L1: Introduction to Operating System.

- whats an OS?
  - middleware between user programs ! & System Hardware.
    - CPU
    - RAM
    - ROM
    - various registers
    - external devices via drivers.
  - manages hardware:
    CPU, main memory,
    IO devices (disk, NIC, mouse etc)

* when we run a program:-

1. compiler translates high level program into executable.
   a.out
   a.exe
2. the .exe contains instructions & data of the program. (all addresses!).
3. CPU hardware is instructed via ISA. Instruct^n set architecture.
4. CPU has registers:-
   PC
   instruction Code
   Memory address

- wehn we run a program:-
  1) read instruction at PC
  2) load data
  3) execute Instruction
  4) Store results.

  most recent data & instruct^ns are cached at CPU for faster results.

→ so what OS do?

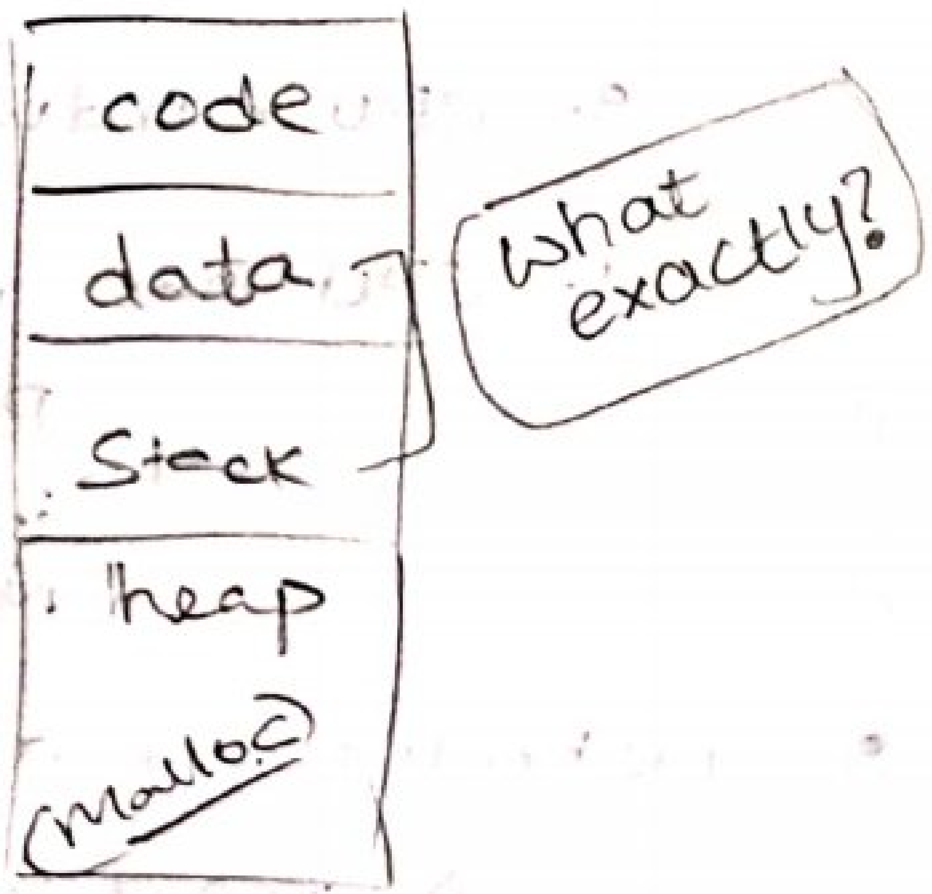
1) manages program memory
   - Loads program executable from disk to memory. ram!
     (code, data)
2) manages CPU
   - initializes PC & other registers.
3) OS manages external devices
   - Rd/wr from files from disk.

I) OS manages the CPU:-

- * Operating system provides process abstraction.
- * OS creates & manages processes.
- * each process has the illusion of having the CPU to itself.
  virtualizes CPU.
- * Timeshares CPU among processes.
- * Enables co-ordination b/w processes.

II) OS manages the memory:-

- * OS manages the memory of the processes:
  code, data, stack, heap



- * each process thinks it has seperate space, numbers code and data starting from 0.
  virtual addressing.
- * OS abstracts the actual placement in memory. Translates from virtual addresses to actual addresses.

III) OS manages the devices:-

- * OS has code to manage disk, NIC, other devices — device drivers.

Eg: persistent data organized in disk.

* OS evolved from running a single program to multiple processes simultaneously.

L-2: The process abstraction:-

hmmm...

