History of x86

4004 → 8080 → 8085 → Z80 (8-bit)

6502   6800

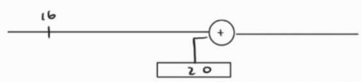
16-bit 8086   toy things - missing a lot
ex. no paging

32-bit 80386

16 bit:

can address 2^{16} Bytes ⇒ 64 KB
NOT A lot of space

you could buy 1MB chips but... there was no way to address it --> 16 bits not enough
need segmentation

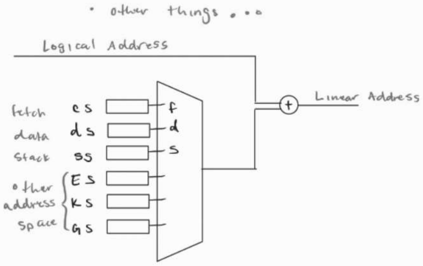


20 bits to point the address somewhere
↳ limit the amt of space for each program but can run multiple programs
↳ how do we use 20 bits in a 16-bit system?
↳ assume lower 4 bits are 0

- each program gets its own 64 kb memory space
  ↳ useful if you want to have multiple programs at a time but... no one really did that
  ↳ pretty much all compers were single user

What if we have 2 registers?
use a different address for fetch and everything else
↳ have different section for program instructions and data --> each 64 bytes(?)
double the address space

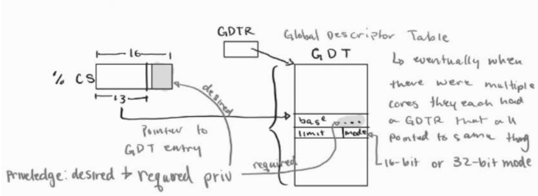
then... we can add more for
- stack
- other things ...



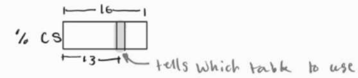
- then there were too many and it got too crazy
  * Note: you couldn't use C in this system

Then technology improved and we got 32 bit Addresses
At this point they could've just used paging but... they didn't

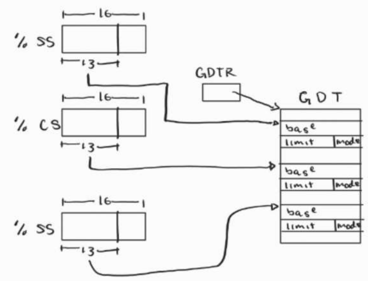
How do we widen the 16 bit value?
- want to keep 16 bits for backwards compatibility
- they... added a level of indirection



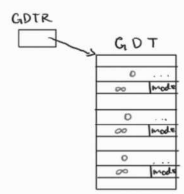
Eventually... created Local Descriptor Table
to have a GDT per process
- how do we know which DT to use?



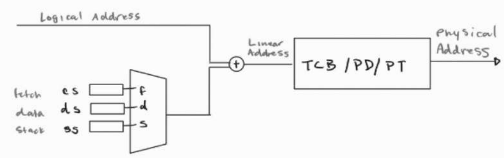
This was DUMB. we don't have LDT anymore



NOW:



