.

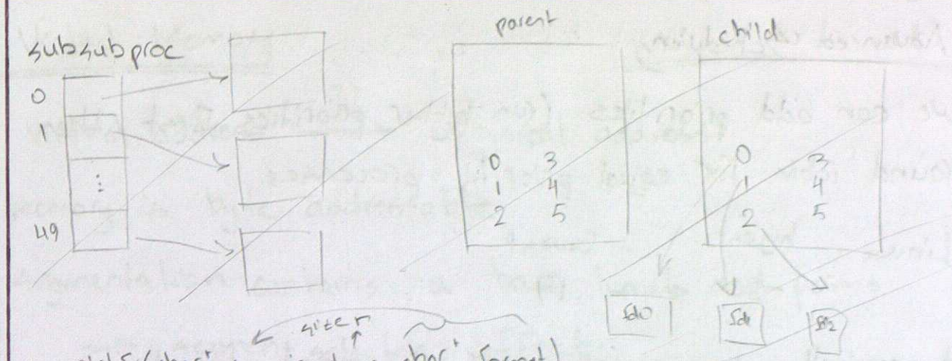
Lab 2

- when looking for file descriptors in /proc/<pids>/fd you must check d_type to be DT_LNK since fd directory also contains "." and ".." (to filter those out).
- execvp turns current proc to new proc, so we must first fork and then turn into new proc. (after fork, close parent fd's)

ssp_proc
- ssp_id: int
- ssp_pid: pid_t
- name: char*
- status: int
- ssp_ppid: pid_t
- child: ssp_proc:

create an array of pointer-to-ssp_proc of size 50 (and make it bigger if it reaches closer to 100) ← later.

Lab 2



snprintf(char* s, size_t n, char* format)

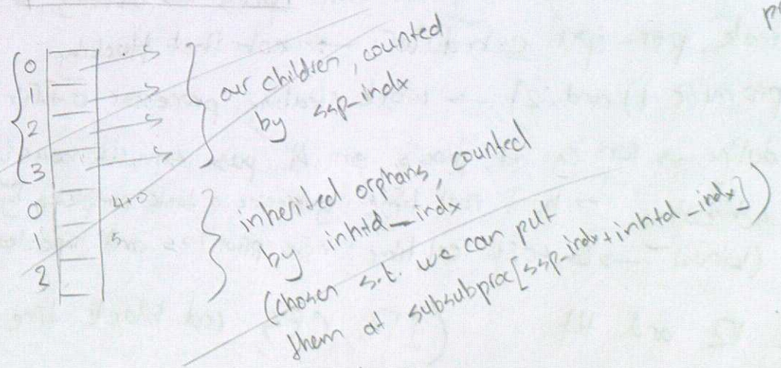
snprintf(filepath, sizeof(filepath), "/proc/%d/fd", (int) pid)

- if a directory entry's "d_type" is set to DT_LNK, printing that as an int yields "10" (the "." and ".." were outputting as 4)
- when you readdir() from a directory you opendir()'d, it creates an internal fd that you should not close. You find it with dirfd(--);

ssp_id:
pid:
name:
status:

- to know which child exited, you can call wait(...) inside <SIGCHLD handler.
  - ↳ look at WIFSIGNALED and others to know which signal caused child to exit.

→ maybe make it pretty and add comments.



just create a separate global array.

Eyolfson Lecture 10 — Sep 20, 2025

Advanced Scheduling

* we can add priorities (run higher priorities first, then round robin for equal priority processes
* Linux: higest lowest
  -20 ⟷ 19
* dynamically change priorities (to avoid starvation)
* priority inversion (caused by dependency
* foreground and background proc's
  response time throughput
* foreground: RR
  background: FCFS
  } RR b/n queues ← can use priorities for each queue
* SMP: symmetric multiprocessing: (all cpu's connected to same physical memory)
  one cpu has to wait for another to finish schedule

1) * one approach: same scheduling for all cpu's → blocking
2) * we can create per-cpu scheduler → not that blocking
3) * can compromise 1) and 2) → work stealing, processor affinity.
4) Gang scheduling → run eg: 4 proc's on 4 cpus simultaneously
5) real-time scheduling → hard real time: guarantee a task completes by deadline
   (Linux) → soft real time: have priorities and hope deadline is met

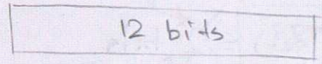
→ Slide 12 and 14 (IFS, CFS, red-black tree)

Eyolfson Lecture 11 — Sep 21, 2025

Virtual Memory

virtual address —??→ physical address

* memory is byte addressable
* segmentation contains: a base, limit, and perms
  → segment selector : offset
* divide memory into fixed-sized chunks (blocks)
* MMU: memory management unit: (translates virt. add → phys)
  ↳ can divide mem into fixed sized pages (eg 4096 bytes)
  ↳ page in virtmem → page
  page in physmem → frame
* offset tells you which byte of the page you want

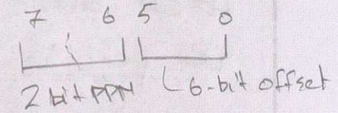


Page Table:	VPN (virt pg #)	PPN (phys. pg #)
page size is 4096 bytes so 2^{12} bytes and 12 bits rep. by 3 hex dig.	0x0	0x1
	0x1	0x4
	0x2	0x3
	0x3	0x7

a) virt 0x0ABO → phy 0x1ABO
(0x0 = VPN index, ABO = offset; 0x1 = PPN, ABO = offset)

b) virt 0x1FAO → phys 0x4FAO

Ex: 8-bit virtual address
10-bit physical address
64-byte page size → $2^6 = 64$, so 6-bit offset



a) how many virtual pages are there?
8-bit virt addr → 8 − 6 (from offset) = 2 bits
∴ $2^2 = 4$ virtual pages

b) how many physical pages? 10 bits − 6 bits = 4 bits
4 bits to address PPN ≡ $2^4 = 16$ physical pages

c) how many entries in the page table
page table entries keep track of a virt page to a physical page ⇒ since 4 virt. pages to keep track of ⇒ ∴ 4 page table entries

Page table: [0x2, 0x5, 0x1, 0x8] PPN's
VPN: 0x0 0x1 0x2 0x3

virtual | 2 bits | 6 bits | (VPN | offset)
physical | 4 bits | 6 bits | (PPN | offset)

virt. addr. 0xF1 → Page table [VPN] ⇒ P.T[0x3] = 0x8

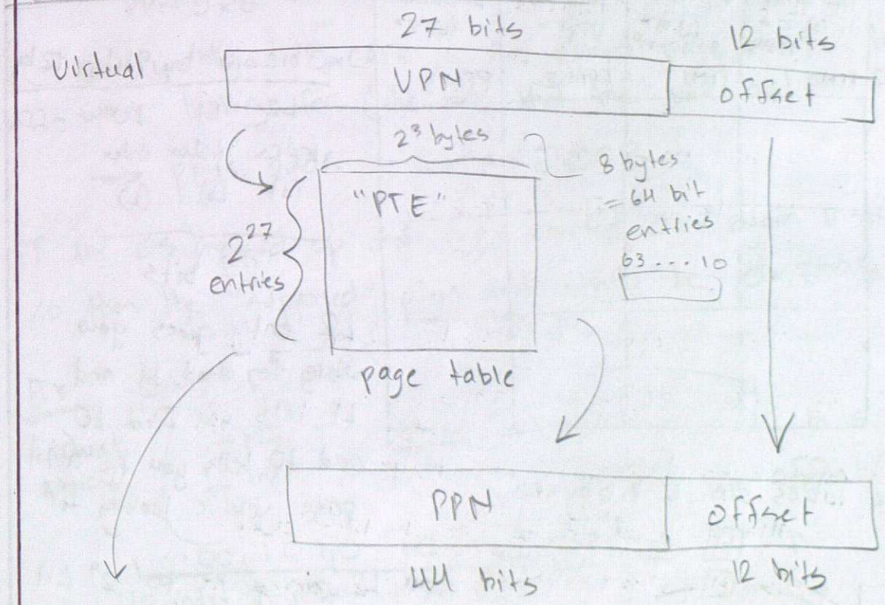
1 1 1 1 0 0 0 1
VPN = 3 | offset

1 0 0 0 | 1 1 0 0 0 1
PPN = 0x8 | offset

⇒ ∴ virt 0xF1 → phys 0x231 = 0x231

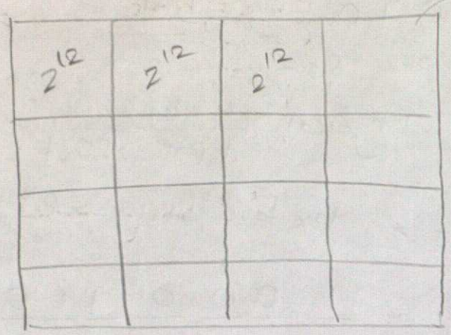
Eyolfson Lec 12: Page Tables

Sep 24, 2025



$2^{27} \times 2^3 = 2^{30} \Rightarrow$ 1GB

2^{10} — 1KB
2^{20} — 1MB



Physical Memory

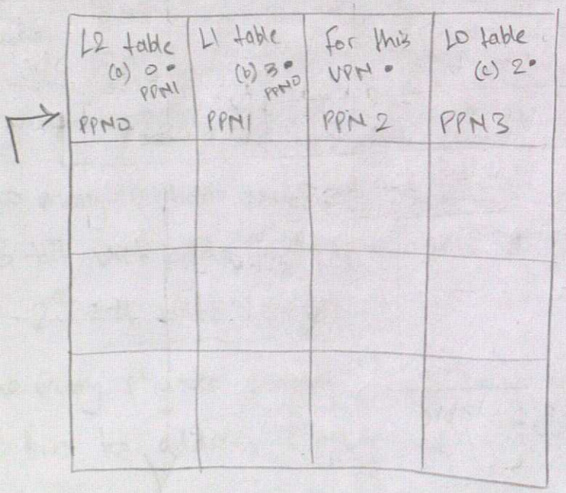
each page is 4096 bytes

PTE (page table entry). size is 2^3 bytes = 64 bits

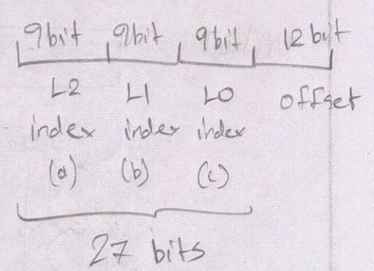
entries we can fit inside 1 page table

$\frac{2^{12}}{2^3} = 2^9$.

* we can make page table fit in a 4096 byte page (but then virtual address can only address 2 megabytes)

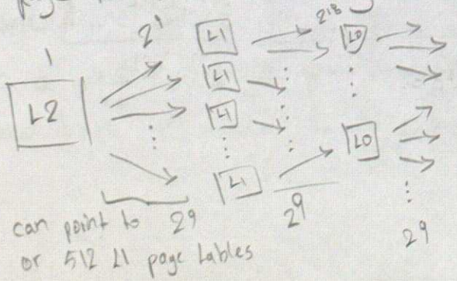


Virtual address



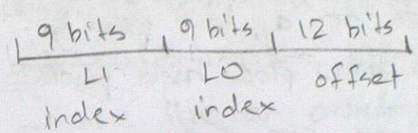
L2 entry gives you where to find L1, and L1 lets you find LO, and LO tells you the actal page you're looking for

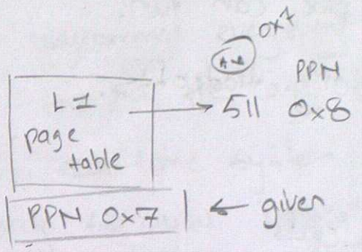
3 page tables are 12 kilobytes

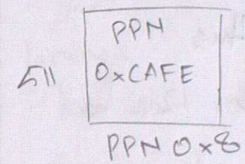


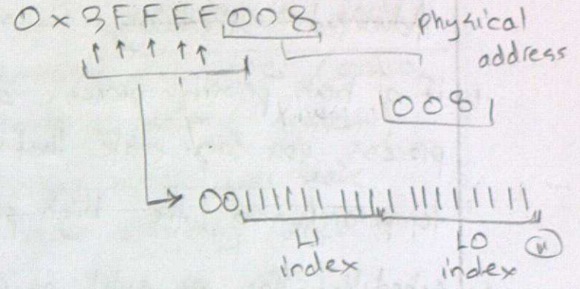
an L2 can point to 2^9 L1 page tables, and each of those can point to 2^9 LO page tables, which can in turn point to 2^9 page tables

Ex: virtual address









① go to L1 p.t. and see what's at entry 11111111 = 511 → where LO p.t is, so go 0x8

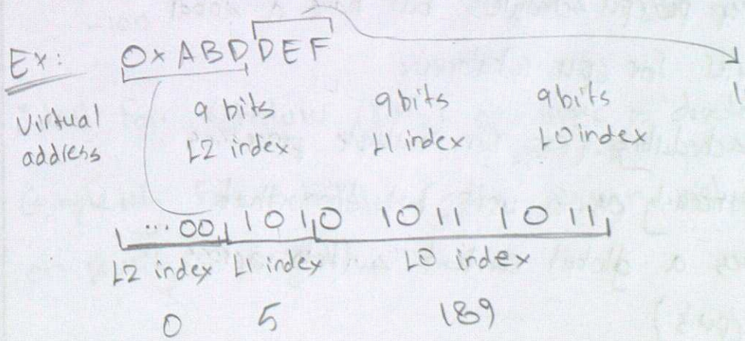
② go to LO p.t. and check whats at entry 11111111 (which I got from LO index of virt. address), so I go there and see that there is 0xCAFE

③ now we know 0xCAFE is the phisical page number that we're looking for (the one that the virt. address translates to)

→ phys. addr. = 0xCAFE008

if the OS messed up and ⑪ pointed 0x7 again instead of 0x8, so then the "supposed" phys. addr. would be 0x7008, which is wrong.