- OS provides a process abstraction.
- OS has a CPU scheduler that picks up one from many active processes to execute on a CPU:

  → Policy: which process to run

  → mechanism: how to "context switch" between processes

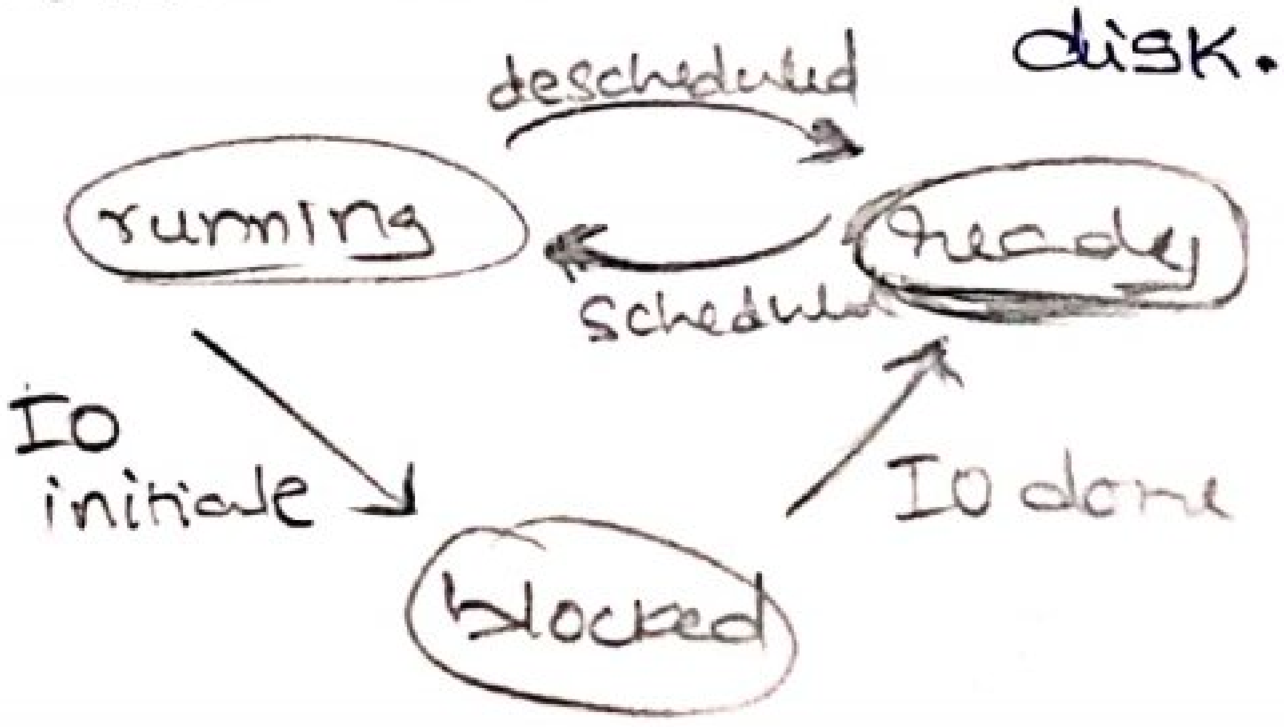
* A process constitutes:-

- a unique process identifier PID.
- memory image
  - code & data (static)
  - stack & heap (dynamic)

  ] ✓✓
- CPU context (registers)
  - PC register
  - current operands
  - stack pointer.
- File descriptors
  - pointers to open files & devices.] STDIN STDOUT STDERR

* State of a process:-

- running
- Ready (waiting to be scheduled)
- Blocked : suspended, not ready to run.
  - waiting for some event; like issued a read from disk.
- New : being created.
- Dead: terminated



* OS data structures. Maintains a data structure (List etc) Of all active processes.

- Each process's info in a PCB (process control block).
    - PID
    - Process State
    - pointers related to other processes (parent process)
    - CPU context Of the process
      [PC, stack pointer, current operand].
    - pointers to memory locations
      (like the absolute position of memory image?)
    - pointer to open files.
      cool.

* This datastructure won't contain the memory image of process dummy! thats in the memory itself.

This contains position of that.

L21 - xv6 introduction :- & x86 background.

xv6 → O.S.

x86 → ISA family.

→ xv6 is a O.S. used for teaching

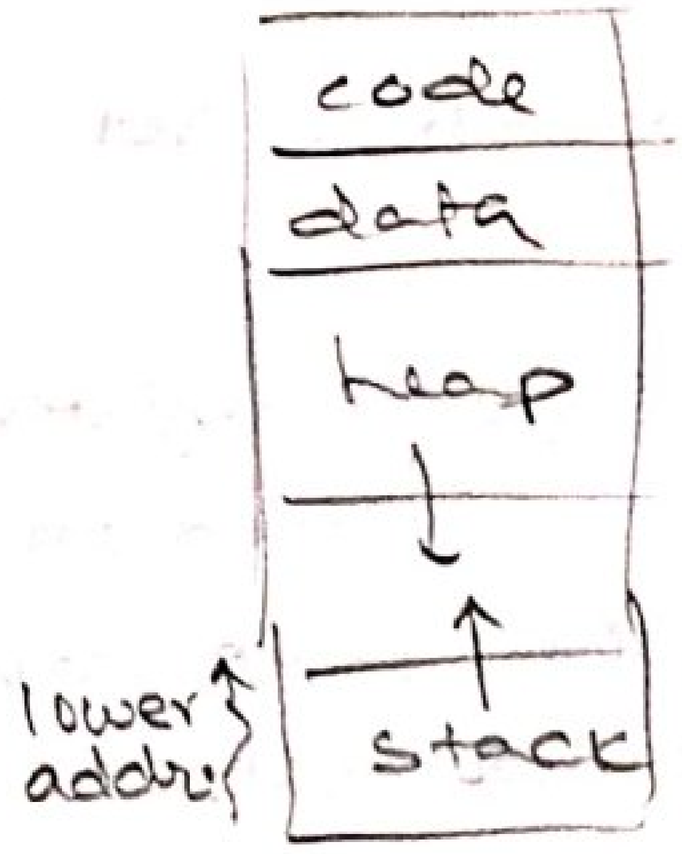
-has 2 versions; for x86 hardware and one for RISC-V hardware