we still have this to store mode and priviledge
↳ And legacy reasons

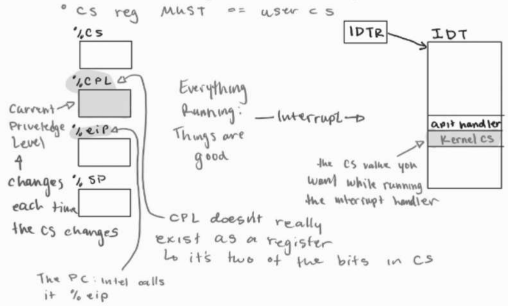


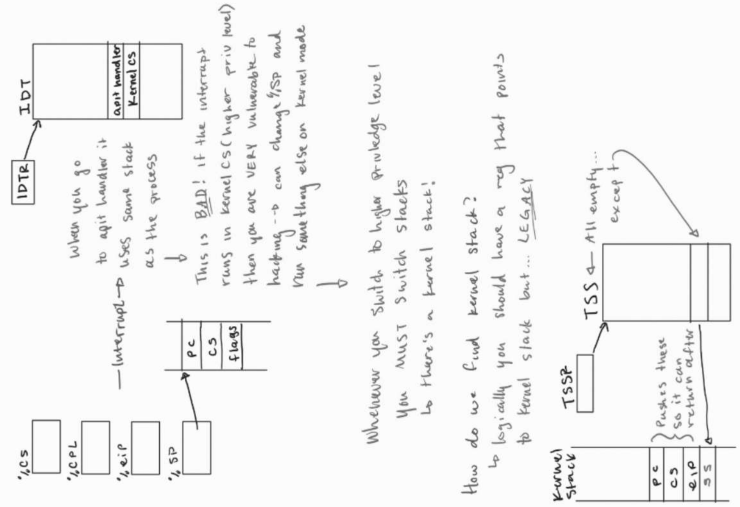
* now... if you run in 64-bit mode you will get an exception if base and limit are not 0 and ∞

USER MODE

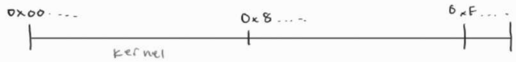
we have 2 values for cs:
- user cs } Points to entries in GDT:
- kernel cs } Both have 0 and ∞ for base and limit but mode is different

when a user program is running...
- cs reg MUST == user cs



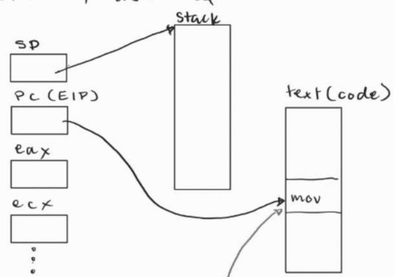


Memory:



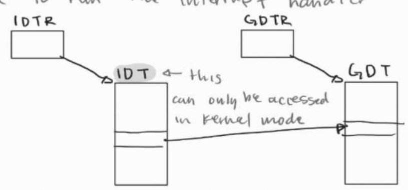
• If a user process tries to access something in kernel memory then the MMU will fail
↳ How does hardware know if it has access?
↓
%CS contains CPL --→ tells if user mode or kernel mode.

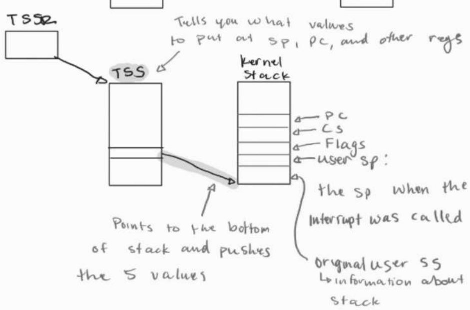
Lets start w/ user mode



What happens if this creates a page fault
↓

• we should flush the pipeline --→ this creates a big security risk --→ easy to exploit
• need to run the interrupt handler





Note: the interrupt handing is NOT instructions. It is in hardware

* If we switch stacks we push 5 regs
* If we DONT swith stacks we only push 3 regs
* SOME interrupts push an extra 6th reg with the error code
↳ It is the responsibility of the handler to take care of the extra reg
↳ iret doesn't know to take the extra reg off.

* Once the hardware finishes:
 - pushing the 5 or 6 regs
 - sets the SP
 - Sets the PC
 - modify the flags as needed
• then you can context switch to the handler
 • change the CPL to 0 --→ kernel mode
 • the handler pops things off the stack
 ↳ Like the extra error reg
 • runs the handler
 • runs iret
• Returns to PC
↳ value to PC depends
 - for page fault PC = the instruction that failed
 ↳ allows the instruction to be run again
 - for other things the PC = the instruction after the failed instruction
 ↳ allows it to continue

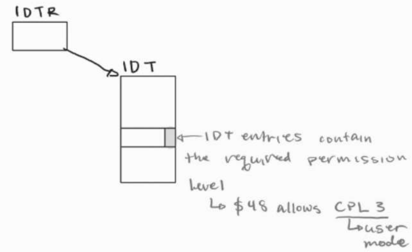
• If running in user mode:
 • There may be a lot of things I want to do that can only be done in kernel mode
 ↳ ex. allocating memory
 ↳ need access to TCBs and stuff which is only available in kernel mode