Ex: 0xABDDEF

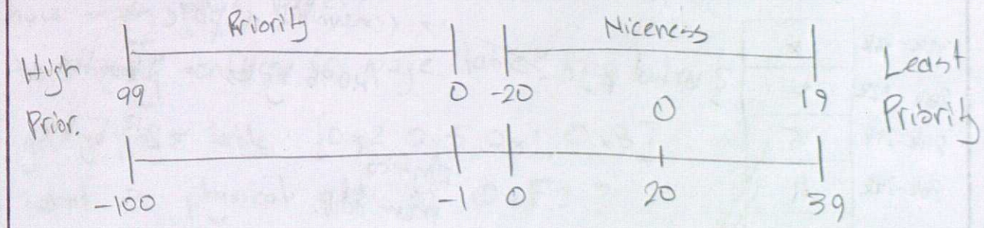


Week 4 Lecture 1 — Sep 23, 2025

Advanced Scheduling

- if a high priority process depends on a low priority process, you can make that low prior. proc. high prior. temporarily so the high prior. proc. can run.
- scheduling has no right answer, only tradeoffs.
- symmetric multiprocessing
  - 1) Use one scheduler for all cpus (not scalable since system blocks on this global scheduler, which halts everything to perform a context switch) (also poor cache locality)
  - 2) Can have per-cpu schedulers (can have load imballance since once a process is assigned a cpu, it basically stays stuck to it)
  - 3) Can keep per-cpu schedulers, but have a global scheduler for cpu rebalancing.
  - 4) Gang scheduling (can run multiple processes simultaneously as a unit, but then that requires a global context switch across all cpu's)

- A hard real-time system: must guarantee a task completes within a certain amount of time (audio, auto-braking system, missile guidance/defence system)
  - → more like an embedded system, where you have programmed everything and know everything (which is not the case for a gen. purpose OS)
- Soft real-time system: give critical purposes higher priority so you basically hope it finishes on time
- Linux:
  - soft real-time proc: schedule higher priorities first → SCHED-FIFO or SCHED-RR
  - normal: schedule based on ageing priority → SCHED-NORMAL

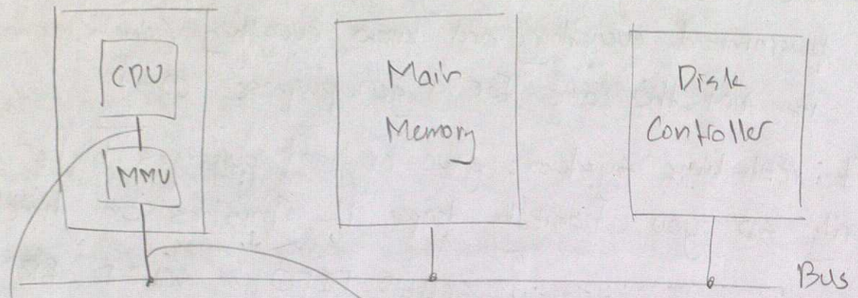


- Ideal Fair Scheduler (IFS): cpu usage is divided equally among all processes.
- Completely Fair Scheduler (CFS): assign virtual runtime (based on priority = weight)

Week 4 Lec 2 — Sep 24, 2025

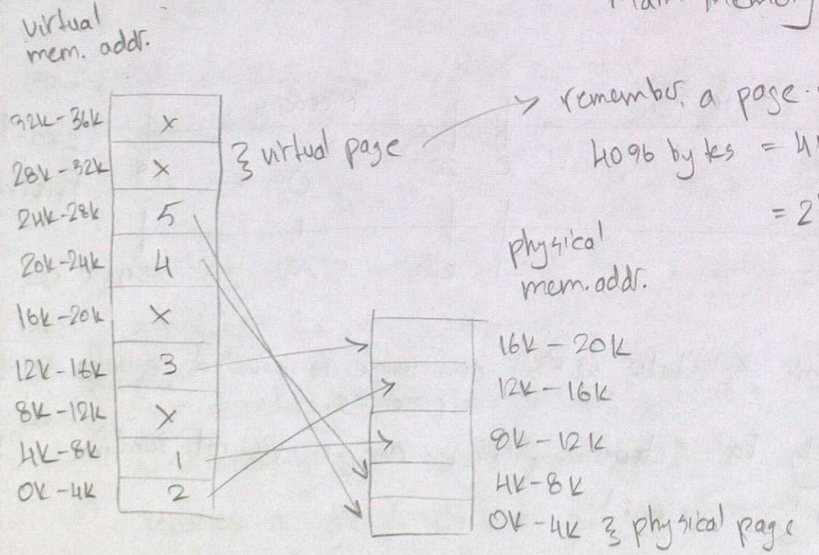
Virtual Memory

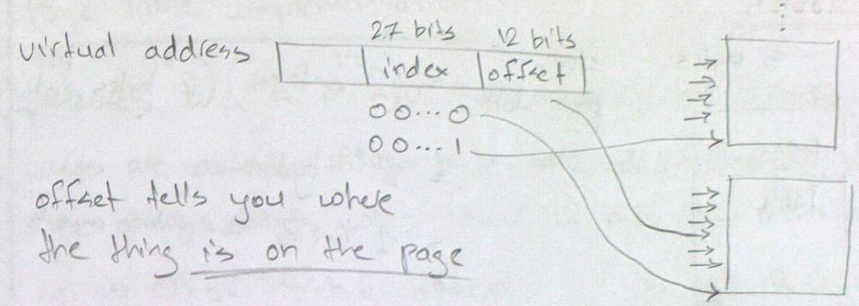
- must free() after malloc(...)



cpu sends virtual addresses to MMU

Memory Management Unit sends physical address to Main Memory





offset tells you where the thing is on the page

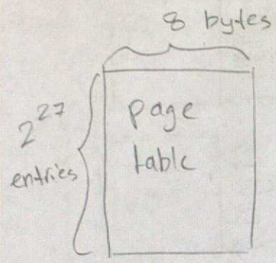
- accessed bit helps you when you want to know what needs to be "reset" back to disk when you have to do, say, a context switch.

Page Translation Example

8-bit virt. addr., 10-bit phys. addr., 64-byte page

a) how many virt. page?
b) how many phys. pages?
c) how many entries in page table?
d) given page table: [0x2, 0x5, 0x1, 0x8]
what is physical addr. of 0xF1?

Page Tables



size of page table $= (2^{27}\text{ entries})(2^3 \frac{\text{bytes}}{\text{entry}})$

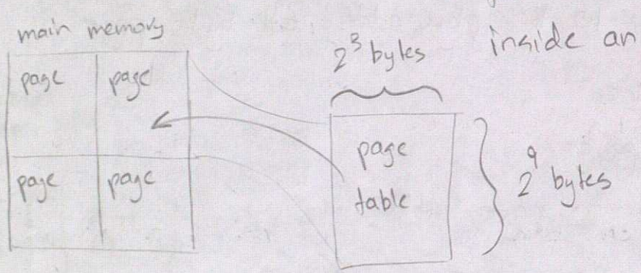
$= 2^{30}$ bytes

$= 1$ GiB ← too much

- most programs don't use all the address space, so how can we leverage this?

page size

$2^{12} = 4096$ bytes ← we will make it so that our page table itself can fit inside an actual page size



- "satp" supervisor address translation protection
- using $2^9 \times 2^3$ page tables allows us to use less "space" if the program is smaller, and we just start using more of the L2, L1, and L0 entries as needed

eg: 1 process (small program) requires at minimum 3 pages ⇒ (3)(4 KiB/page) = 12 KiB and at maximum

$2^{12} + 2^{9+12} + 2^{18+12} = 4$KiB + L2 + L1 + L0 max

Page Table Implementation

- processes use a register (like satp) to set the root page table
- pages are always 4096-byte-aligned in memory, so each page always starts when lower 12 bits of mem address is 0.

eg: is 0xEC 8-byte aligned? → not 8-byte aligned

0xEC = 1 1 1 0 1 1 0 0

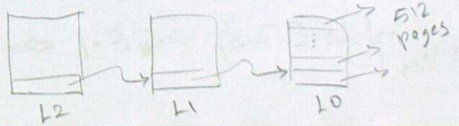
How Many Page Tables Needed

prog. uses 512 pages, what is min and max page tables?

min:

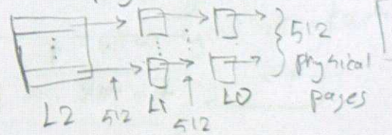
you always have one L2 pointing to at least 1 L1 pointing " " " 1 L0, so you would need one L2 with one entry pointing to one L1, with one entry pointing to one L0, with all entries pointing to a physical page in memory (total 512 pages)

⇒ ∴ need 1+1+1 = 3 page tables



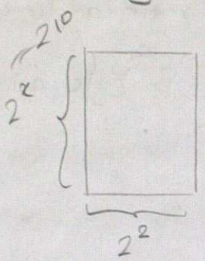
max:

you always have one L2 page table, but say you got unlucky and each entry in L2 is full and points to an individual L1 page table (so 512 L1 page tables total), and each of those L1 page tables only has one entry each pointing to an L0 page table (so total 512 L0 page tables), and each of those L0's have 1 entry pointing to a physical page ⇒ 1+512+512 = 1025 page tables



Example: 32-bit virtual address
4KiB (2^{12}) page size
4 byte (2^2) page table entry (pte) size.

a) How many pte's per page?



$2^x \cdot 2^2 = 2^{12}$
$x = 12-2 = 10$
$\therefore 2^{10}$ entries or 1024 entries/page

b) Virtual Address:

L1 index	L0 index	offset
10 bits	10 bits	12 bits

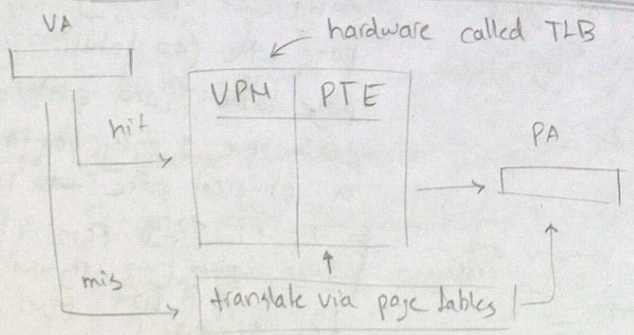
"ceiling" →

* Levels $= \left\lceil \frac{\text{virtual bits} - \text{offset bits}}{\text{index bits}} \right\rceil = \left\lceil \frac{32-12}{10} \right\rceil = \left\lceil \frac{20}{10} \right\rceil = 2$

∴ we'd need 2 levels of page tables (as shown above)

* Using the page tables for every mem address is slow
  ↳ we can use caching

⇒ Translation Lookaside Buffer (TLB) ← caches page table entries



* effective access time (ETA)
  ↳ if you have 3 page table layers, 4 mem access to translate a single VPN or VA (virtal address)
  ↳ yo can add a TLB to reduce that (in case of hit)
    TLB_Hit_Time = TLB_search + Mem

* context switching requires handling the TLB (usually a flush)
  ↳ on x86, loading a new root page table flushes TLB

Eg: ./test-tlb <size> <stride>

./test-tlb 4096 4

you ask for 4096 bytes of memory and the access/request 4 bytes at a time
⇒ due to TLB, we only have cache miss one time at the start and then keep having hits (since we are accessing things on the same physical page and thus only need to translate one VPN → PPN once at the start)
⇒ 1.37ns (≈5.3 cycles)

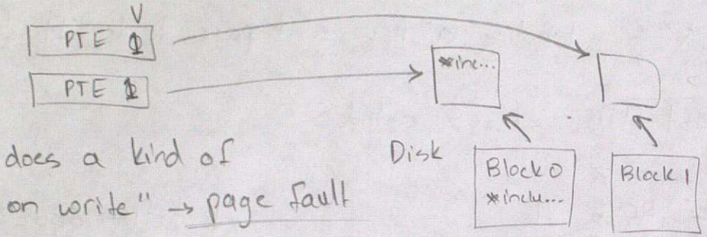
./test-tlb 536870912 4096

this allocates ~512 MiB of memory and you make mem requests every 4096'th byte ⇒ this case leads you to have a cache miss every time because you are guaranteed to request a new page every single time
⇒ 29.43ns (≈ 114.8 cycles)

locality of reference when you program

Priority Scheduling and Memory Mapping

- mmap() or memory map lets you map files to a process's virtual address space
  - ↳ no need to call read() or write(), can access file directly



kernel does a kind of "copy on write" → page fault

- reason why you segfault if you try to modify a string literal (the mmpag in this case is read-only)
- if you change some flags, you can also write to the file without ever calling write()!

Ex: loading 20GB Llama model with only 6.8 GB RAM

$$20\text{ GB} = \frac{20 \times 2^{30}}{2^{12}} = 20 \times 2^{18}\text{ pages}$$

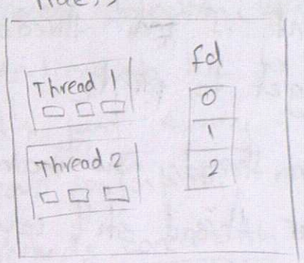
this means we'd need 20×2^{18} entries (pte's)

$$20 \times 2^{18} \times 2^{3} = 20 \times 2^{21}\text{ bytes} \simeq 40\text{ MB}$$

$$\frac{20 \times 2^{18}}{512\text{ entries/L0}} = 10240\text{ full L0 page tables}$$

$$\frac{10240}{512\text{ entries/L1}} = 20\text{ full L1 page tables}$$

- parallel vs. concurrent
  - ↳ talking and drinking concurrent, but not parallel
  - ↳ talking and gesturing parallel
- threads are like processes but with shared memory.
  - ↳ threads have individual (separate) their own registers, PC, and stack, but they have same address space.



- POSIX threads: #include <pthread.h> and do "-pthread" during linking
- pthread_create(...);
- process dies/exits → all threads die/exit.
- wait() equivalent for threads: join
- pthread_join(...);
- exit() for ending thread early
- pthread_exit(...);

- joinable threads need someone to wait on them before they can release resources (default thread behaviour)
- detached threads will cleanup and remove themselves fully (release resources) without needing anyone to call join() on them —— they terminate and release on their own.

⟹ pthread_detach(...);

- but now, thread doesn't necessarily print... if main thread dies, then detached thread doesn't get to print.

↳ can call pthread_exit() in main thread: now, main thread exits, but since ∃ another thread still, your overall process doesn't exit.

↳ if last thread calls pthread_exit(), process terminates.

- threads enable concurrency.

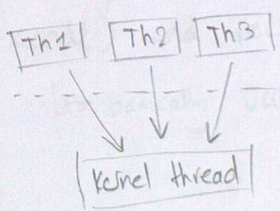
- Multithreading models: user threads and kernel threads
- need a thread table.