- we'll learn x86 version. (lets see x86 basics).

→ memory image of process:-

* consists of ~~compiled code~~ :-

- compiled code
- Global / Static variables (memory for these, is allocated at compile time)
- Heap (grows on demand)
- Stack (temp. storage during func calls etc.) local vars. grows 'up' towards lower addresses
- Others like shared libraries



• x86 registers & example instructions:-

1) general purpose eax, ebx, ecx ..

2) eip

3) esp, ebp

esp → stack ptr.

ebp → base ptr.

4) cr3 → metadata; pointer to page table.

5) Segment registers cs, ds, es, . . .

mov %eax, %ebs

mov (%eax), %ebs

push %eax onto stack.

pop %eax send stack.top( ) into eax.

] both of these modify esp.

jmp %eax

* levels of privelage. (0 to 3)

0 → OS code

3 → user code

Eg: setting cr3 reg, io devices | load value to eax.

* user should request OS services (syscall) for high privilege instructⁿ

→ Function calls and stack:-

* What happens:-

1. push arguments to stack
2. "call" fn (this pushes current eip onto stack & jumps)
3. Allocate local vars & complete function.
4. "ret" (this pops the return address & jumps back)

* Register values get clobbered na!
of CPU

1) caller saved registers:-
→ saved on stack by caller before invoking the fun.
→ callee code can freely change them.
→ caller restores these registers after return.

2) callee saved:-
→ caller expects these registers to have same value before & after Function invocation.
→ saved by callee function & restored as the function ends.

automatically done by C-compiler.

L22:- processes in xv6

PCB → process control block

* the list of all PCBs is critical kernel datastructure & maintained in kernel memory.

struct proc in xv6
task-struct in Linux

PCB has:-

- size of memory for proc
- pageTable pointer.
- kernel stack pointer
- state of process.
- ~~PID~~
- list of open files.
- process name (for debugging)
- curr dir.

* in xv6; process states are

UNUSED
EMBRYO
SLEEPING ✓ (blocked)
RUNNABLE ✓
RUNNING ✓
ZOMBIE ✓ after killing a process.

1) Kernel stack:- * for syscalls from process

* when a function is called in process; user stack stores stuff.
* but when process calls syscalls to run kernel code;

CPU context stored on kernel stack (security).

- this seperate area for each process on kernel stack not accessible by users.
- the link to this Kstack is through struct proc of process.

2) List of open files:-

* Array of pointers. when process opens a new file; a new element is created; & its index is passed as FD. (file descriptor).

* First 3 elements of list are open by default:

stdin, stdout, stderr.

3) Page tabel:-

- * every entry in memory image has an address.
  - virtual address starting from 0.
  - actual physical address will be different.
- * Page table maintains mapping from virtual address to physical address.

nice!
(more on this later)

→ process table in XV6:-

```
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

- * fixed size array of all procs.
  (practically, might have dynamic size).
- * CPU scheduler loops through ptable & sets a runnable process to running.

→ Process state transition:-

i) A process that needs to sleep will set its state SLEEPING & invoke scheduler.

ii) A process which has ran for its fair share will set its state to RUNNABLE & invoke scheduler.

L3:- process API

* OS API is provided by a set of "system calls".

- syscall is a function call into OS code which runs at higher privilege levels.
- access to hardware, is allowed only at higher privilage levels.

* POSIX API :

in order to maintain code portability over various OS; all OSs have to be POSIX compliant.

* programming lang. libraries hide the details of these POXIS calls

Eg: printf calls the write system call

→ process related system calls (in UNIX):-

- fork() creacts a new child process
  - All processes are created by forking their parent. except init process. init is ancestor of all processes.
- exec() replaces the complete memory image of process,
- exit() terminates a process.
- wait() causes a parent to block, until child terminates.

→ fork:-

* A new process is created by making a copy of parent's memory image. This new process is added to ptable & set to runnable.

* the return value from fork() = pid of child ; in parent process
= 0 ; in child process.

Eg:

```
int pid = fork();
if (pid==0){
      print "child";     ] child prints this.
}
else
   print "parent";       ] our parent prints this
```

→ wait for children todie...

* process termination scenario:-

1) exit() syscall. (called automatically, when end of main)

2) OS terminates misbehaving process.

- Terminated processes are state - ZOMBIE. waiting to be reaped by their parent process.

* when parent calls wait(); its zombie are cleaned.

* if a parent terminates before child; the init adopts the child.

→ exec():-

- After forking, parent and childing are running the same code. meh! Not too useful.

* A process runs exec to load another executable into its mem img.

→ How does a shell work:-

- init process created just after bootup.

init
 └fork─ Shell/bash process

& then this shell, after taking user command, forks a child & uses exec() to run command executables.

the commands like wc
ls
are all executables.

* if we wanna redirect output to a txt file;

- spawn a child
- close Std_OUT & open the file.
- call exec().

this'll just update mem image.
won't change open files.
nice! stored in struct proc na!

L23:- System calls for XV6:-

→ what happens on a system call?

* system calls are avaliable to user programs, defined in user librar header "user.h".

* system call implementation, invokes a special ISA instruction called "trap" instruction, called "int" in x86 ISA. (in file usys.S)

- This 'int' instruction causes a jump to kernel code that handles the system call.

System call number is stored in eax register.

1) fork() call:-

- new process, set to runnable, returns PID to parent, 0 to child.

achieved by Setting $eax to 0. in child.

2) Exec():-

- copy new executable into memory
- new stack, heap.
- Switch process page table to use new memory img.

★ 3) Exit():- ★

i) exiting process cleans up state. (Eg. close the files)

★ ii) Pass abondoned children to init. (orphans) ↳ the non-zombie ones...?

(iii) mark itself as zombie & invoke scheduler. sched()

4) wait():

- search for dead (zombie) child in ptable & clean up.
- if no zombie; wait for one to die.
- if no children exist; return -1. !!

XV6 code in Slides. L23.

usys.S

```
#define SYSCALL(name) \
  .globl name; \
  name:
     movel $SYS, %eax;
     int $T_SYSCALL;
     ret

