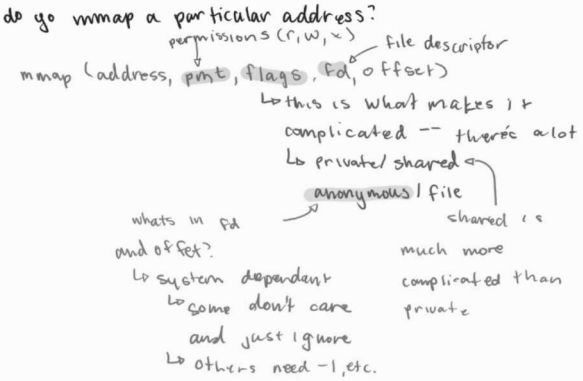
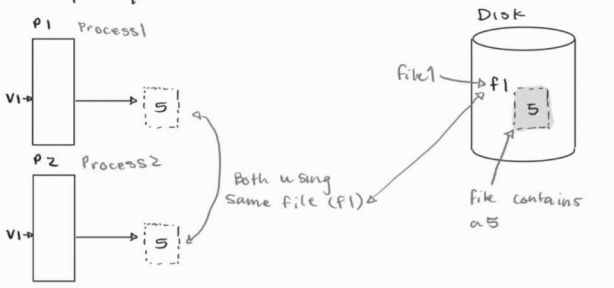
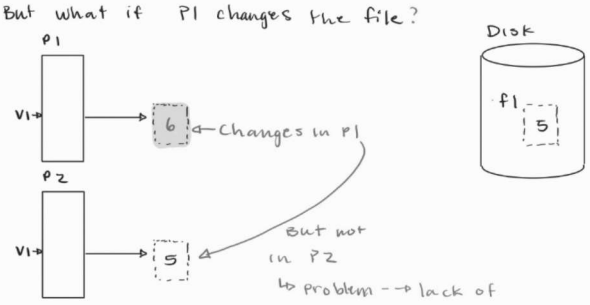
how do yo mmap a particular address?



How do you syncronize file access?



But what if P1 changes the file?

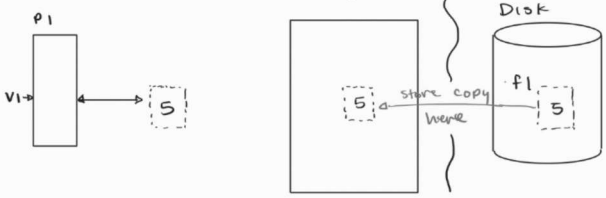


↳ problem --> lack of synchronization

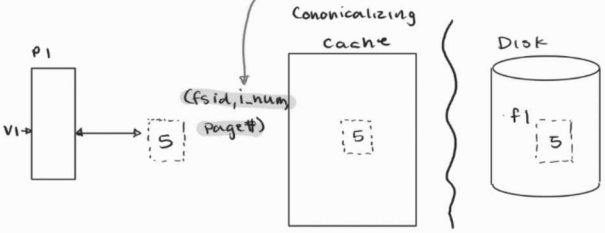
Page cache

use Cononicalizing ~~Map~~ Cache (RAM)

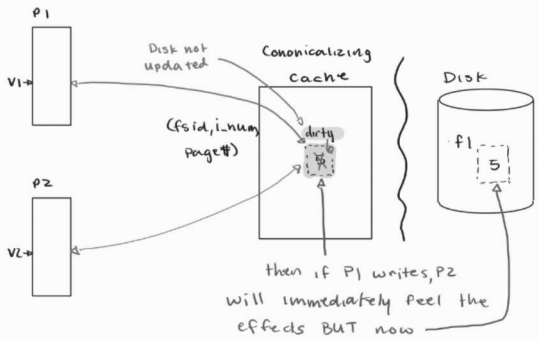
hashmap



How do we find the item in the cache? (what do we hash)



Then each process can just point to the instance in cache



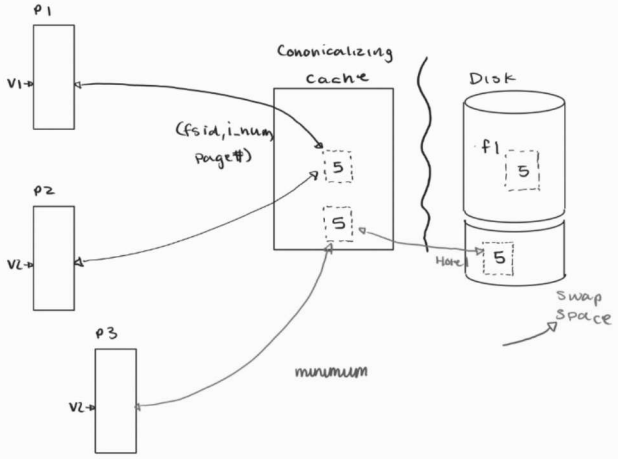
*When another process mmaps this file it will go to the cache and find it
↳ if exists --> point to it
↳ garunteed to be updated
↳ else --> fetch from disk and put in cache
↳ slow!!