User Threads
- very fast to create, no syscalls, no context switches
- one thread blocks, whole process blocks (kernel can't distinguish b/c to its POV, it's just running one of your (potentially) multiple threads concurrently

Kernel Threads
- slower since syscalls
- if one thread blocks, kernel can schedule another b/c now, it's aware that your process has multiple threads.
- allows for parallelism

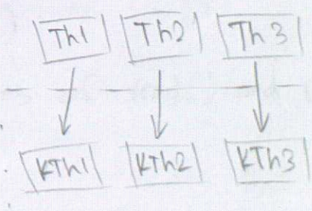
- Mapping user threads to kernel threads

many-to-one



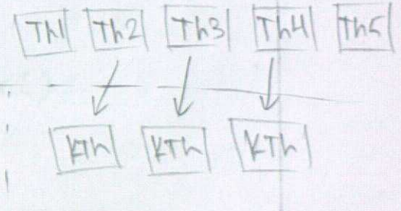
(the case where it's basically all user threads and kernel can't distinguish)

one-to-one



(kernel is fully aware and parallelism is fully achievable)

many-to-many



(some user threads, some kernel threads)

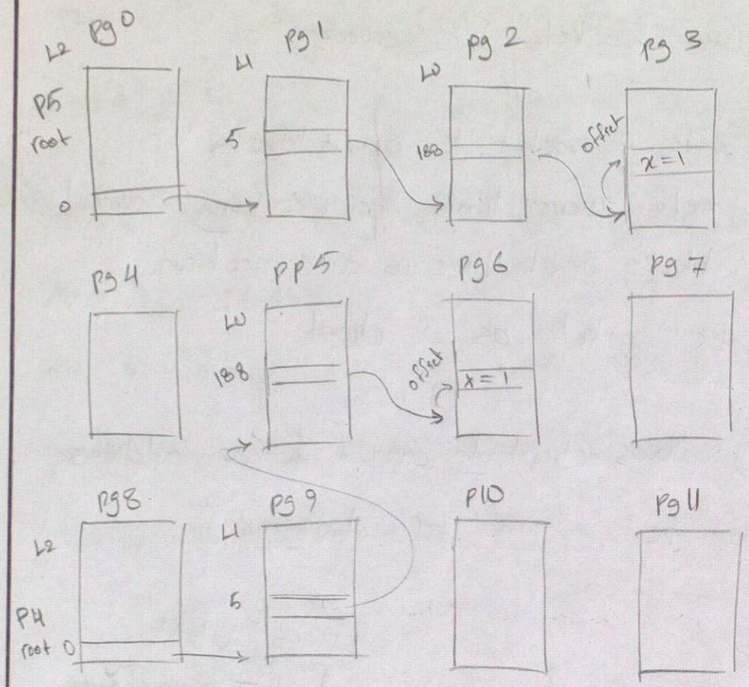
# of kernel threads < # of user threads

* linux picks a thread at random when that process gets sent a signal. (hard: any thread can be interrupted)
* instead of many to one, can use a thread pool
* Ex: 8 threads each incrementing 10,000 times:
  does not necessarily mean final sum = 80,000
  due to "interweaving" and each thread "not getting the most recent value @ counter's mem local"
  - ↳ can potentially solve by join() right after create() ... this sadly means no parallelism.
  - ↳ can use malloc() and then add finally in main thread. → "map-reduced"

Eyolfson Lec 18: Sockets — Oct 5, 2025

* sockets are another form of IPC: pipes, signals, shared mem.
* 4 steps to use sockets for servers
  - ↳ socket()
  - ↳ bind() : sets a socket to an address
  - ↳ listen() : sets queue limit for incoming connections
  - ↳ accept() : blocks until there is a connection.
* 2 steps to use sockets as a client
  - ↳ socket()
  - ↳ connect() : allows client to connect to an address
* socket type can be "stream" or "datagram"
  - "stream": * uses TCP; * reliable, but slow
  - "datagram": * uses UDP; * fast, but messages may be reordered or dropped
* send() and recv()
  - ↳ basically versions of read() and write()

4096 bytes



Process 4

Virtual address

0x ABC123

L2 root page table

0x 8000 address

L2	L1	L0	offset
000...	000101	01111000	000...11

once you fork(), you create P5 with new page table and same variable x and same virtual address "journey"

↳ P5 has root page table 0

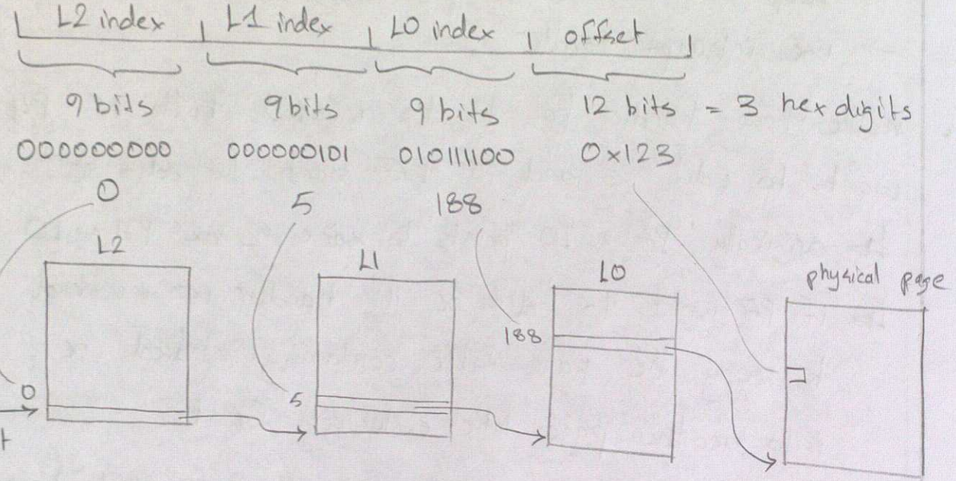
- copy-on-write: data (page) shared until change made by any process
  - → keep track of write permission in L0
  - → use interrupt handler
- notice how handler has to be invoked both if P4 wants to edit x and if P5 wants to edit x
  - ↳ originally, P5's L0 points to same x as P4's L0
  - ↳ if P5 wants to edit x, the handler is invoked to copy the page table containing "shared" x into another page table exclusively for P5
  - ↳ if P4 wants to edit x, handler is called because you realize that something else (aka P5) was pointing to page table that P4's L0 was pointing to (you can't just edit it since then P5 would see edited value of x) → so you do copy-on-write for P4 as well in the sense that you create a copy of said page and change P5's L0 entry to point to that newly copied page table ↓ not P4

  ⇒ you want to preserve the virtual address of the original parent (basically, original parent always keeps original page table).
  - → but you could argue otherwise, for efficiency.

(although a different implementation I used in VM lab does the opposite)

Lab 3 Notes — Oct , 2025

Virtual Address: 0xABC123



* gotta change the pages, but the locations of entries within the pages must stay same → so same virt. addr. works

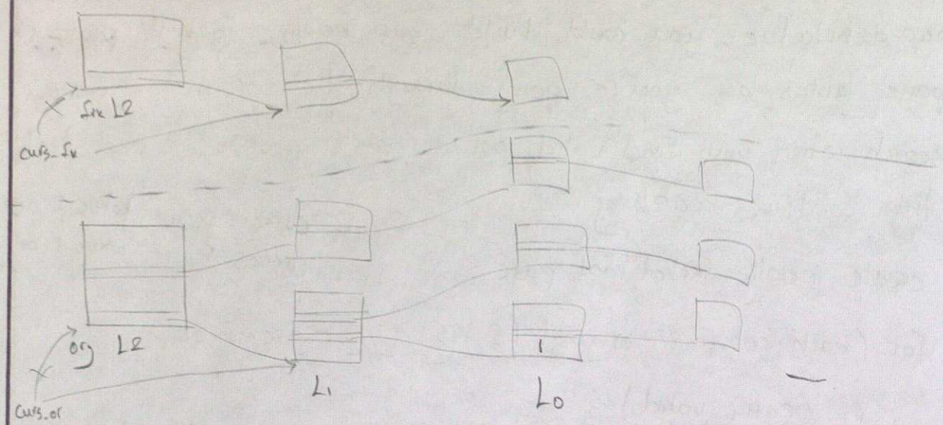
```
uint64_t* valid_tracker[512] = {0};   ← define this struct
          num_valid = 0;
```

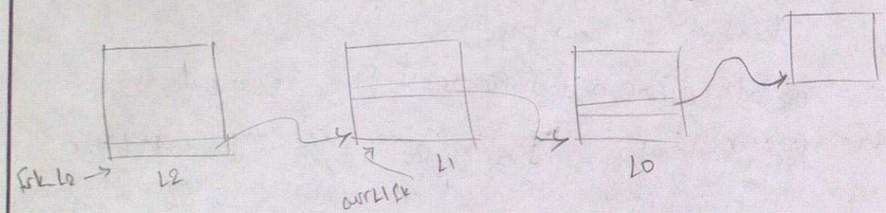
* vms_fork_copy(); flow:

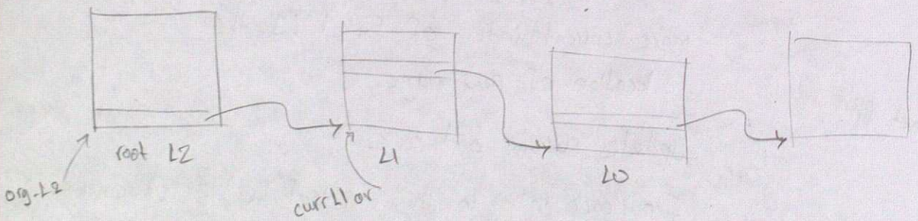
① Find all valid root L2 pte, put into struct L2 array
↳ ② each valid entry contains location of L1 page table, so for each valid pte, go to the L1 location and find all valid L1 pte's, put into struct L1 array
↳ ③ for each valid L1 pte, go to L0 location and find all valid physical pages, put into struct L0 array

* new architecture: you could "build" your newly forked process's page tables as you're going through the original "journey"
* search, when you find valid, pause, create new page and link up, then continue searching...

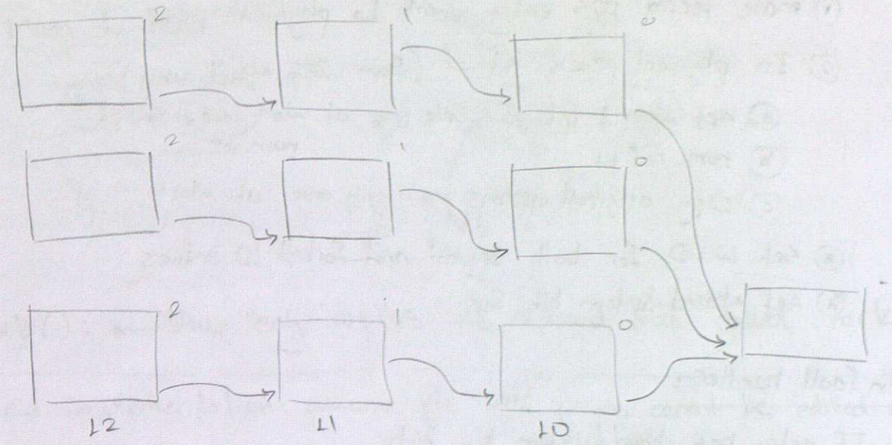
```
create newly forked root page;   ← L2
                                   cursor_o; ← where I am at, original
                                   cursor_f; ← where I am at, forked
for (each entry in original L2) {   VA: 0x7FC0000000
                                        0x0 → to
    if (entry valid) {
        create new page;   ← L1
        make the current index (entry) of forked L2 the
            location of new page;
get ppn → update cursor_o and cursor_f;
        for (each entry in this original L1) {   VA: 0x3FE00000
                                                      0x0 → to
            if (valid) {
                create new page;   ← L0
                make current index of forked L1 the
                    location of new page;
get ppn → update curs_o and curs_f;
                for (each entry in this original L0) {   VA: 0x1FF000
                                                              0x0 → to
                    if (valid) {
                    create new page;   ← get PPN
                    make current index of forked L0 the
                        location of new page; ← set PPN
set V,R,W ←         copy origial page to new page.
                }
            }
        }
    }
}
```



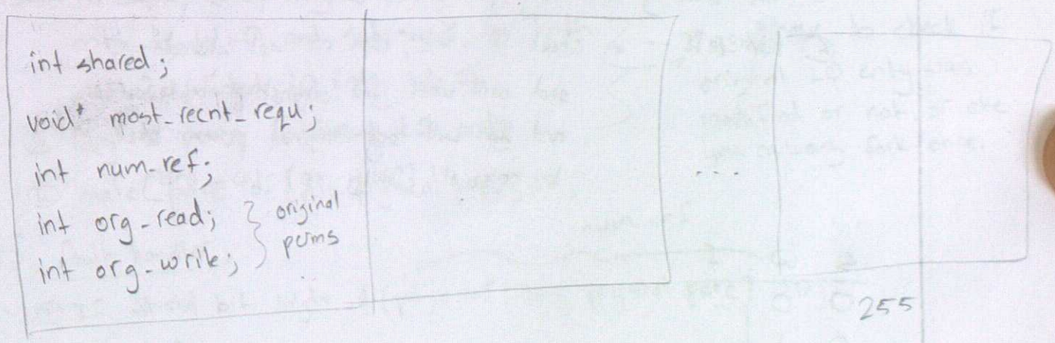




- decision: only faulting process gets a new page table.



- can use custom bit to know if it's shared
- Eyolfson recommends tracking number of "references" made to a page → can use fact that system has 256 total pages.