SYSCALL( fork)
```

name variable.

into eax

trap instruction

* every trap instruction has

param n in "int n"

param in $eax.

n → unique to a device/ T_syscall etc..

$eax → more Specific call

L4- mechanism of process execution:-

Low-level mechanisms:- (Bye bye 10K feet view)

- how OS runs a process
- how OS handles a system call
- how OS context switches from one process to another.

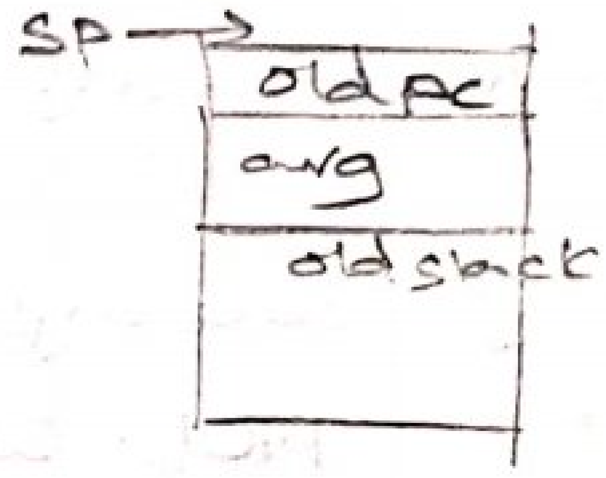
1) OS handling a process:-

sequential instructions via PC. have $sp, $bp

→ when function call happens:-

1) function call means jump instruction.
2) A new stack frame pushed onto stack & SP updated
3) old value of PC (return value) pushed to stack & PC updated.
4) Stack frame contains return value, function arguments etc.

old PC value



→ when system call happens:-

"we don't know the address of the sys.call instructions!"

* Kernel doesn't trust user stack-

uses a seperate kernel stack in kernel mode.

Seperate for each process.

* kernel doesnt trust user provided address to jump to

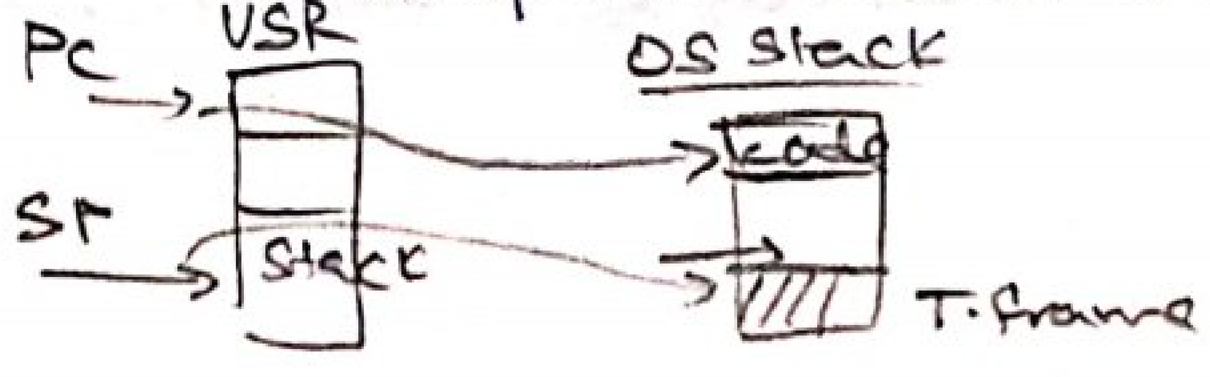
- Kernel sets up Interrupt Descriptor Table (IDT) at boot time.
- IDT has addresses of kernel function instructions to go to, for system calls.

* when syscall is made, a trap instruction is run. with syscall code saved; to lookup IDT-

Baked into silicon.

Trap instruction:-

- Moves CPU to higher priviledge level.
- Switches to kernel stack
- Save trapframe (old PC, registers) on kernel stack.
- lookup addresses in IDT & jump to trap handler function in OS code



→ Trap instruction:-

- trap instruction is executed on hardware in the case of

- System call (by user program)
- Program fault (program access illegal memory) segfault.
- ★ // Interrupt (external device needs attention of OS)
  or timeshare interrupt from O.S. Eg: network packet arrived on NIC.

* IDT has many entries:-

syscall/interrupt store an ID in CPU register before calling trap;

→ Returning from trap:-

* when OS is done with syscall/Interupt; it calls an instruction called return-from-trap. Baked in silicon

- reverse everything done by trap instruction.

* Must you always return back to the same process?
NO! · OS first checks if it should switch.

Why switch?

- Sometimes OS can't return back to same process it left
  - process has exited or terminated (seg fault)
    1. syscall   2. program fault.
  - process made a blocking system call.
- Sometimes we want to timeshare CPU.

In such cases OS performs CONTEXT SWITCH.

OS scheduler:-

Policy:- which to pick.
Mechanism:- How to pick

Policy:-

- Non-preemptive (co-operative) schedulers: switch, only when running process is blocked/terminated
- Preemptive (non coperative) schedulers: CPU generates periodic timer interrupts
  - after servicing the interupt, OS checks if process has ran for too long.

Mechanism:-

context switch happens; when both processes are in Kernel mode.

Now;

- Save context (PC, register, kernel stackpointer) of A on kernel stack.
- Switch SP to kernel stack of B.] stored in struct proc.
- Restore context from B's kernel stack.

  [B's format would be similar; which was previously set up by OS itself or even allocproc()].
- call return-from-trap instruct^n.

* Trapframe vs switch context:-

when going user code → kernel code:-

trap frame stored by trap instruction.

a single, special instruction

when switching:-

context switching code stores context.

L24:- Trap handling in xv6

* Traps:- when OS wants to switch; it traps user process.

- System calls
- Interrupts | every device has IRQ number
  ↳ interupt request :
- program faults.

* in Usys.S, "int" instruction is invoked.] for syscalls.

- for hardware interrupts, device sends signal to CPU & CPU executes 'int'.

* Trap instructn has a(param n) to indicate type of interrupt. syscall & Keyboard interupt have different value for n.

* The following happen as part of int(n):-

eip & esp are pointing to usercode & user stack previously

1) Fetch n^{th} entry from IDT. (CPU knows location of IDT)
2) Save esp into internal register.
3) Switch esp to kernel stack of current process. (CPU knows this)
4) On kernel stack, save old esp, eip
5) Load new eip from IDT to kernel traphandler.

Also, syscall trap instruction can access syscall IDT but not disk-interrupt IDT. (made sure via CPU priviledge levels)

* Trapframe:

State pushed onto kernel stack during trap handling.

- CPU context of where execution stopped is saved. (so as to resume after trap)
- Some extra information needed by traphandler.

* The intn instructn has only pushed the bottom few enteries.

- The traphandler kernel code will push the remaining.

<lec24> <pg5>

the C structure just indicates the complete structure of trapframe; bottom half of which is built by int instrutn & top half built by alltraps kernel trap handler.

→ Kernel trap handler :- (alltraps) assembly code.

- IDT enteries for all interrupts will set eip to point to the Kernel trap handler "alltraps".

four pushed by int'n instr".

- "Alltraps" assembly code will push remaining registers to complete trapframe on kernel stack.
  - "pushal" pushes all general purpose registers.
- invoke C traphandling function named "trap".
  - push pointer to trapframe (esp) as argument to trap().

we have a mix of assembly code & C code. ☹

→ C trap handler function:- trap (struct trapframe *t)

this was passed via assembly code.

- C trap handler; written in C; invoked in assembly.
- if in int(n); n=="T_SYSCALL" (as in usys.S); indicating this trap is system call.
- Trap handler invokes common system call function
  - looks at call no. stored in eax & calls the corresponding funt".
  - return value of syscall stored in eax. (fork or exec or ----)

hence in fork(); we make eax 0. in child.

myproc() returns the current struct proc.

check myproc()→tf→eax & call the corresponding syscall.

& store return value in same eax.

* seperate code to handle interrupt from devices.

(each device has different n for "int n" insruction).

* Timer is special hardware interrupt, & is generated periodically to trap to kernel.

On timer interrupt, a process yeilds CPU to scheduler.