Solution?
use fake interrupt

int $48

fake interrupt instruction

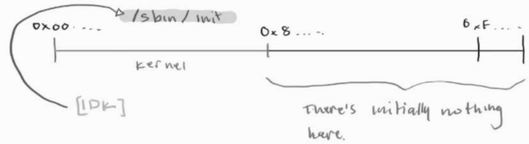
• tells hardware to act as if that interrupt is called



• This still runs everything from before
↳ pushing to kernel stack & stuff

• you still need to tell it what to do
↳ use %eax and put a number
↳ System call: software interupts
↳ different numbers mean different things
↳ the handler checks if you're allowed to perform the operation

Now. How do I enter Usermode in the first place?

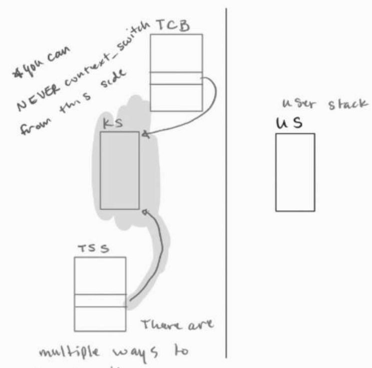


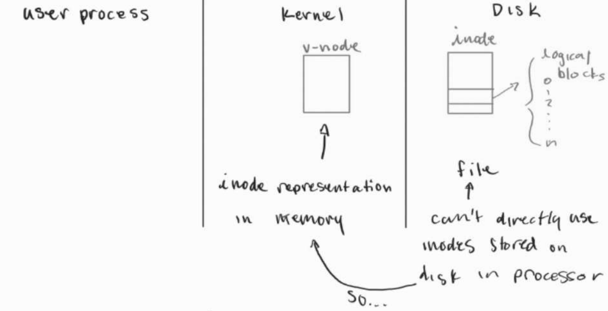
• kernel Main shouldn't be running silly programs
↳ it should be loading/running user code
• kernel needs to mmap the user program
↳ need to set up the world:
 - set CPL to 3
 - set the SP
 ↳ switches the SP but you're still running kernel code but with a user stack
 - Set PC

remember: there's no instruction to set CPL
↳ the only two ways are exception handler and iret

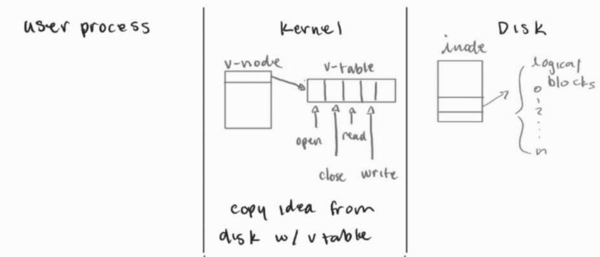
• How do we set these 3 things?
↳ iret conveniently does these 3 things
⇓
push 5 values on the stack then call iret

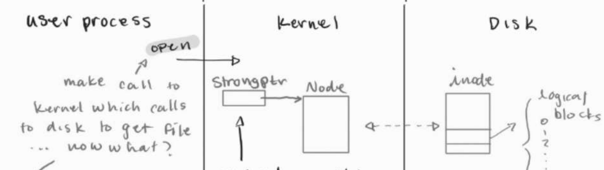
What happens to the kernel Thread?
• It's still there! Kernel Thread is the thing that called iret!
• The kernel Thread has "morphed" into a user process
• When an exception happens... you're still on the same thread!





what if ... I wanted my file system to be not local for my machine
↳ ex. labmachines