In fork:

① make forked L0's entry point to physical page
② In physical page's struct (from 256 struct array):
   ⓐ set shared int ← only once at start, non shared if num_ref == 0
   ⓑ num_ref ++
   ⓒ copy original perms ← only once at start
③ set W=0 for both original and forked L0 entries
④ set shared/custom bit for " " " " "

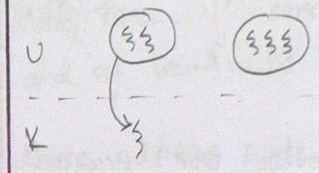
In fault handler:

If entry has shared/custom bit high
① Find physical page's info using struct array
② create new page and make faulting L0 entry point to it
③ copy into new page all bytes from this shared physical page
④ set faulting L0 entry perms ← original perms (saved in struct)
⑤ num_ref-- ← check if == 1, then can set shared = 0
   and will wait till original process faults
   and then will copy original perms back in
   to original (faulting rn) L0's entry

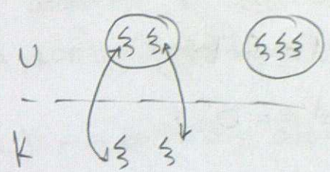
S	W	F
0	0	
0	1	
1	0	
1	1	

Threads:

Many to One



One to One



- fork(): ← linux only copies the thread that called fork():
- New implimentation: assume pte W/R perms cannot be edited after setting
- Reduced to array: page_ref[256] = {0} ← int array

In fork:

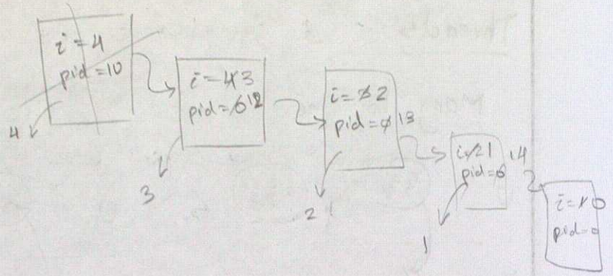
① make forked L0's entry point to original physical page
② if original L0 entry had W=1, set both original and forked L0 entry to W=0 and set S=1 for both
   else, set forked L0 entry W=0
③ Copy R perms to forked L0 entry
④ make page_ref[this physical page]++;

→ you have to check if original L0 entry was modified or not. or else you can only fork once.

In fault handler:

① If shared bit high & (page_ref[this physical page] > 0) ← num_ref
   ⓐ create new page
   ⓑ set faulting entry's PPN to the new page
   ⓒ copy to new page from old page
   ⓓ clear shared bit and set write bit
   ⓔ num_ref--;
② " " & " " == 0
ⓓ ....

```
int main(void) {
    int i = 4;
    while (i != 0) {
        int pid = fork();
        if (pid == 0) {
            i--;
        }
        else {
            printf("%d\n", i);
            exit(0);
        }
    }
    return 0;
}
```



The output won't necessarily be in the same order all the time because the scheduler may decide to run the parent branch or the child, and the child may decrement, fork, and exit before the parent prints.

- atomic operations finish fully once they start
  - ↳ cannot be stopped / preempted
- data race: 2 concurrent accesses to the same variable and at least one is a write operation
- three address code (TAC) → each line atomic
  - ↳ "Gimple" or "TAC" lets you reason code low-level without having to deal with assembly
- counter++ example:

```
D.1 = *pcount;
D.2 = D.1 + 1;
*pcount = D.2;
```

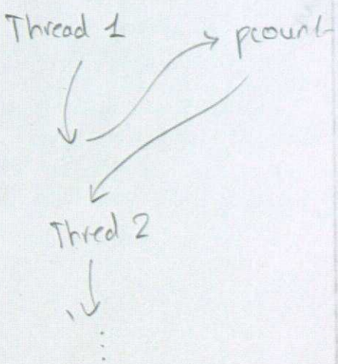
pcount is pointer to our count variable

counter++ is made up of these 3 atomic instructions

*pcount = ∅ 1

Thread 1	Thread 2
D.1 = 0	
	D.1 = 0
D.2 = D.1 + 1	
	D.2 = D.1 + 1
= 1	
	= 1

Correct Behaviour



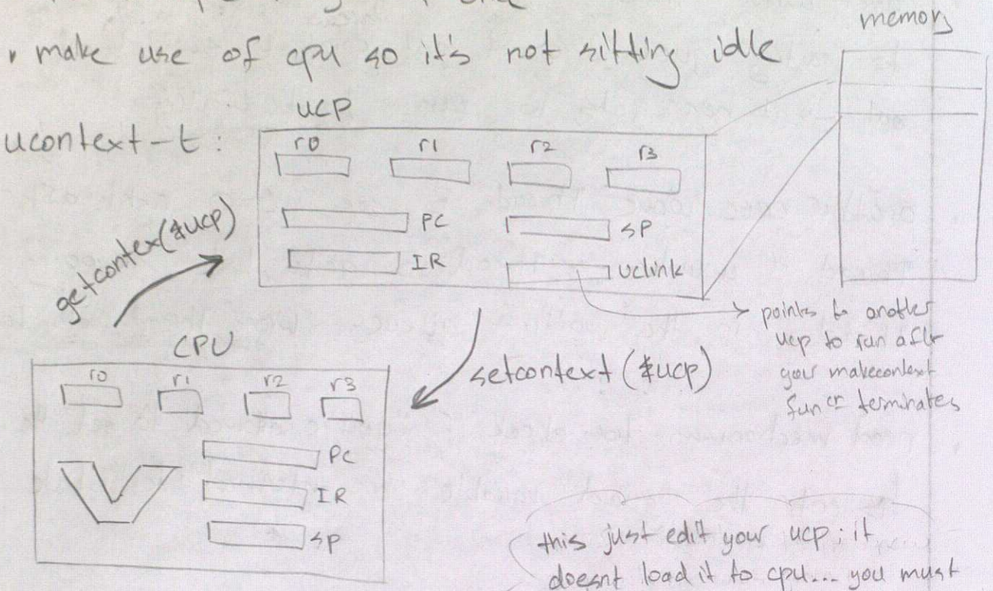
- can analyze race conditions by considering all "preemption combos"
  - → R1 W1 R2 W2 → val = 2
  - but R1 R2 W1 W2 → val = 1 ← bad
- mutual exclusion → "mutex"

```
void* run() {
    int i;
    for (i=0; i<10000; i++) {
        pthread_mutex_lock(&mutex);   ← (a)
        ++counter;
        pthread_mutex_unlock(&mutex); ← (b)
    }
    return NULL;
}
```

Say process has th1 and th2. Th1 gets to initialize i, and the th2 gets to do the same, so that's fine. But once Th1 executes (a), it means that the sandwiched

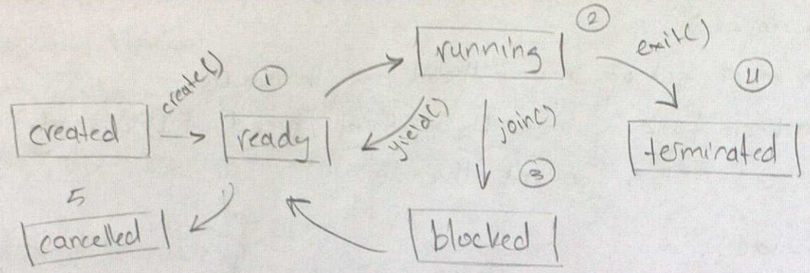
- atomic compare-and-swap
- case where Thread 2 doesn't get to add itself to waiting queue b/c it got context switched out → it never gets to run: Lost Wakep
- another case where Thread 3 sees L=0 right after Thread 1 unlocks → Thread 3 gets lock, skipping Thread 2 in the waiting queue. "Wrong" Thread Gets Lock
- need mechanism to check if "you're allowed to get the lock"
  - ↳ enter the "guard" variable., an intermediate state

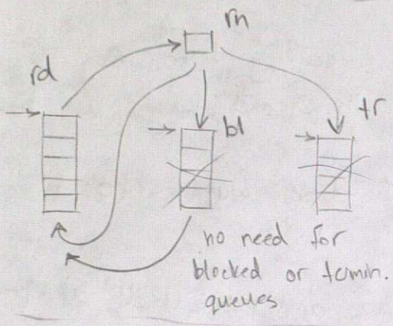
- Concurrency Uses:
  - "run" multiple things at once
  - make use of cpu so it's not sitting idle
- ucontext_t:



- makecontext(...): lets you "edit" where your context starts execution from by providing a "pointer to a func" to run. (if that function returns/exits, the process exits)
  - this just edit your ucp; it doesn't load it to cpu... you must call set... or swap... to actually run
- swapcontext(..ouctp,..uctp): solves the "process exiting" issue by saving current context to ouctp before making control go to uctp → now can setcontext(ouctp) at the end of the run function to return execution to calling thread
- must remember to delete_stack() when done w/ threads.

- wut_init():
  - → must set up "main" thread as thread 0
  - → keep track of currently running thread, FIFO ready queue, and TCB's for all threads
- wut_id():
  - → returns currently executing thread's id
- wut_create():
  - → create new thread that starts executing from "run" function
  - → returns id (sequential, start with 0, and always use lowest avail. #)
  - → add to ready-queue
  - → when thread completes "run" function, wut_exit(0) is called implicitly.
- wut_yield():
  - → lets next thread take its place and puts itself at back of ready-qu.
- wut_cancel(): → can't be joining any other thread anymore
  - → remove thread from ready-queue and free its stack and ucp
  - → set status to 128; maintain info on TCB
- wut_join(id)
  - → blocks until "id" thread terminates/exits or is cancelled; after that, calling thread put at end of ready-queue
  - → frees all stack and ucp memory and TCB deleted and "id" now avai.
  - → a thread can only wait on one other thread, and that thread you're waiting on must be in ready-queue
- wut_exit();
  - → exit after setting status in TCB





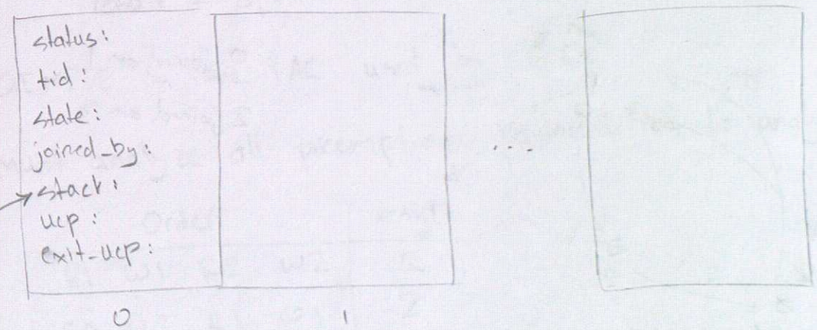
TCB

```
int status;
ucontext_t* ucp;
int id;          char* stack
int state;
ucontext_t* exit_ucp;
int joined_by; (-1)
```

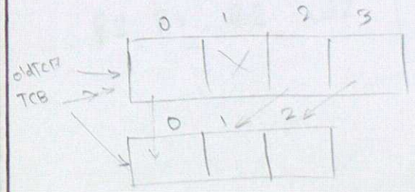
- trick to ensuring all threads by default, exit by wut_exit() is to create an "exit_ucp" copy that you makecontext()'d to the wut_exit() function as the run function.
- assuming that you cannot cancel a blocked thread.
- getting lowest TID: [ ] [ϕ1.1]
- globals: num_threads | currently_running | ready_q
- threads stay in running until they yield() or exit()
- in wut_exit → 1) check if this was the last thread
  2) check if this was joined by another thread
  3) else run next thread from ready_q (and you go to ④)
- in wut_join → 1) check errors 4-7
  2) if thread "id" already terminated return status and cont. execution
  3) else run next thread from ready_q (and you go to ③)

Errors

- wut_yield(): 1) no available threads to switch to/run
- wut_cancel(): 2) invalid thread to cancel ← (valid means thread you wanna cancel is in ready_q)
  3) cannot cancel self
- wut_join(): 4) thread "id" already joined by another thread
  5) invalid thread to wait on ← (valid means thread you wanna join is in ready_q)
  6) cannot wait on self
  7) thread "id" is blocked (already waiting on another thread)
- in wut_yield() → 1) check error 1
  2) else run next thread from ready_q (and you go at tail of ready_q)

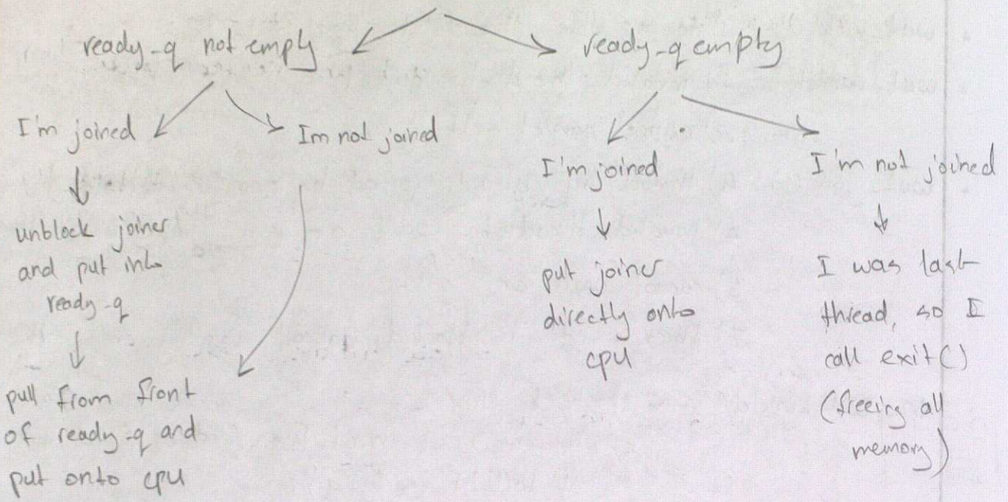


- the exit_ucp needs its own stack → ∴ 2 stacks per thread
- ∃ a potential issue with main thread 0's exit

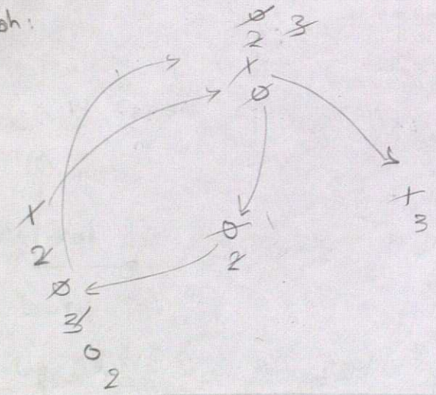


also, must run idx = TID_ind_(id) again after context'ing back!!!