```
if (myproc() && myproc()->state== RUNNING &&
        tf -> trapno == T_IRQ0+ IRQ_TIMER)
    yeild();
void
yeild (void)
{
  acquire(&ptable.lock);
  myproc() -> state= RUNNABLE;
  sched();
  release (&ptable.lock);
}
```

→ Return from trap:- trapret:

- pop all state from kernel stack.
- instruction "iret" does the opposite of "int" instruction.

  int → i enter

  - changes privilege levels
  - pops values pushed by int. (esp & eip)
- execution of user code resumes.

L25: Context Switching in xv6:-

* every CPU has a scheduler thread (special process, running the Scheduler code)

* After running for some time, a process switches to scheduler when

- process terminated
- process wants to sleep (blocking sys call)
- process yeilds after running for a longtime. (timer interupt)

* context switch happens only in Kernel mode.

P1 running user —interupt→ P1 running kernel → Scheduler → P2 running Kernel → P2 running user

→ **scheduler() and sched():-**

* Scheduler thread shifts to a process via scheduler().

* user process shifts to scheduler thread via sched(). invoked from exit, sleep, yeild.

Both have swtch (& context*, context*)...

→ **Struct context:-**

```
struct context{
   uint edi;
   uint esi;
   uint ebx;
   uint ebp;
   uint eip;
};
```

Set of registers to be saved; when switching processes.

* In both scheduler() & sched(); switch() switches between two "contexts".

* context is pushed onto kernel stack. struct proc maintains a pointer to the context structure on stack. p→context.

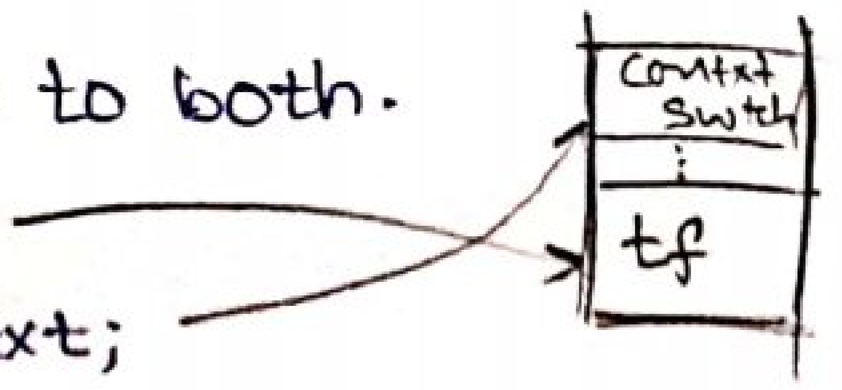
→ **Trapframe vs context:-**

- traframe saved, when CPU switches from user to kernel mode. eip in trapframe is when syscall was made in usercode.
- context structure is saved, when CPU switches from process to scheduler. eip in context structure is when swtch() is called. (In kernel code)