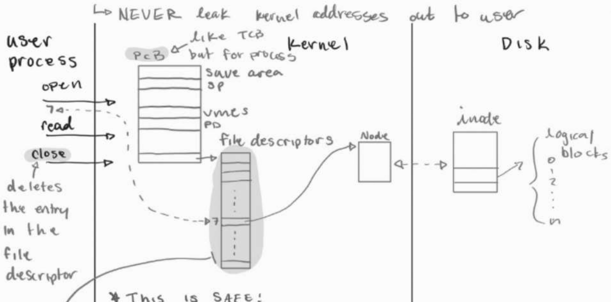




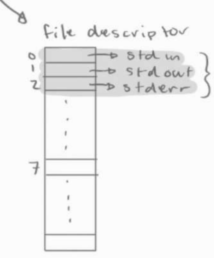
you tend to want to reference count your access to node
↳ coordinate access to Node

what about access in user process?
↳ can't directly read from disk
↳ need to use system call

open() returns the pointer to the memory
↳ the user process shouldnt have access to the file cause it's stored in kernel heap ↳ Protected
↳ what if we only let user process call read() using the pointe?
↳ how do you know if its a pointer you gave user? → you don't. HUGE security risk
↳ * open() CANNOT return a direct pointer → too unsafe
↳ use indirection → open returns a pointer to the pointer to the file
↳ NEVER leak kernel addresses out to user

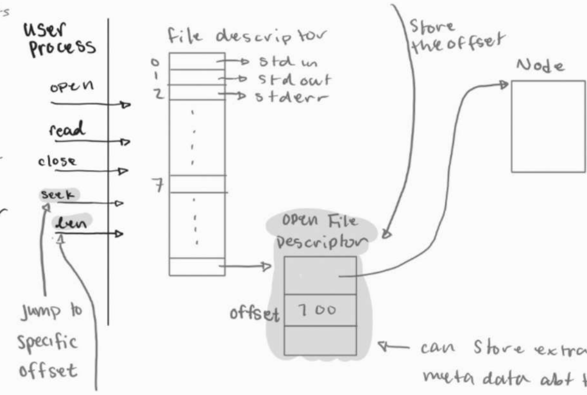


* This is SAFE!
↳ if you try and use a file descriptor value that WASNT given to you, either:
1) the index in file descriptor is invalid and creates an error
2) it will give you some other file that you already had access to.



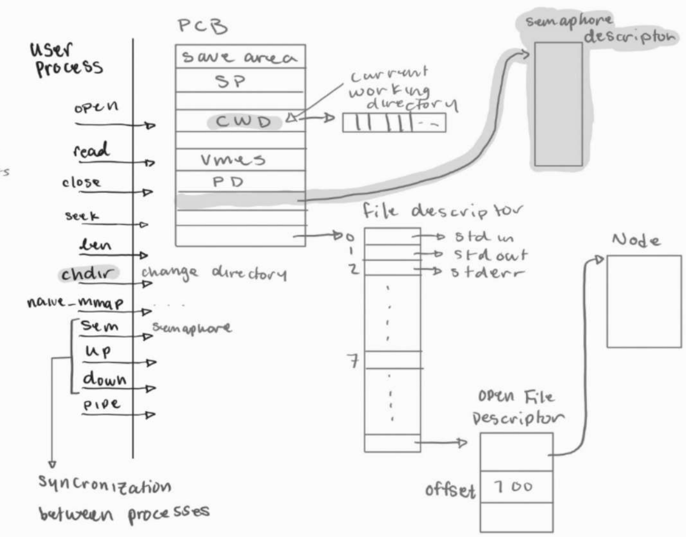
by convention these are at 1, 2, & 3
↳ NOTE: you can absolutely change this (ex. you can close stdin or stdout)

Remember: most of the time we read, we are going to read multiple chunks in order