* special cases: in exit:



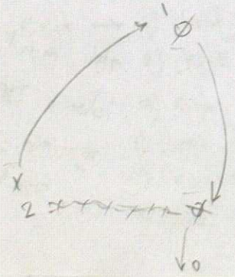
rishabh:



0 joined on 1

2 joined on 3

rachelpie1



* data race: when 2 concurrent operations happen on a memory location with at least one of them being a write operation → may get unexpected values "you get old value"
* atomic operation: it happens all at once (unsplittable)
  ↳ but can pre-empt between atomic operations
* Three Address Code (TAC): intermediate code used by compilers, each is an atomic operation (basically like easier assembly with infinite registers)

  result = operand 1 operator operand 2
* GIMPLE is the TAC used by gcc
* must analyze all preemption possibilities to analyze data race

Order				result
R1	W1	R2	W2	2
R2	W2	R1	W1	2
R1	R2	W1	W2	1
R2	R1	W2	W1	1
R1	R2	W2	W1	1
R2	R1	R2	W2	1

- mutex → "mutual exclusion"

```
pthread_mutex_lock(&m1);      ← regular code
        ⋮                     } protected code
pthread_mutex_unlock(&m1);    ← regular code
```

- safety, liveness, bounded waiting.
  - safety → single thread in critical section
  - liveness → something must run
  - bounded waiting → starvation-free

- efficient, fair, simple
  - efficient → not consume resources while waiting
  - fair → maybe a FIFO
  - simple → easy to use, hard to misuse

- problematic lock:

```
void init(int *l) {
    *l = 0;
}
void lock(int *l) {
    while (*l == 1);
    *l = 1;
}
void unlock(int *l) {
    *l = 0;
}
```

← this has a data race issue because 2 threads could read *l = 0 and then both try to write *l = 1

- Peterson's Algorithm and Lamport's Bakery Algorithm
- ⇒ compare_and_swap(int *p, int old, int new) ← atomic
  - ↳ returns original value pointed to
  - ↳ only swaps if original value equals old; and changes it to new
- valid lock: spinlock

```
void init(int *l) {
    *l = 0;
}
void lock(int *l) {
    while (compare_and_swap(l, 0, 1));
}
void unlock(int *l) {
    *l = 0;
}
```

*l = 0 | Th1 | Th2
cas - 0 goes thru while(0) | cas - 1 gets stuck in while(1) until thread 1 unlocks *l = 0

still has a "busy wait" problem

- we can add a yield() inside while(cas(...)), but now we have thundering herd problem, also unfair
- I can add myself to lock wait queue in while(cas(...)) b4 calling yield() and have unlocking thread wake up first guy in lock wait queue if ∃ someone
  - ↳ now have lost wakeup and wrong thread gets lock

* lost wakeup: A thread A already has lock. A thread B ∴ can't cas and goes into while loop, but before it can add itself to lock wait queue, it gets contexted out. Thread A unlocks, checks the wait queue, finds it empty, and then just moves on. When thread B is put on cpu again, it continues what it was doing inside the while and ∴ adds itself to wait queue and goes to sleep → but we know ∃ no one to wake B up!

* wrong thread gets lock: Thread A has lock and Thread B is first in wait queue and Thread C is about to attempt to get the lock. Thread A unlocks and then immediately gets contexted out. Thread C comes on cpu, sees unlocked lock, and takes it (cas = 0). Now our wrong thread got lock issue already happened b/c Thread B didn't get lock ... what happens next is irrelevant. One possible series of events now is Thread A comes on cpu, and wakes up Thread B. Thread B wakes up, attempts to get lock but fails since Thread C stole it, gets sad and goes back to sleep.

* add a guard variable to create "protected" code → spinlock

```
typedef struct {
    int lock;
    int guard;
    queue_t *q;
} mutex_t
```

essentially prevents lost wakeup and wrong thread getting lock by ensuring that a thread can execute what it wants (like adding itself to queue, or unlocking thread unlocks, checks queue, and wakes right thread) all at once → possible due to mutex like spinlock thru guard var.

```
void lock(mutex_t* m) {
    lock(guard);
    if (m->lock == 0) {
        m->lock = 1;
        unlock(guard);
    } else {
        enqueue(m->q, self);
        unlock(guard);
        thread_sleep();
    }
}
```

```
void unlock(mutex_t* m) {
    lock(guard);
    if (queue_empty(m->q)) {
        m->lock = 0;
    } else {
        thread_wakeup(dequeue(m->q));
    }
    unlock(guard);
}
```

If thread gets contexed out here, thats fine b/c we know it will go to sleep at some point next. worst case, the wakeup call has to be done twice (again retry)

Bonus

read-write lock lets readers go as normal but only lets (locks) on one writer at a time.

Eyolfson Lec 23: Semaphores — Oct 29, 2025

- locks ensure mutual exclusion
  - ↳ only one thread can run protected code (b/n lock and unlock) at a time
  - ↳ does not let you choose ordering b/n threads → ORDERING
- semaphores are used for signaling (with value always ≥0)
  - ↳ wait decrements value atomically → blocks until value is > 0
  - ↳ post increments value atomically

sem_init()	sem_wait()	sem_post()
sem_destroy()	sem_trywait()	

1) lets place wait first (what thread do I want to keep from executing?)
2) pick an initial value (how many threads should be able to pass by the wait initially) → for 2 thread ordering, pick val=0
3) set the post (what wants to unblock a thread)

- to "order" more than one pair of threads, you'd need another semaphore.
- fun fact: we can use a semaphore as a mutex if we set val to 1 → wait ≡ lock, post ≡ unlock
- Note: must now worry about if your programs are thread-safe → eg: printf() is multiple thread safe.
- if some function isn't thread safe and you cannot change its implementation, add mutex lock before and unlock after so code within executes atomically.
- producer-consumer problem:
  - ↳ don't want consumers to consume before producers produce
    1) place wait(&filled_slots) before empty_slot()
  - ↳ 2) pick val=0 since you want all consumer threads to block at first.
  - ↳ 3) post(&filled_slots) after fill_slot() in producer

has an issue with ~~producer~~ ordering when we try to fill a slot that's already been filled (and no produce has emptied it)

- ↳ 1) place a wait(&emptied_slots) before fill_slot() in producer
  2) pick val = buffer size so you'd let the buffer get filled by up to buffer_size, # of producer threads.
  3) place post(&emptied_slots) after empty_slot() in consumer

- some languages have "monitored" methods, where you'd mark methods as such and the compiler would automatically add calls to lock/unlock as needed (mutex)
- condition variable → behave like semaphores
  - ↳ must be paired with a mutex
  - ↳ the wait(...) has a mutex so you can safely edit a variable or add yourself to the queue
  - ↳ another thread calls signal()/broadcast() to wake the first/all thread(s) from the queue
- wait(...) puts you in queue, signal(...) wakes one up
- a thread calling wait(...):
  1) adds itself to queue
  2) unlocks mutex
  3) goes to sleep
  → a/p, another thread must call signal to wake it up
  → 4) tries to get lock after waking up
  ↓
  wait returns when it gets the lock
- we can use condition variables for the producer/consumer example from last lecture.

```
producer() {
    lock();
    while (nFilled == M) {wait(...)}
    // fill a slot
    ;
    signal();
    unlock();
}
```

(you'd use cond. var. for has_empty)

```
consumer() {
    lock();
    while (nFilled == 0) {wait(...)}
    // empty a slot
    ;
    signal();
    unlock();
}
```

(you'd use cond. var. for has_filled)

- idea: what's after your wait is what always runs when your variable becomes what you want it to.
- the issue we saw with the condition while was that we only had lock() and unlock() for T_1 and not T_2. This meant that after going in the while (!condition), T_1 could have been about to call wait, but then got contexted out before it could add itself to the queue. T_2 came on cpu, updated condition, then tried to signal() but found no one in queue (since T_1 got booted off b4 it could add itself to the queue)
  ⇒ T_1 never gets woken up
- solution ez: just wrap T_2 code with lock() and unlock() so now, it can't even do anything until T_1 gets to do its thing (aka, add itself to q, release lock, then sleep).
- you'd want to use a while() instead of if() because if you use if(), there is a chance of returning from wait() and falling through without rechecking the condition
  - ↳ this can happen because wait() would fall thru and continue immediately after returning, but some other thread could have change the condition while that wait thread was asleep (changed it from wat got you into the if (...) {wait(...)} to something else)
  - ↳ bottom line is, if wait returns, it will fall through regardless of the value of the condition.

- locking granularity:
  1) locking large sections of code: less granular and (basically) sequential
  2) dividing locks (smaller sections): more granular and more parallel
- things to consider: overhead, contention, deadlock ← final
  - overhead: creating locks and calls to lock() and unlock() are not free!
  - contention: how likely it is that 2 threads will fight over same lock (more it's gonna end up running in serial)
  - deadlock: no thread can make progress
- 4 conditions (all must hold) for deadlock:
  1) mutual exclusion
  2) hold and wait ← we'll break 2 and 4 to stop/get out of a deadlock
  3) no preemption
  4) circular wait ←

Thread 1	Thread 2
get lock 1 ①	② get lock 2
get lock 2	get lock 1

after ① and ②, you are in a deadlock.

- can avoid 2) and 4) by always calling lock() in order and unlock() in that reverse order → consistent ordering across all threads
- can use trylock.

- initially, takes 10 s
- add 8 threads, but not 8× as fast
- turns out rand() is thread safe, so it implements its own locks on the global seed
  ↳ fixed by using rand_r() and providing local seeds
- but now we still have crazy data races
  ↳ add simple locks to the struct and calling

```
lock(from)
lock(to)
  ⋮
unlock(to)
unlock(from)
```

→ can still cause deadlock if:
T0: A → B
T1: B → A
— hold and wait and deadlock

- can have a while with a trylock to break the hold and wait: you get from-lock, then keep trying to get the to_lock, and if you fail, give it up before yielding and trying again:

```
lock(from)
while(try_lock(to)) {      // returns 0 if it got the lock, returns 1 if it could not get lock
    unlock(from)
    sched_yield()
    lock(from)
}
  ⋮                        // ← if we get here, then that means I have both locks
unlock(from)
unlock(to)
```

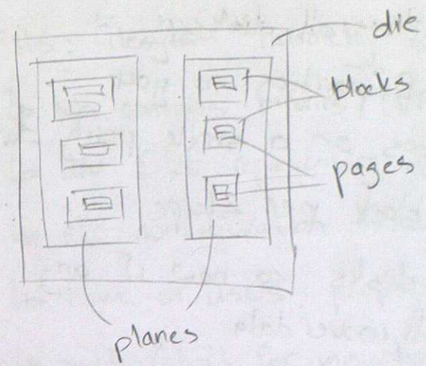
- but ∃ still a problem!!! NOTHING prevents from and to from being the same account →

- if from and to are same account, then the to and from locks are the same, so once I get the from mutex. I will 100% never get it, the same lock btw, a second time
  → no deadlock, but cpu @ 100% but no actual progress!
- easily fixable: add an if condition to account for to and from being same
  → final sol^{n} works
- notice that if there are less accounts, then ∃ higher chance of clash b/n threads* and ∴ higher contention; increase # of accounts >>100 and your speed improves a lot → close to the originally predicted 8x faster (due to 8 threads).

→ if you don't want while, can just use id to check what lock to lock First → so now:

T_0: A → B translates to lock(A) lock(B)
T_1: B → A also // // lock(A) lock(B)

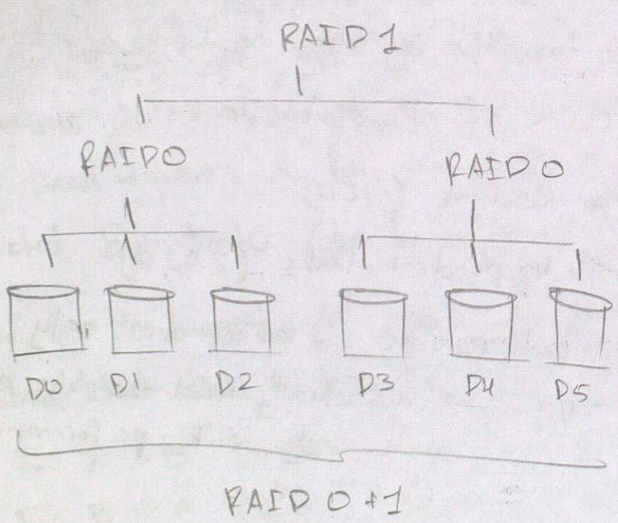
} since checking with id's ensures ordering of locks, (consistently across all threads) I won't have deadlocks



bit harder to write drivers for since can't erase individual pages at a time, only blocks

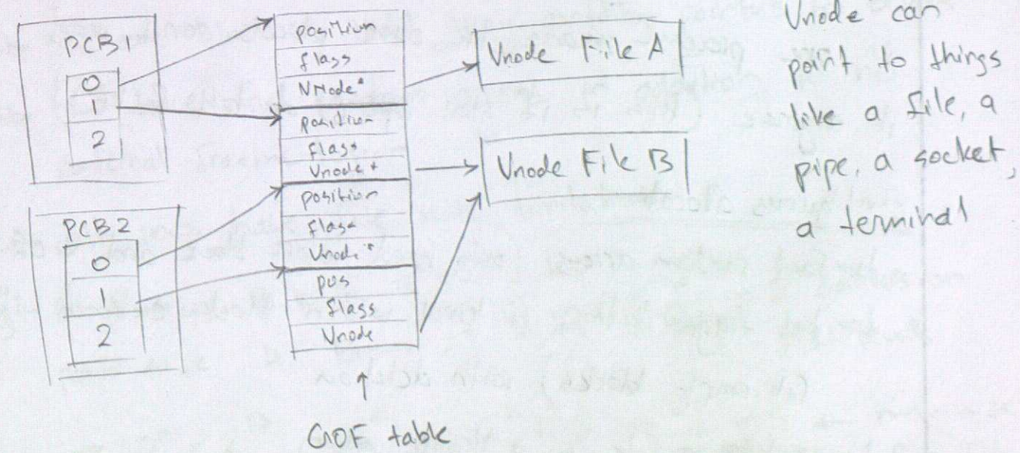
- reading < writing < erasing; can only erase blocks at a time
- you can write to freshly erased pages only (implies that you had to erase that entire block previously)
- OS communicates to SSD regarding "garbage collection" so it can make use of idle disk time
- Single Large Expensive Disk (SLED)
- Redundant Array of Independent Disks (RAID)
- RAID 0 is a striped volume → performance only.
- RAID 1 is a "mirror" → can actually read quickly as well ... only write performance doesn't go up.
- RAID 4 has striped volumes + a parity disk:
  ↳ need at least 3 drives
  → uses XOR to tolerate a disk failure.