→ struct proc has pointers to both.

struct trapframe *tf;

struct context *context;



→ Swtch function:-

- This is the one; which actually creates the context struct.
- Both CPU struct & proc struct maintain a context struct pointer (struct context*)
- Swtch takes 2 arguments

  ↱ since we need to update this pointer.
  address of old context pointer, to switch from;
  new context pointer to switch to; ↰ we just need to read this context.

- when invoked from scheduler();

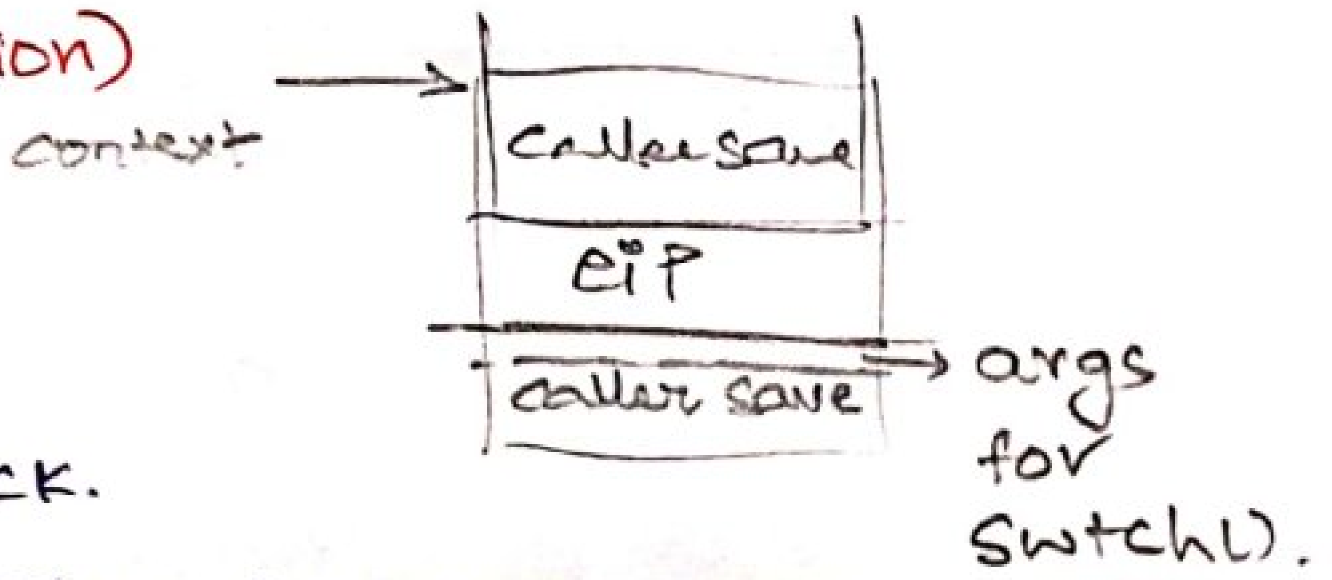
  swtch(&(c→scheduler), p→context);

  when invoked from sched();

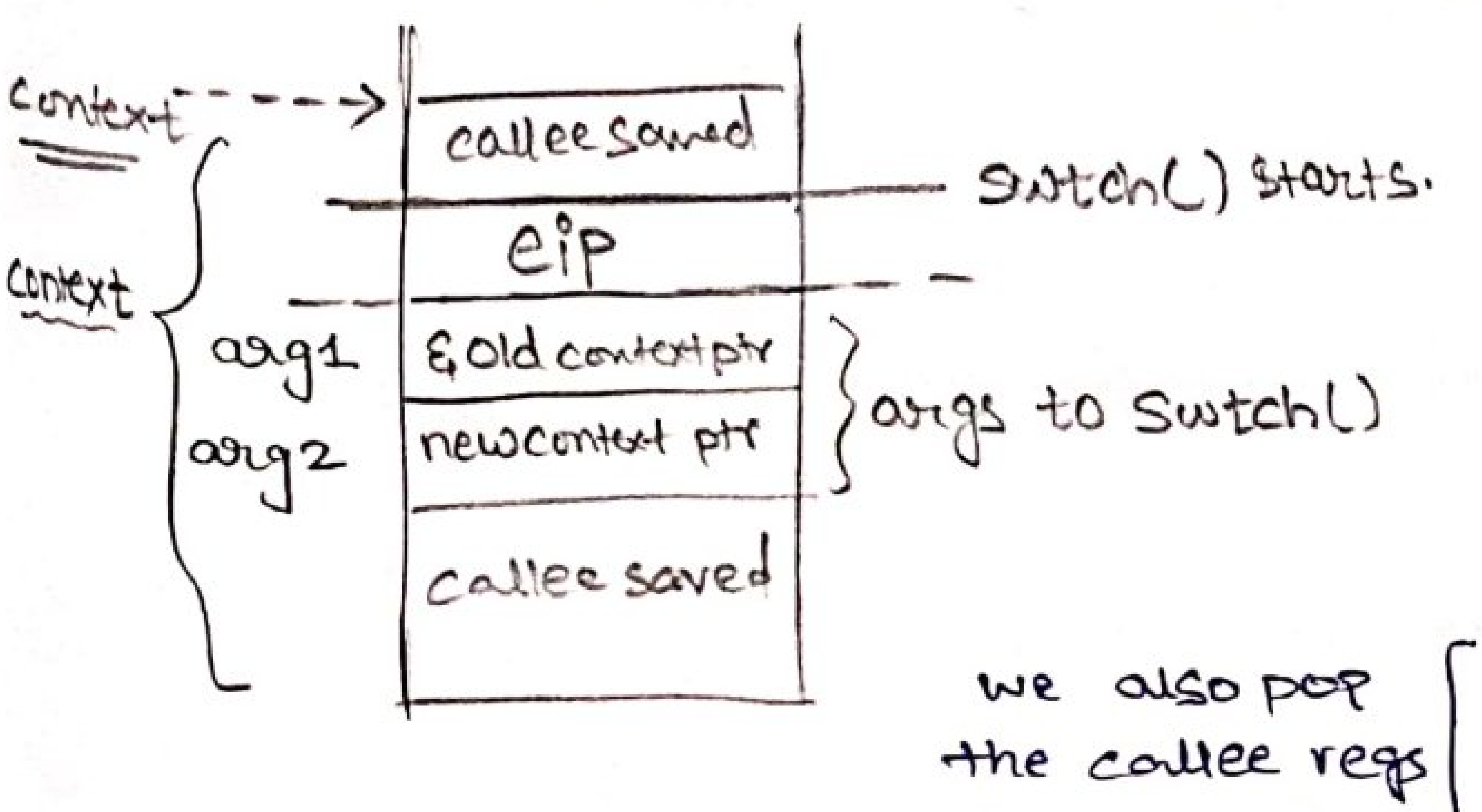
  swtch(&(p→context), mycpu()→scheduler);

---

1) what is on kstack, when a process has just invoked swtch();
   - caller save registers (c calling convention)
   - Return address (eip)



2) What does swtch do?
   - push callee saved regs onto the kstack.
   - save pointer to this context in the struct proc → context.
   - switch esp from old kernel stack to new kernel stack.
   - pop callee saved registers from new stack
   - return from function call.



movl 4(%esp), %eax;
movl 8(%esp); %edx;

we also pop the callee regs & return from swtch() in the new process.

<l25 p9> assembly code for swtch.

Tough to write in C. ☺

L26: process creatn in XV6:-

what's this?

* init process: first process created by xv6 after boot up.

- This init process forks shell process; which inturn forks all other user processes.
- init is ancestor of all other processes, in unix-like systems.

* allocproc() is called during both init process creation & in fork system call.

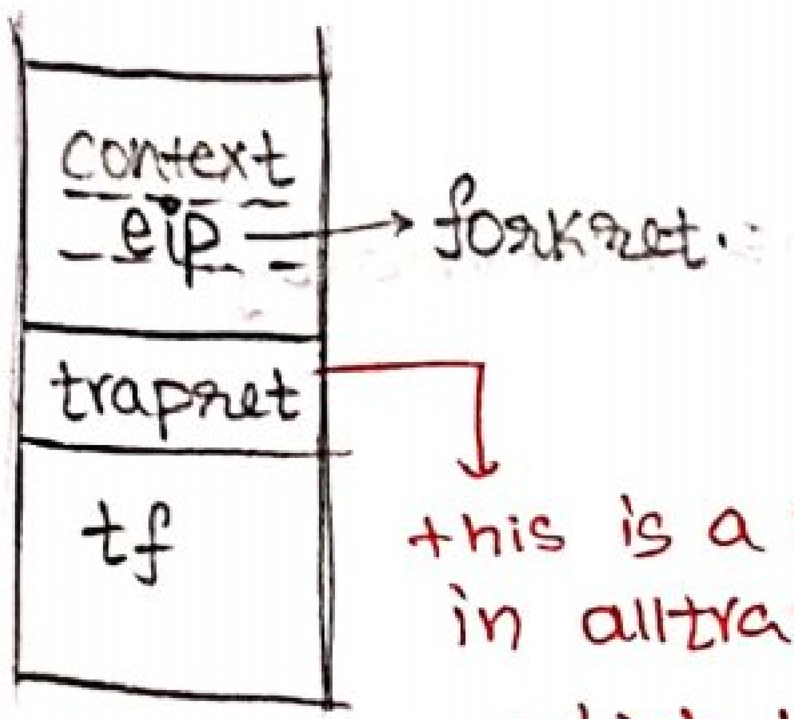
- Allocates new struct proc, PID etc.
- [IMP] sets up kernel stack of this new process so that it is ready to be context-switched in by scheduler. COOL.

→ Allocproc:-

1) Find unused entry in ptable(). mark it as embryo.