*when we close the file...
↳ update it on disk...

[IDK]

* What happens when no one points to the file anymore?
↳ you COULD reference count and delete...
but WHY?
↳ getting it from disk was so slow and the RAM memory space is already so big
↳ keep it in case someone else wants to mmap it later?

What about private mmap?



But... this will waste a lot of RAM
not everyone will try and write

so...

make a "temporary office" for everyone to share until someone tries to write

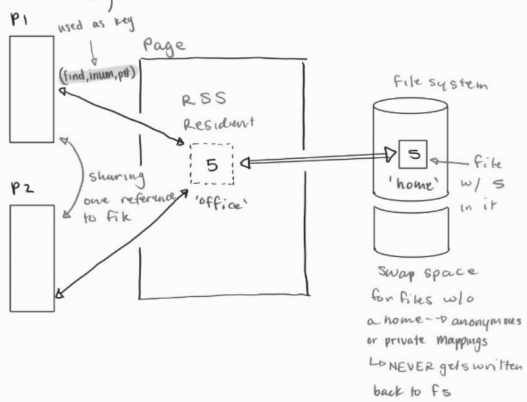
*COW --> copy on write

execl(path, 0)

- This means there are 0 argv's (argc == 0)
- This is valid - it's just weird cause there is typically at least one arg which is smth

dynamic library: libc.so

static library: libc.a



what happens if you run out of physical frames?

- In p6 we would panic but... now what?

- write it back to fs and set the present bit in that process to 0
  - ↳ next time it tries to access the file it will page fault
- we want to avoid clearing a page that will be used soon
  - ↳ how do we know which file will be unlikely to be used?
    - * most optimal is to throw away the page you will need furthest in the future
      - ↳ but... we dont know this...
    - * If the item in residence hasn't been written yet... you don't need to write back
      - ↳ worthwhile optimization.

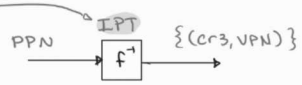
need to find the vpn's

we need some sort of function to invert this...

converts the VPN to get a PPN

~~Inverted~~ Page Table

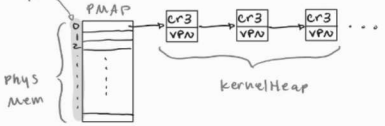
There's a similar hardware component



get the vpns pointing to a ppn

called IPT;
we will call it
PMAP - Physical Map
↳ Hash map?
↳ there's a simpler way!
Array indexed by the ppn
↳ each item points to a linked list
↳ or some other representation of a set

* software
↳ part of physical memory



now when you evict a page you can traverse PMAP and unlink all the VPN's

*why isn't PMAP also in kernelHeap?

- you don't want it to be in virtual pages b/c virtual pages could be evicted
- you also want PMAP to be contiguous to make traversing it easy

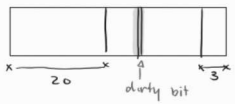
* This should get setup in kernel init

* when we evict...

- If it's dirty, write back to home, else just delete
  - ↳ how do we know if it's dirty?

more metadata!

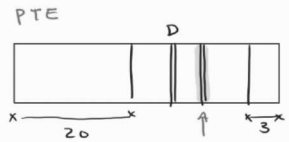
we can mmap the file as read only
↳ if/when we page fault we can see it was supposed to allow write and flip the dirty bit.
↳ This metadata is one of the bits in PTE we had originally ignored



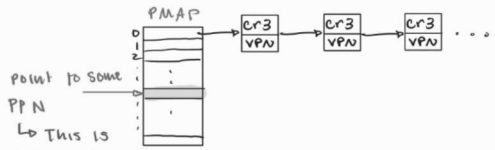
* NOTE: the dirty bit is stored in the PTE NOT in the PPN
  - ↳ to check if a PPN is dirty you need to us the PMAP to check the PTE

- This is very slow due to the write back
  - ↳ Many systems will use a helper thread to constantly check if pages are dirty
    - ↳ they will often require some % of free frames
    - ↳ this makes page faulting faster since you will ALWAYS have free frames
  - ↳ this also makes sure files are written back in case of an unexpected shutdown
- You can also do similar expected operations ahead of time
  - ↳ ex. zeroing out pages

How do we know which PTE's to clear?