- RAID 5 distributes parity across all disks
  - ↳ each disk has quarter of parities so you no longer concentrate writes on a single parity disk
- RAID 6 adds another parity block per stripe
  - ↳ now essentially 2 parity disks, so now if any 2 drives die, we can still recover data
  - ↳ first one still uses XOR, second one uses black magic
  - ↳ minimum needs 4 disks
- can combine different RAID systems. example:



- but RAID 1+0 is slightly more preferred

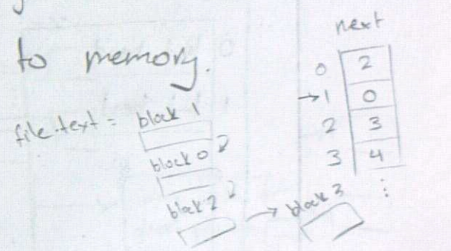
- FHS: filesytem heirarchy standard
  - ↳ bin contains binaries (like ls)
  - ↳ dev files representing physical hardware
  - ↳ etc configuration files
  - ↳ home or users, people's own files
  - ↳ mnt (short for mounted) → used for usb's etc.
- absolute paths start from "/"
- relatives paths are relative to current directory.
- fun bug → feature: hidden files start with "."
- Each PCB contains a file table, and an entry in that points to things inside a "globen open file" or GOF table



Vnode' can point to things like a file, a pipe, a socket, a terminal

- on fork(), the PCB's fd table gets copied, so the 2 processes' fd table point to the same things in the GOF table
  - ↳ means if you do lseek() in one process, it affects the fd of your child process.
- doing open() after fork() solves this
- this why typing in stdin means only one process gets to see it (since the forked processes will have their seek position altered by the process that reads from stdin)
- also, this is why if you have 2 processes reading in one process means the other process can't read it anymore (this is if you open()'d b4 fork())
- contiguous allocation
  - ↳ fast random access (only need start block and # of blocks)
  - ↳ but fragmentation (internal within block, external with full empty blocks) with detection
  - ↳ modifying file is a tragic story

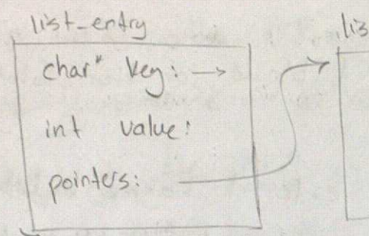
- linked allocation
  - ↳ space efficient (need just start block, can then follow next pointers to NULL)
  - ↳ files can grow/shrink, no external fragmentation
  - ↳ but slow random access (must read full block just to read next pointer at end)
- file allocation table (FAT) — used for Windows boot FAT32
  - ↳ moves the list to a separate table (of every block on disk)
  - ↳ basically, move all the next pointers to one block
  - ↳ gets worse and worse the larger your hard drive is
  - ↳ the FAT can actually be loaded to memory.

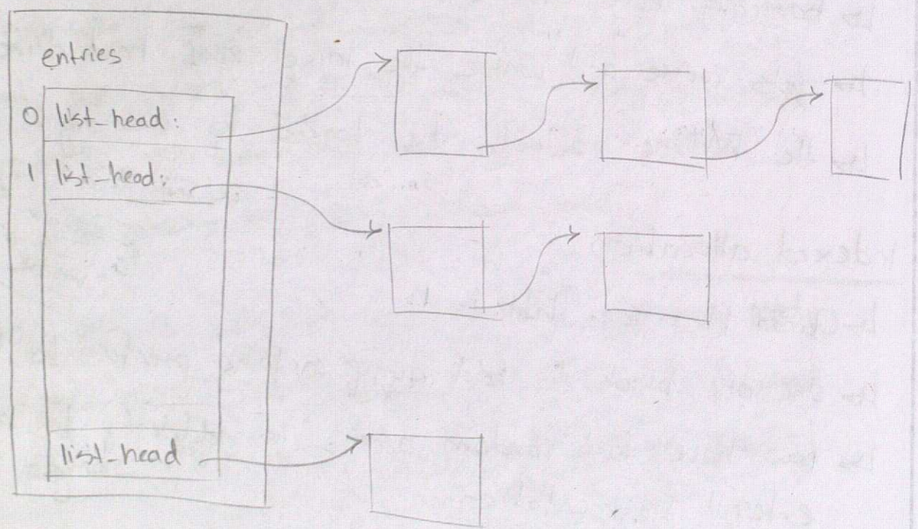


- Indexed allocation
  - ↳ kinda like page table
  - ↳ use one block to hold array containing pointers to blocks
  - ↳ now have fast random access in addition to no external fragmentation
  - ↳ but now have file size limitation

eg: Block size: 8 KiB = 2^{13} bytes
pointer size: 4 B = 2^2 bytes
$\left.\begin{matrix}\\\\\end{matrix}\right\}$ $\frac{2^{13}}{2^2} = 2^{11}$ pointers on that block

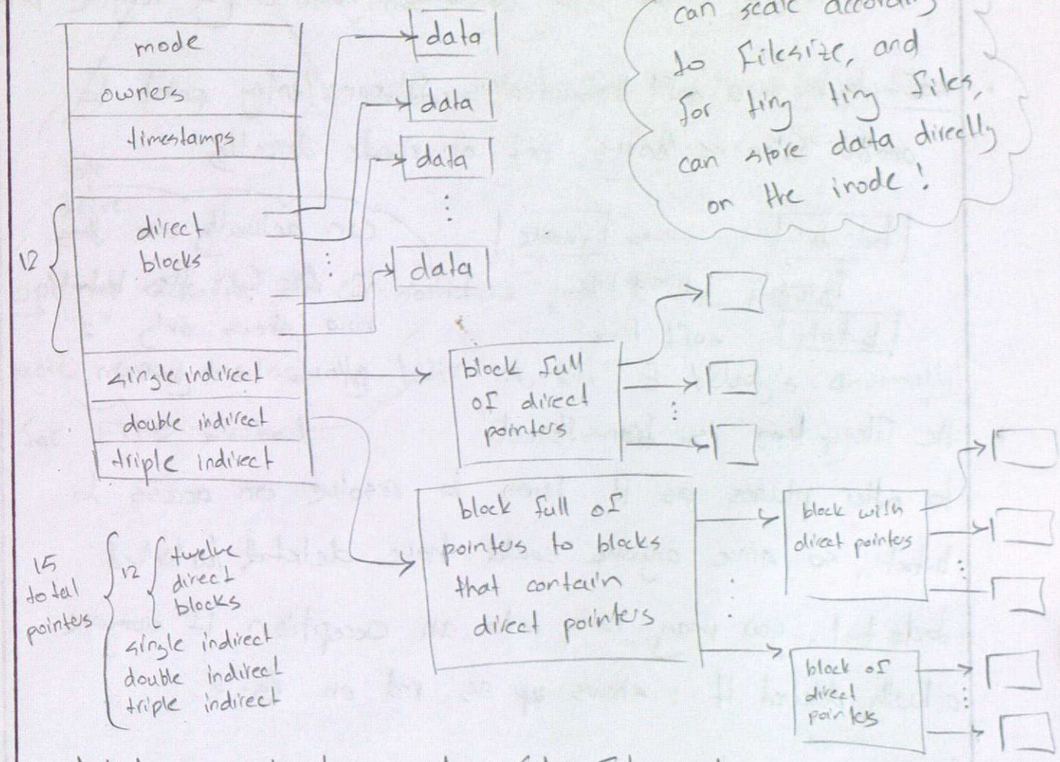
→ $2^{11} \times 2^{13} = 2^{24} = 2^4$ MiB = 16 MiB ← max size of a file right now





inodes: describes a filesystem object (files and directories)

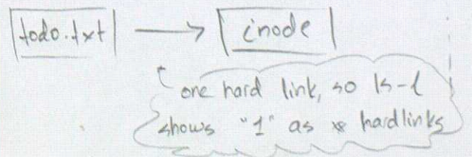
↳ helps us use a "vm" idea solution to solve the limited max filesize issue with indexed method



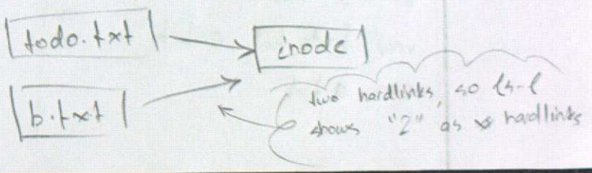
can scale according to filesize, and for tiny tiny files, can store data directly on the inode!

hard links: a directory entry (aka filename) points to an inode ← called hard link

can link filenames to the same inode

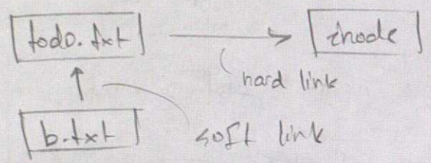


ln todo.txt b.txt



- "removing" a file (with filename) basically unlinks that filename from the inode
  - ↳ you actually erase inode if you unlink the last hardlink → this is when you empty recycle bin

- soft links: a soft link makes filename/entry point to another filename/entry, not an inode directly



can actually see this in ls-l: the b.txt now shows only "1" for ∞ hard links

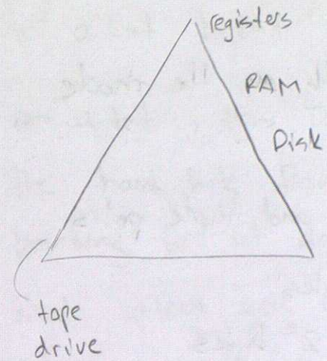
- the filesystem is transferred to other places as it tries to resolve an access to b.txt, so since anyone could have deleted/unlinked todo.txt, you may run into an exception if someone actually deleted it: shows up as red on ls-l

- "touch tod.txt" ← creates empty file basically

touch todo.txt
ln todo.txt a.txt
ln -s todo.txt b.txt
mv todo.txt c.txt
rm a.txt

- directory inodes contain filenames

Memory Hierchy



	speed	capacity	cost/bit
registers	fast	small	expensive
slow	largest	least expensive	

- optimal example → can somehow predict the future
- more memory is actually better irl, but ∃ belady's anomaly for FIFO method.

Eydlson Lec 28: inodes — Nov 22, 2025

- inodes are efficient for small files with support for large blocks
  - can scale according to file size
  - if file very small, can store directly on the inode
- inode theoretical file size:
  - 12 direct ptrs, one single, double, and triple ptrs
  - disk block is 8 kib $= 2^{13}$ bytes
  - ptr to block is 4 bytes $= 2^2$ bytes
  - indirect contain just block*

Ans: have $\frac{2^{13}}{2^2} = 2^{11}$ block*/block

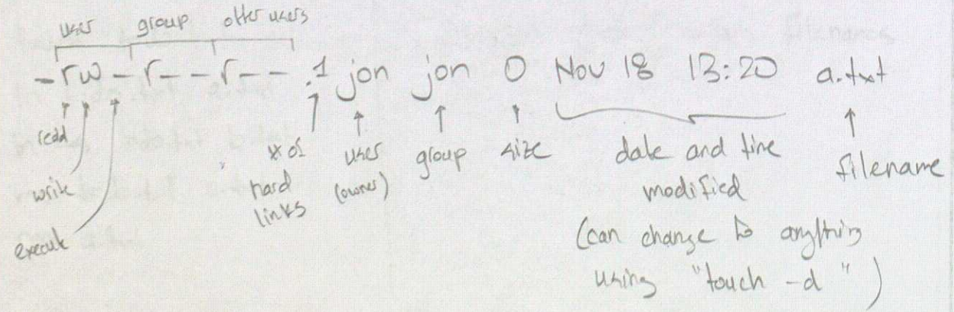
$$= 12(2^{13}) + 2^{11}(2^{13}) + 2^{11}(2^{11})(2^{13}) + 2^{11}(2^{11})(2^{11})(2^{13})$$

$$= 2^{13}(12 + 2^{11} + 2^{22} + 2^{33}) \simeq 64\,\text{TiB}\ !!!$$

- hard links are pointers to inodes

→ "touch a.txt" creates new file a.txt

→ "ls -l" lists directory with details



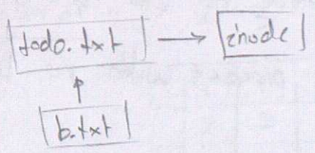
→ doing "ls -li" shows the inode numbers in a new column

→ ln a.txt b.txt

→ rm a.txt : this will remove the hard link from a.txt to the inode it was pointing to. It does not fully erase/delete the actual inode unless you call rm on the last hard link to the inode, otherwise it just removes the hardlink associated with the filename you specified, keeping the inode and other hardlinks.

→ mv b.txt a.txt : this just renames b.txt to a.txt and the hard link points to the same inode (nothing is actually moved)

- a <u>soft link</u> is a name to a name



useful when, say, you'll be changing the todo.txt hardlink again and again for whatever reason but still want to access the current inode via a single static name, b.txt.

→ "ln -s todo.txt b.txt"

- kernel prevents infinite loop of soft links
- directories are just files of type "directory"

→ "stat a.txt" gives info on inode.

- Use a filesystem cache in main memory to speed things up and only write to disk periodically or as needed
  - temporal locality: used block x, likely to use x again
  - spatial locality: used block x, likely to use x+1 soon
- flush() or sync() force a disk write.
- journaling filesystem: to delete a file, ∃ 3 steps and failing/crashing between any of those leads to a storage leak. Journaling means I flush to disk my "plan", execute plan, and then delete journal entry (or mark as done). If I crash in steps, kernel can just check journal to see what I wanted to do and what actually got done → fix/complete process with no leaks.

- using a "disk" as a cache for memory.
- each level wants to pretend having speed
- demand paging: use memory as a cache for disk (only load pages when I need them (get a page fault))
- a process's "working set" should fit in memory, or else it will "thrash" (constantly swapping pages in and out of disk/cache/swap)
- page replacement algorithms:
  - optimal
  - random
  - FIFO
  - LRU

  } we want the least number of page faults (aka going to disk to retrieve a page that I need)

Optimal: 1 2 3 4 1 2 5 1 2 3 4 5

1	1	1	1	1	1	1	1	1	1	4	4
	2	2	2	2	2	2	2	2	2	1	1
		3	3	3	3	3	3	3	3	2	2
			4	4	4	5	5	5	5	5	5
x_1	x_2	x_3	x_4			x_5				x_4	

kick out 4

can kick out anything other than 5

Optimally, ∃ 6 page faults

- FIFO: 1 2 3 4 1 2 5 1 2 3 4 5

1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3
x_1	x_2	x_3	x_4			↑ x_5	x_1	x_2	x_3	x_4	x_5
						kick out 1					

FIFO has 10 page faults

- Belady's Anomaly says that for FIFO page replacement algorithm, more page frames (aka able to hold more pages in memory) means more page faults.