can store extra meta data abt the file

in our prog:
simplification of 'stat' command
↳ length of file ⇒ size in bytes



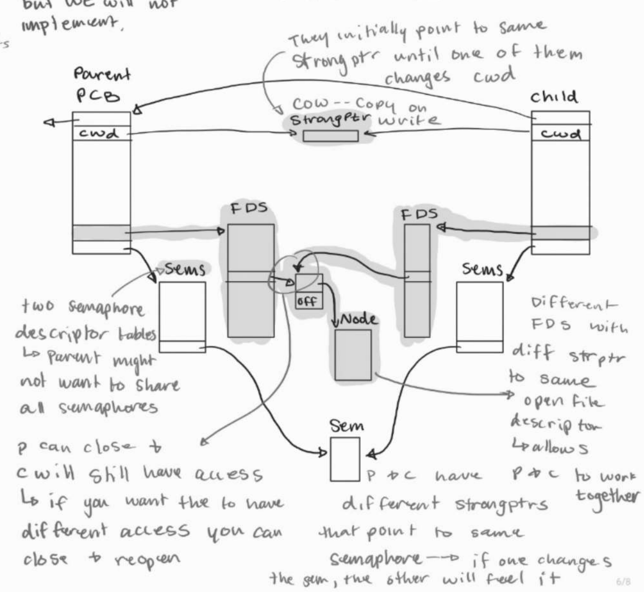
syncronization between processes

Fork! Ψ

int id = fork(); // makes a clone of the process
↳ they have the same history

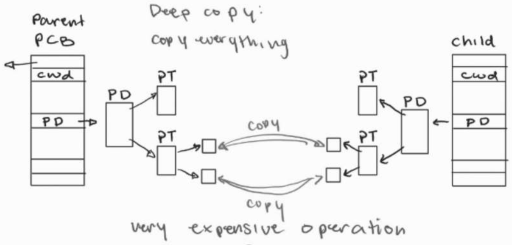
returns int:
0 ⇒ child process
>0 ⇒ Parent -- returned value is id of child
↳ allows you to do things like wait for child
<0 ⇒ Error: you are parent and no child was created

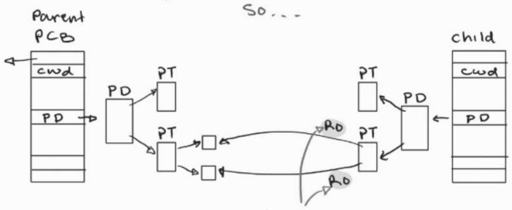
parent knows children but children don't know parents
↳ there is system call for child to get parents but WE will not implement.



What about File system?

Deep copy:
copy everything



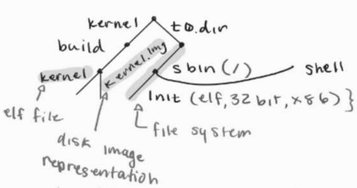


Read only pointer to same file until...
↳ EITHER Parent OR child writes
⇓
then make copy of file
↳ COW --> copy on write

Welcome To User Mode

now:
- We REALLY want to avoid panicing
  ↳ only panic when we REALLY don't know what to do
- Tests no longer have 1 kernel per test
  ↳ it doesn't recompile for each test
    ↳ it just reboots
    ↳ unless you make clean

test cases structure:



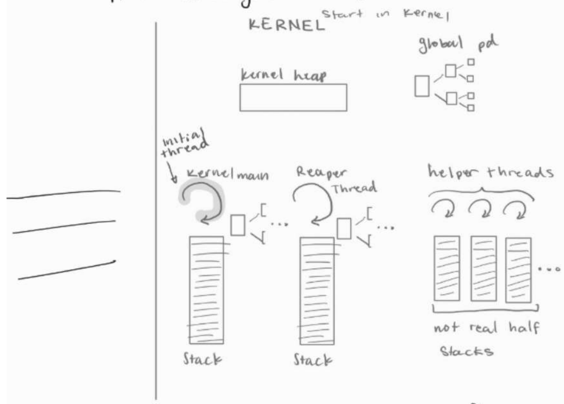
} doesn't matter what is here as long as it includes a executable

- when debugging GDB doesn't know which executable you're debugging so you have to tell it

Notes:
- There's no printf in the test case --> you have to move some printf functionality to t.c and do a write system call

So what do we need in t.c?
- dir directory
- user process has its own memory space
- syscall.cc / syscall.h has the system calls in t0
  ↳ Missing 3 -- Described in README

what happens when you run t0?