- Accessed bit set to 1 whenever accessed (done by hardware)
- Periodically the bit is set to 0
- Second chance algorithm - when the accessed bit is 0, it's given a second chance
  - ↳ if its still 0 a second time then we evict it.



→ traverse the linked list and looks for an accessed bit == 1
↳ if there is one that's 1 then set all the A-bits to 0 and continue
↳ if we ever wrap around and everything is still accessed then this is BAD
↳ computer is TOO busy --> you need more RAM

called the clock algorithm

↳ If you ever find a PPN where all of its VPN's have accessed == 0
↳ evict the page

In P7 we quit if we get a page fault and don't know how to handle it
↳ what we really want to do is try to handle it
- You can let the user try and handle it
  - ↳ ex. they could mmap or smth
↳ Similarly, what if a process wants to get run or called periodically (ex. a clock or sleep function)

↳ Introducing... Signals!

- Like an interrupt
  - ↳ we can't trust the user to set up the IDT and stuff by itself so kernel does it for them

signal(signum, handler) → diff nums for diff issues (segfault, illegal instruction, etc.)

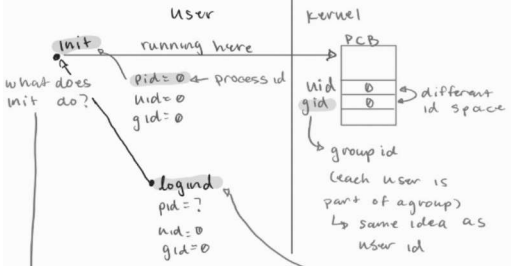
void handler(int signum)

run in user mode
↳ when done → sigreturn(), → syscall: returns to if you don't call it will implicitly call Sigreturn at the end

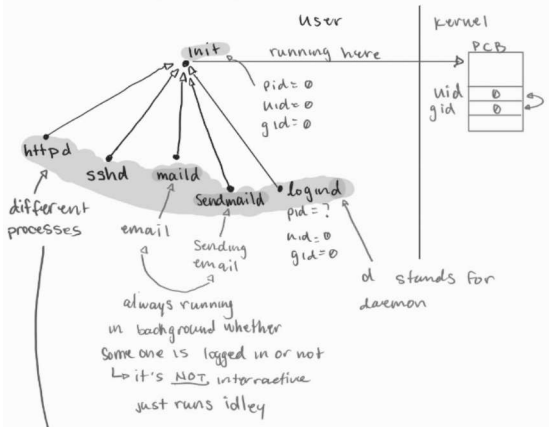
failed instruction

Kill(pid, signal);
↳ if it isn't called from within a handler then it will simply kill the process

* Kill() is syscall to send signal()



- first process created (in user)
  - doesn't know anything
  - in the PCB for every process it will store the user ID
    - ↳ the root of all permissions
      - ↳ init is the root process
      - ↳ the UID is just some small integer
  - There is a SuperUser which has power to see anyone's data
  - after init you will typically need someone to log in and run some operations → logind child of init



* how do these different processes get started?
  - there is some sort of configuration folder with a bunch of different binary files and stuff that tells how to start these processes
    - ↳ these files will also say how to restart in case of failures
    - ↳ they are also all formatted differently
* init starts a config daemon which finds the config files and starts the different config files

Logind
- given uid = 0 which gives the process its super power to access sensitive files
  - ↳ it's CPL is still in user mode!

How do you authenticate & authorize users
- authenticate: the user proves its identity
- authorize: we decide what powers this user has

username & password
↳ proving you have information that ONLY that user will have

↳ what's the login process look like?

```
printf("user name: ");
scanf("%s", &s);
```

- this will tell the login daemon that someone is trying to log in with a particular uid

```
printf("password: ");
scanf("%s", &s);
```

☆ Note that for username the terminal will echo your input & put your typed characters on the output...
BUT NOT for password
how?