- LRU: 1 2 3 4 1 2 5 1 2 3 4 5

1	1	1	1	1	1	1	1	1	1	1	5
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	5	5	5	5	4	4
			4	4	4	4	4	4	3	3	3
x_1	x_2	x_3	x_4			↑ x_5			↑ x_3	↑ x_4	↑ x_5
						kick 3			kick 4	kick 5	kick 1

Least Recently Used: 8 page faults

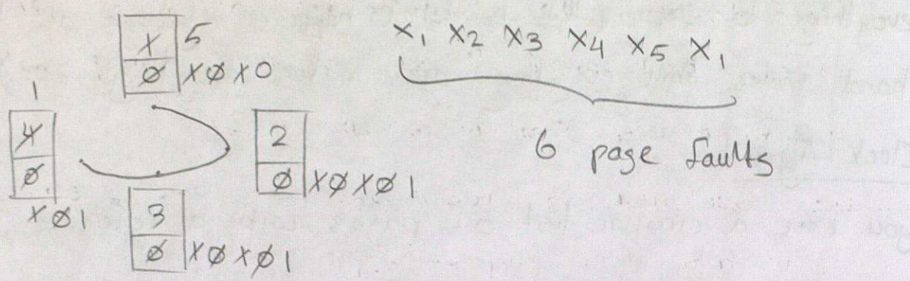
- LRU is software-expensive → create an approximation of LRU → clock algorithm next lecture

- if everything is a 1, keep hand where it is and change everything to zero (this is what happens anyways as the hand moves thru everything and makes everything zero)

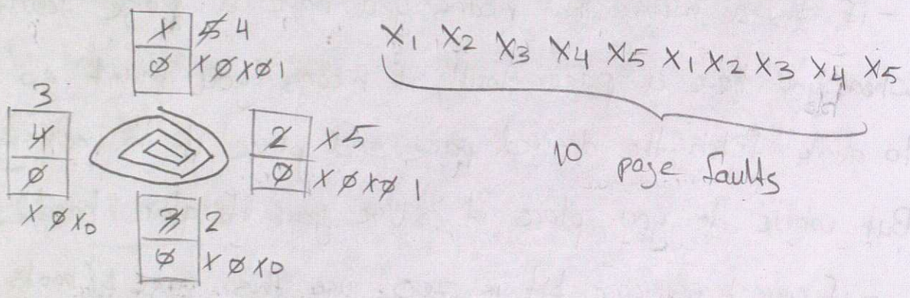
Clock Algorithm

- you have a circular list of pages with a reference bit
- say you want to access a page
  - if it's a hit, set reference bit to 1
  - if it's a miss, that means you have a page fault
- when you have a page fault, it means you must go to disk, fetch the desired page, and place it in memory
- But where do you place it? Use your iterator "hand"
  - if current reference bit is zero, use this page (boots current page off so you can place newly fetched page)
  - if current reference bit is one, set it to zero and move to the next page to start this process again
- Think of reference bit like that page's life
  - if a page is accessed, then it gets a life since it might be accessed again (so if iterator checks it, it doesn't get booted off and instead just has its "life" removed)
  - if ref was 0, that means it prev. lost its life already and can get kicked.

Example 1: 1 2 3 4 5 2 3 1 2 3 (max 4 pages)



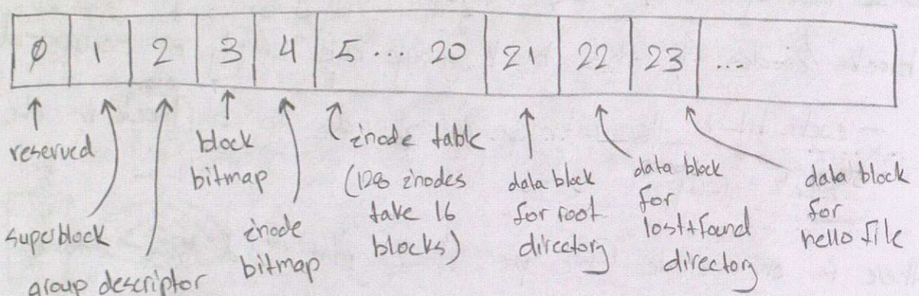
Example 2: 1 2 3 4 1 2 5 1 2 3 4 5



- you'd rather know about low memory instead of running slowly (using swap memory is slow)
  - can disable swap memory and just kill a process to free up memory
- increasing page size speeds things up
  (trading fragmentation for more TLB coverage)

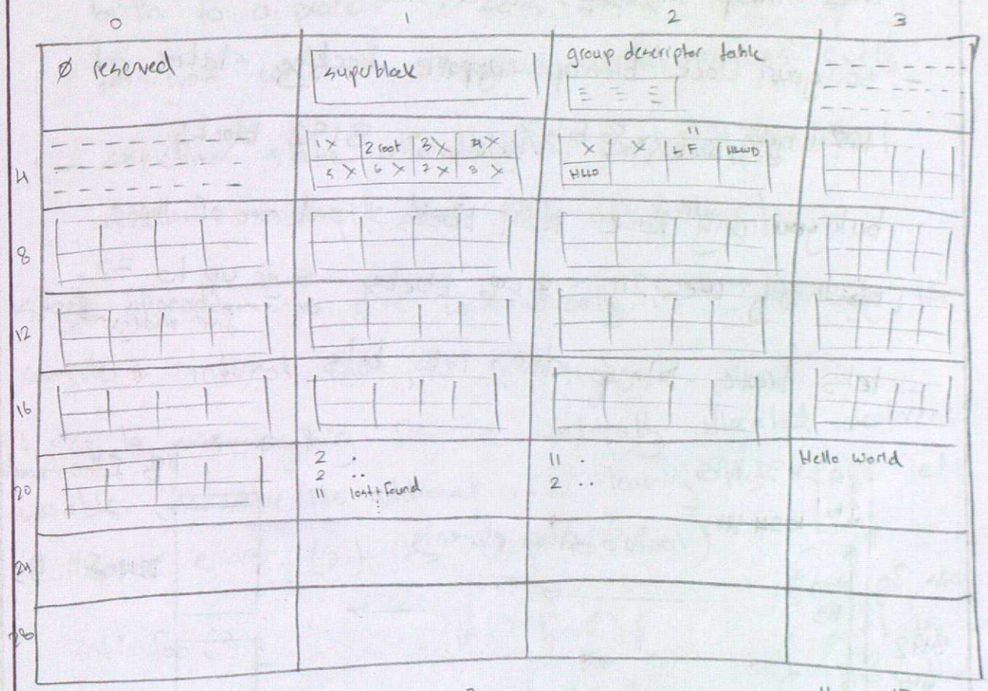
- superblock helps you when you mount
- block group descriptor table contains info of where things are in each block group (block bitmap, inode bitmap etc.)
  - an entry contains info for a block group
  - eg: if you have 3 block groups, will have 3 entries in this block group descriptor table
- block bitmaps and inode bitmaps let you know which blocks/inodes in this block group are empty or occupied
  - each bit in the bitmap corresponds to a block in the block group
- there is one inode table per block group and this inode table's entries are the inodes in that block group
  - inode can be anything: regular file, directory, socket etc.
- if an inode is a directory, then we have something interesting:
  - its direct data block pointers point to data blocks that contain a directory table
  - this directory table has entries for more directories (like "." and "..") and files etc
  - entry contains things like inode # and name, so you can go to that inode when following a path for example
- the rest of the blocks in a block group are data blocks!

- it is defined that superblock starts 1 kib after the very start of the disk, so since our filesystem has block size 1 kib, we know superblock starts at exactly block 1 (the second block, after block 0)
- if for eg block size was larger at 2 kib, then superblock would be on block 0, just 1024 bytes = 1 kib into block 0

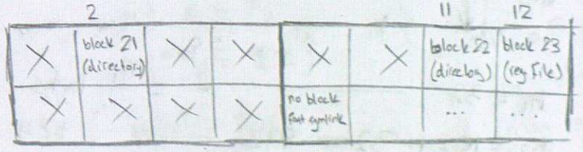


- the symbolic link will not waste a whole block since we will use the tiny optimization (store it on the inode itself)
- first inode at 11 (1 DNE, 2 is for root)
- defined already which inode to store lost+found, hello world etc.
- root user id and root group id is 0
- all directories will fit on a block for this lab.
  - 2 ("inode block size") = 1 (our block size) => 2(512) = 1024
- will write a directory, a file, and a symbolic (<60 bit) link

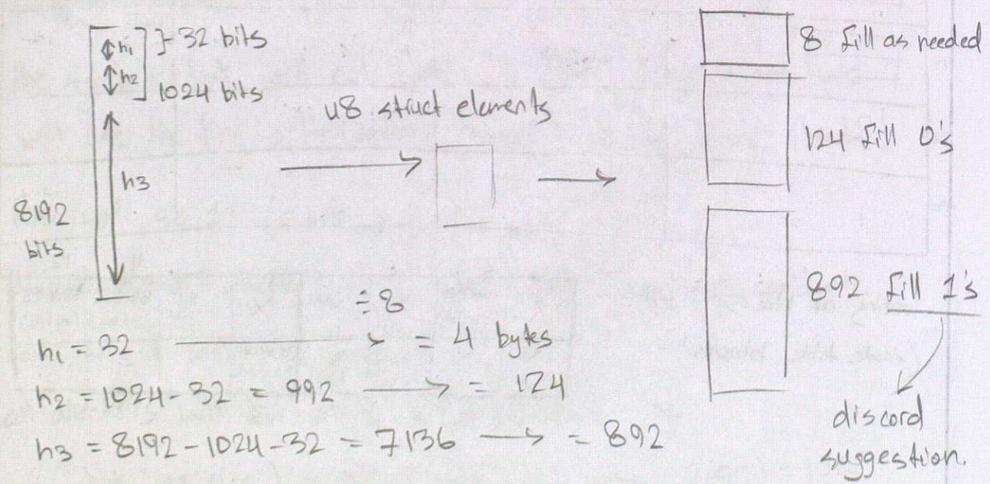
- for symbolic links, must make sure "i-blocks" is set to zero on the inode, since we aren't going to be using any blocks to store the path to the file.
- had to use memcpy to fill the symlink inode's i_block[] instead of write()ing directly to a lseek()'d cursor inside fd.



looking at blocks 5 and 6 (inode table blocks)



- The bitmap padding issue:
  - you have total 1024 blocks, and each of those is 1024 bytes big → 1024×1024 = 1 MiB file system
  - now, your bitmap is a block big, and each bit in the bitmap corresponds to a block
  - so your block bitmap supports tracking status of 1024 byks/block × 8 bits/byte = 8192 blocks
  - but you only have 1024 blocks! and out of those, you'll only use 24 = 3×8 blocks → go up to 32 for margins
  - let's break bitmap down into bits



$h_1 = 32 \longrightarrow \div 8 = 4$ byks

$h_2 = 1024 - 32 = 992 \longrightarrow = 124$

$h_3 = 8192 - 1024 - 32 = 7136 \longrightarrow = 892$

- static allocation = simple strategy: eg: char buff[4096]
  - memory that needs to be allocated is known (and fixed) at compile time (and assembly)
  - when kernel loads program into memory (hence, giving birth to a process) it sets aside that memory for you, which is all yours until process exits
  - sometimes, static allocation is wasteful, size not known at compile time (only at runtime)
- stack allocation is done automatically in C through the compiler's internal calls to alloca();
- trying to use anything from a statically allocated/declared variable (its value, its address) after leaving its scope lets C cause errors. (but it's easy to fool)

```
int* foo() {
    int x = 1;
    return &x;
}
```

↑ compiler cause trouble

```
int* foo() {
    int x = 1;
    int* p = &x;
    return p;
}
```

↑ compiler is fine with this.

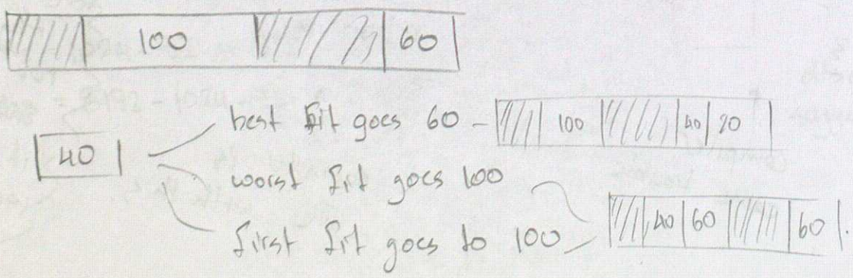
moral of story: don't try using thing that are out of scope (on stack, it may or may not exist depending on if it got overwritten or not

Kernel cannot use malloc bc. it doesn't have standard C library.

- Dynamic allocation has fragmentation issue
  - you'll have "holes" in memory because allocations are permanent and not movable (until user frees it)
  - fragmentation happens if (1) different allocation time, (2) different (and ∴ unknown to kernel) sizes and (3) unable to move around allocated memory
- external fragmentation: wasted space between allocated blocks
- internal fragmentation: wasted space within a block
- most allocators implemented using "free list" concept
- 3 heap allocation strats:
  - best fit: most optimized for space, but more searching
  - worst fit: most wasted space
  - first fit: pick first space that's big enough.