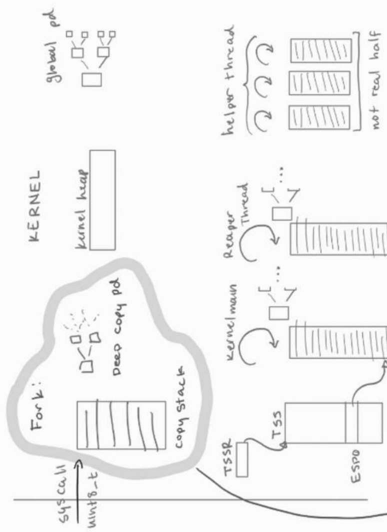


this is the harsh boundary you cannot cross

initializes all this
then looks for elf file
↳ there is a file that loads the elf
elf.cc

Wait:
- parent waiting for child to exit
  ↳ returns the exit code
- you can disown a child
  ↳ close()
  ↳ tells kernel to clean after the child.

Fork



- parent will run on kernel stack
- child will run on its own stack
- the expensive part of the deep copy of pd

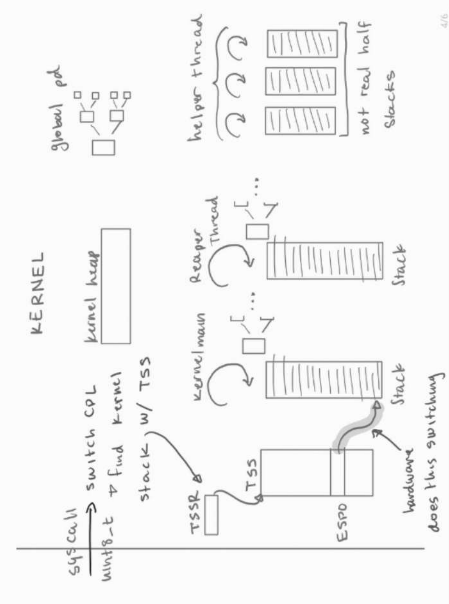
Exec:
execl (path, argv0, argv1, ..., 0)
- stops current program and switches to the program at path
- If successful --> it will NEVER return
  ↳ you need to empty your address space
  ↳ retrieve the new elf file
  - create brand new stack
  - pass the arguments to new program
    ↳ need to load the args into kernel
      ↳ stack before deleting old address space
      ↳ load back to new address space
        ↳ doesn't matter where in address space as long as the stack points to the right address.

↓ If we allocate stack lazily how will the page fault know its from stack?
↳ you can store the location of stack in TCB
  ↳ in page fault if it's close enough to stack the we just give the page

*Note: There will be some upper limit in stack size --> there will be an Ed post

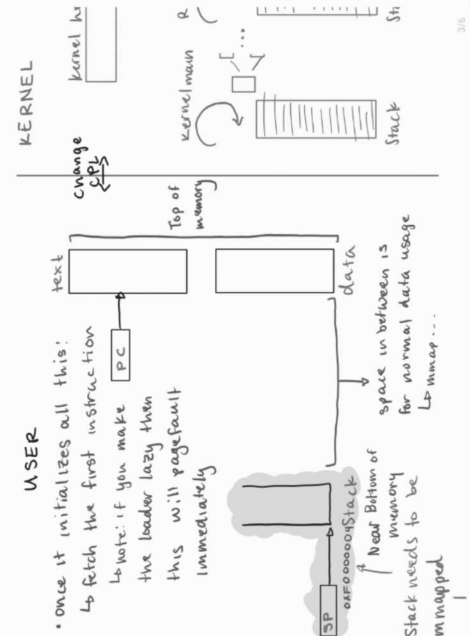
hello love

When you do a system call:



What is the elf file?
- has information on the data of the executable
  ↳ ex: how big, where it is, which va's to use, etc
- The elf file can not handle position independent
- It's a good idea to demand page the program
  ↳ most programs will not run every instruction
- The elf loader is NOT LAZY --> might be a good idea to change it to be lazy
- You will need to modify your mmap implementat to take an optional parameter for va-

User mode:
- Ideally you want the stack to be far far away from your program code



- once it initializes all this:
  ↳ fetch the first instruction
    ↳ note: if you make the loader lazy then this will pagefault immediately