* ioctl special sys call that changes the terminal settings → can change the terminal to echo or not
  - raw mode → no echo
  - cooked mode → echo -- usually in this mode

How does the shell allow you to backspace?
- there is a chacter buffer that stores your inputs
  - ↳ NOT in the actual file
- doesn't commit your characters until you press enter and commit the whole line
  - ↳ this is why shell is line based
- other editors will encode the backspace as an actual ascii character and let the IDE handle it
- if the terminal has tab completion it's different
  - ↳ has to be in raw mode b/c the shell can't handle the complex operation
  - ↳ raw mode allows the program to handle the output by itself

/etc/password
- All users can read
- Only root can write
- one line per user

15 : gheith : Ahmed Gheith : 30 : password
- 15 = uid
- gheith = username
- Ahmed Gheith = real name
- 30 = pgid
- password

internally, program only uses uid → username & real name are only for convenience

now we only use a one-way hash

one way hash:
easy to go from actual to encoded → Hard to go from encoded to actual
↳ the hashing function is publicly known

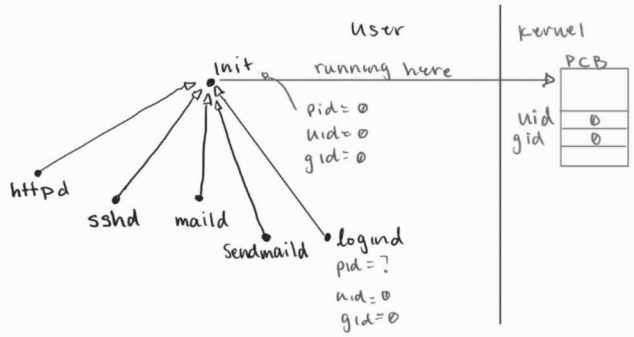
Why isn't /etc/password only readable by root?
It's convenient for the process to have access to it's uid & other user info

* Supposedly the password input will read the password, hash it, and IMMEDIATELY forget it
  - ↳ this WILL fail -- since we're using vmm
    - ↳ when you page out the password is still ther in physmem → can be accessed

Suppose you have an AMAZING hash function
- But someone on the system may choose a password from the dictionary
  - Ex. they use the word "please"
    - ↳ maybe you replace the l with a 1 or write it backwards ... etc
    - ↳ you could write a program that takes all the common possible values and apply the hash function
      - ↳ then you could compare all the possible values to all the hashed passwords
        - ↳ eventually you will get a hit
          - ↳ once you get in you can use other methods to hack other things
    - ↳ Solution: introduce randomization
      - ↳ have some random value associated with each user
        - ↳ now to use this dictionary method you have to iterate over everything again which makes it an unusable system again

All large systems will use LDAP (or something like it)
- Server that manages the users
  - ↳ too many users to fit on one file

Let's switch back...



* now that we can log in we need to run some program
  - ↳ how do we tell the system what program to run?
    - ↳ in etc/passwords we store the default program

15: gheith : Ahmed Gheith : 30 : password : shell

chsh operation: change shell

* most of the time... the default shell is /bin/bash
  - ↳ back in the day it was /ssh/shell

* After login, how does it create a new shell?
  - It forks!
    - ↳ parent: wait()
      - ↳ when child quits, parent takes over and allows process to keep running
    - ↳ child: runs shell
      - ↳ execl("/bin/bash")
        - ↳ shell
      - ↳ when it enters shell it will be set up with root permissions
        - ↳ logind uid = 0
      - ↳ before execl() it needs to set things up to NOT be root
        - ↳ change uid to whoever just logged in
      - ↳ chdir to user home dir (also in /etc/password)
      - ↳ after setup, switch to bash

15: gheith : Ahmed Gheith : 30 : password : shell : home

Now... what if we type ls?

- After the whole line is typed AND you press enter
  - ↳ driver takes the line and gives it to the shell program
    - ↳ tokenizes by white space
      - Assumes the first token is the command name
        - > ls -a abc
        - tokenizes into
          - <ls> <-a> <abc>
          - ↑ command name
          - args
      - There are a handful of built in command
        - ↳ cd is built in
        - ↳ ls is NOT built in
        - ↳ MOST commands ARE NT built in
      - If it is built in --> just run it
      - If it is NOT built in
        - ↳ find a program with the command name
          - ↳ execl to the program to pass the all the args

- how does it find the program?
  - ↳ there is a subset of folders that it will check
    - ↳ PATH = /bin/sbin/usr/bin
      - ↳ some systems use colon instead of slash
    - ↳ if you don't find the program --> error
  - ↳ then it execl to the program
    - ↳ BUT if you execl then you lose the shell

so...

fork() then execl()
- ↳ parent stays on shell and wait()

vfork()
- scary
- fork but don't copy address space...
  - ↳ uses SAME stack as parent
- how do you prevent them from clashing?
  - ↳ child "borrows" the address space
    - ↳ prevents parent from running
    - ↳ until child execl or exit
    - ↳ child has to keep the stack clean
- use very rarely for very small operations
- now... COW is pretty good so we don't use vfork() very much anymore.