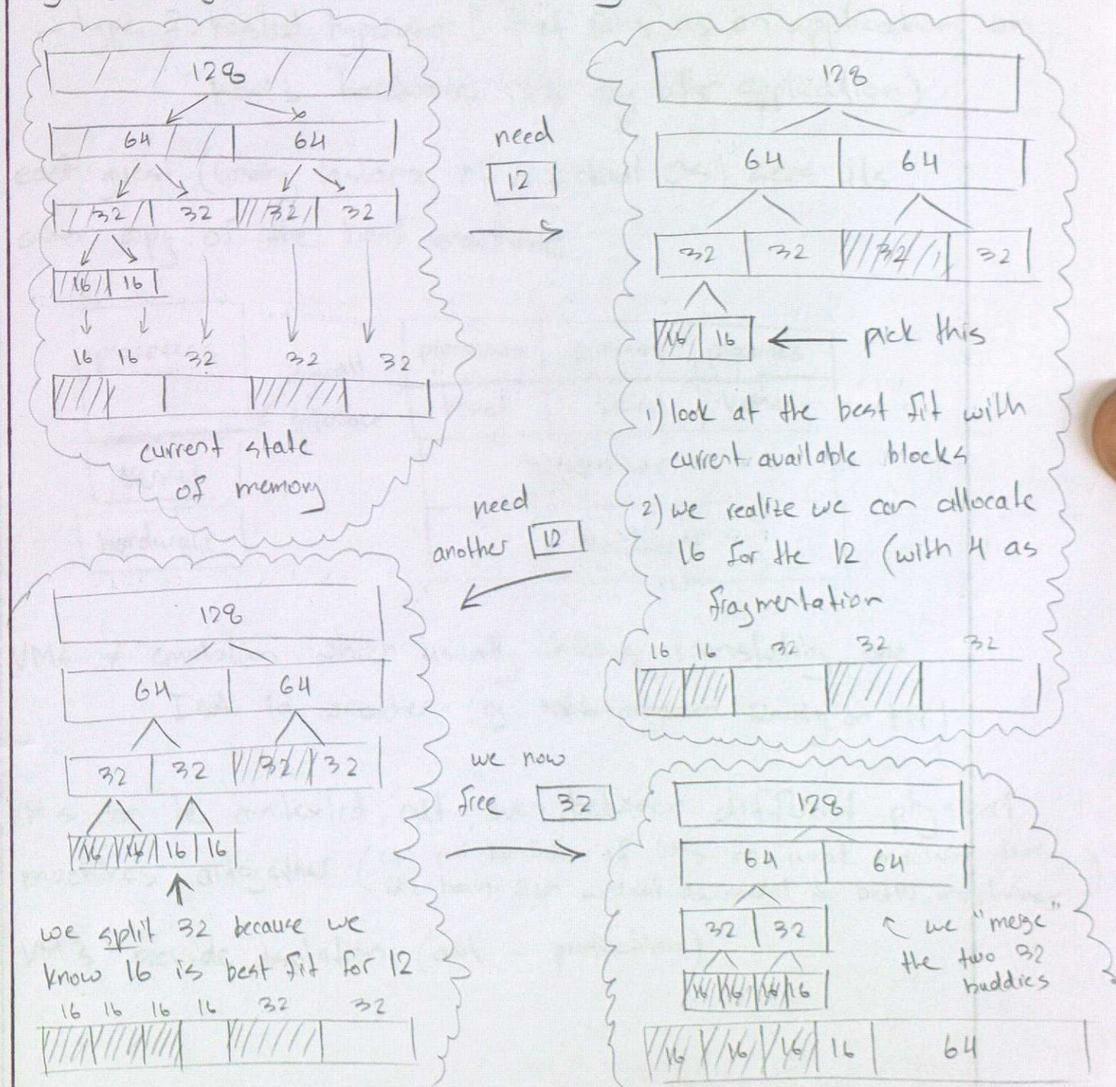
not good but usually leaves tiny holes
worst

on average the best.



- Buddy Allocator: allocate "power of 2" chunks
  - will have some internal fragmentation
  - but faster to allocate, and merging/splitting allows overall fast implementation that's efficient (with space)

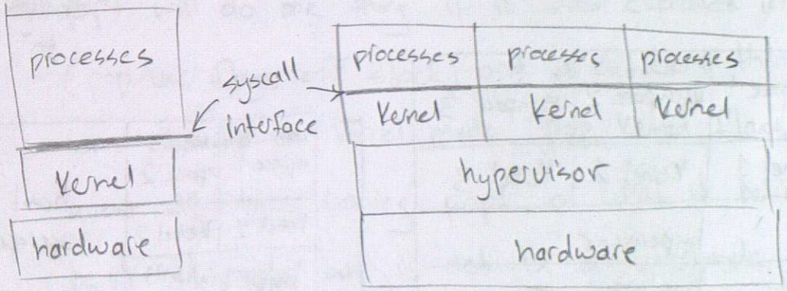
eg: your system has 128 memory in this state:



Slab Allocators

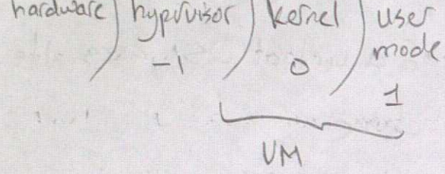
Eyolfson Lec 33: Virtual Machines — Nov 29, 2025

- multiple OS's run on the same machine, and each OS thinks it's the only one running
- Hypervisor: (virtual machine manager)
  - type 1: bare metal hypervisor (runs directly on host hardware)
  - type 2: "hosted hypervisor" that runs as an application on host's hardware (like any other application)
- each guest (running instance of a virtual OS) sees its own copy of the host machine

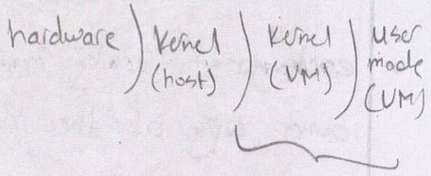


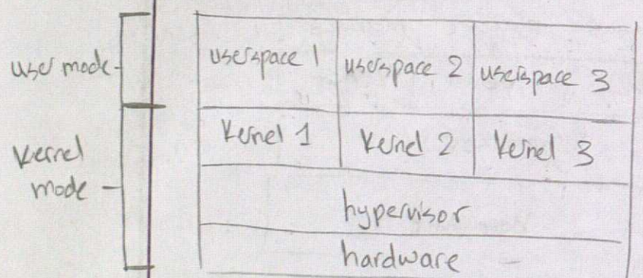
- VMs ≠ emulation, which usually means translating one ISA to another (eg: x86 program running on M1)
- VM's can be contexted out, even between different physical machines altogether. (can put hundreds of VM's on weak machine during idle hours, then context as needed to better machines)
- VM's provide isolation (and ∴ protection)

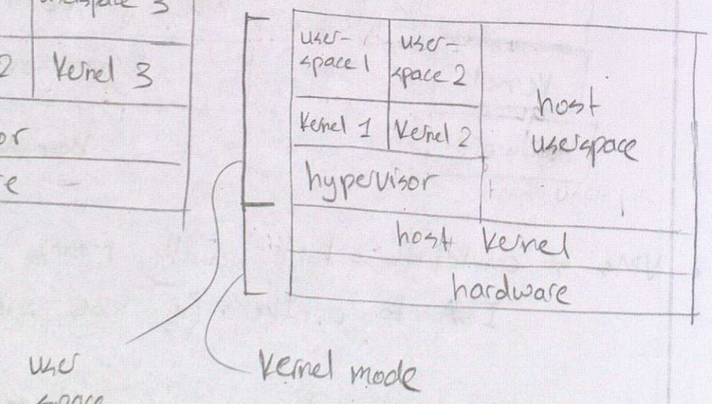
- you have a virtual cpu (Vcpu) that actually store entire state of cpu b/n context switches (compared to just a few user-needed cpu things)
- guests still have a kernel and user mode
  - type 1 hypervisor:



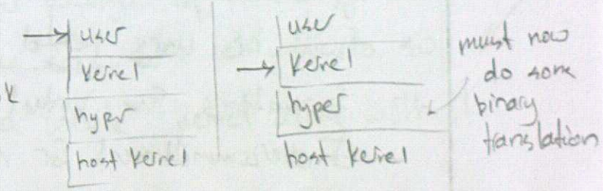
  - type 2 hypervisor





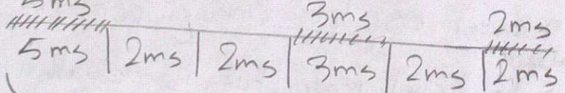


- Trap-and-Emulate: the type 2 hypervisor must "simulate" a privileged instruction. Normally, if a userspace process tries executing a privileged instruction without using a system call, the kernel will terminate that process. In this case though, the entire VM is in userspace, so if the VM's userspace needs to execute a privilaged instruction, a trap is created, which the hypervisor (also in usermode) must "emulate" and return + update vcpu as needed.
  - some instructions exist for both usermode and kernel mode eg: popf will do one thing if it was executed in usermode (pop user flags off stack) and a different thing if it was executed in kernel mode (pop kernel flags). If guest OS tries running popf in what it believes to be kernel mode but is actually a usermode popf in real life (since type 2, entire VM in host's userspace), trap for hypervisor may not trigger and u end up popping user flags instead of kernel flags for VM.
  - those special instructions need binary translation (hypervisor inspects every instruction)



- host OS has access to (claims) hypervisor mode
- Virtual scheduling:
  - ∃ hypervisor threads (like # of actual cores)
  - overcommitting: when there are more vcpu's than there are actual cores → we must start sharing cores (must use scheduling alg's)
  - overcommitting causes virtual (guest) soft real-time tasks to run unpredictably

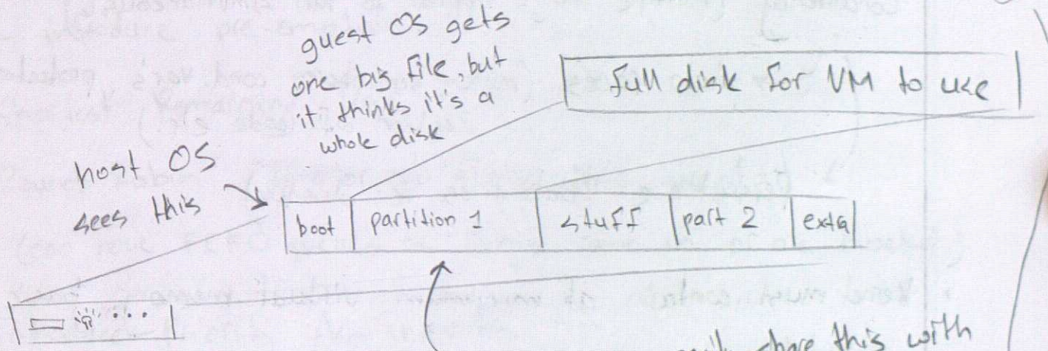
    eg: real-time time slice is 10ms, guest might think that's what's happening, but irl the cpu is being shared with other things so the guest's 10ms might actually be



irl 16 ms passed, but guest cpu thought it ran for 10 ms straight

this one is for type 1 ~~2~~ I think

- Virtual memory: VM processes use virtual virtual addresses
  - hypervisor mode: let's us tell memory controller that I will be using nested page tables, so it can cache translations from virtual virtual addresses directly to phys. addr.
  - ∃ overcommitment for memory too (guest does page replacement)
- guests could share pages if they're duplicates
  - use copy-on-write.
- hypervisor provides virtualized I/O Devices,
  - IOMMU: lets guest use I/O directly during runtime without needing hypervisor to translate continuously as middleman. (useful to let VM's use GPU's without latency)
- VM's boot from a virtual disk



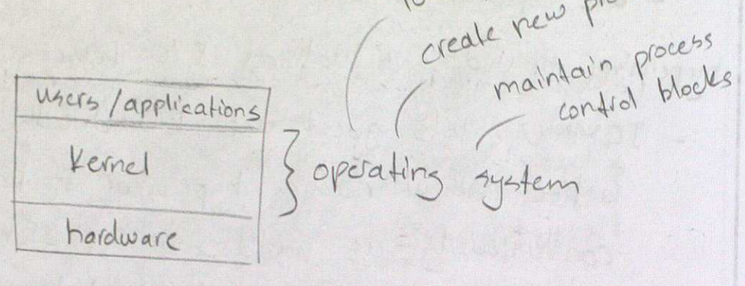
Docker Desktop: installed one Linux kernel VM, and now all your docker container applications are running on that one virtual kernel → can have Docker on Mac or Windows

- Uses of VM's
  - isolation of applications

    (can package applications into isolated VM's s.t. ABI changes in libraries don't cause issues)
  - containers, like docker, let you package applications into isolated containers, but at the same time let you do so without having to include a whole kernel along with it

Tome Review Lecture — Dec 2, 2025

Batch processing



Virtualization (each application thinks it's the only one running)

↓

Concurrency (multiple things appear to run simultaneously)

→ data races (mutex, semaphores, cond. var's, protected section of code etc.)

→ Persistence (data files, ie: disks)

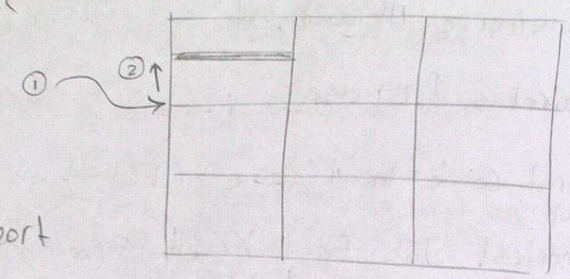
- Kernel must contain at minimum virtual memory, basic scheduling
- fork(); — parent / child
- IPC

Scheduling:

- maximize cpu usage
- minimize wait time and response time for users
- maximize throughput
- fairness for users
- First Come First Serve (FCFS)
- Shortest Job First (SJF) (minimizes wait time)
  - introduce pre-emption
- Shortest Remaining Time
- Round Robin (time or cpu slices, time quanta)
  (can have FIFO queues as things come in or are blocked)
- Introduce Priority Assignments.

Main Memory

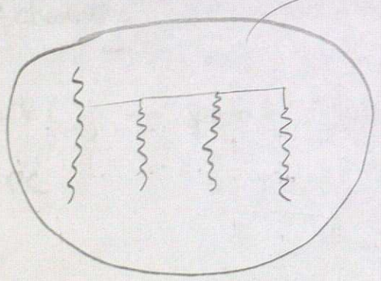
- virtual memory
- take main memory and divide into "chunks" or "blocks"
  - ① find which block/page
  - ② find what on that block/page



- page table concept (need hardware support for this)
  - have page table entries
  - caching TLB

good question: will a first access to a brand new virtual/phys. address always be a write?

Threads and Concurrency



(we don't make distinction b/n parent and child threads)

Q: whether or not kernel schedules b/n threads

- must protect against data races. (mutex: critical sections)
  (idea of atomic instructions)

- technically, multiple reads or multiple writes → no data race
- data races when you have a "read-write" case
- if you cannot guarantee purely reads, must expect data race.

Threads and Concurrency

- protect against data races
- dining philosopher, reader-writer, producer-consumer
- lost wakeup, wrong thread gets lock

Disks

- SSD
- RAID

Filesystems

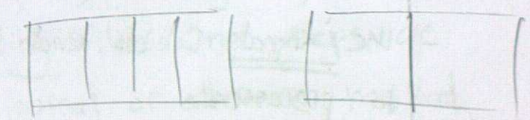
- inodes*
- file structures*
- FAT*

stuff, goal
bing* diff.*
→ 2017
2019 w

Page Replacement

- Clock page replacement
- Buddy and Slab Allocators.

if you know you'll need fixed sized chunks



you'll reduce external ~~allocation~~ frag. but can still have internal fragmentation.