   Status = UNUSED. (mark as runnable, after creation completed)
2) New PID allocated
3) New memory for kstack allocated.
   Kalloc() will return first byte addr.
4) Go to bottom of stack. Leave space for trapframe (move on this later)
   I think fork() fills in.
5) push return address of "trapret".
6) push context structure, with eip pointing to "forkret"

- When this new process is scheduled; it begins execution at forkret (in kernel code) then returns to trapret (in kernel code); then returns from trap frame to user code. 😀

"we created a hand-crafted kstack, to look like the process was trapped & context-switched in past."



this is a piece of code in alltraps.S which takes us from kernel to user code after poping some & calling iret instruct^n.

L26
Page 3

awesome

Allocproc code.

intertwine assembly and c.

→ Init process creation:-

→ Allocproc() has created a new process.

Trapframe of process set to make process return to first instructn of initcode.S in userspace.

But isn't initcode.S kernel?

* the initcode.S simply performs "exec" to run the init program.

*. init program opens STDIN, STDOUT, STDERR files.

- inherited by all subsequent processes; as child inherits parent's files.
- forks a child, execs a shell.
- reaps dead children (its own or other orphan descendants).

<read L26, P 4,5 code>

→ Forking a new process:- (more technical view)

- fork() allocates new process via allocproc().

2) parent memory & files copied.

3) Trap frame of child, copied from parent.
   - Hence, child returns to the exact same line of code as parent.
   - different physical mem. addr. but same virtual memory address.
   - only the value in eax (syscall return values are stored in eax) is set to 0 in child.

4) set the child to runnable.

5) parent returns from fork() syscall & runs normally.

<L26, Pg6> fork() code beautiful.

L5- Scheduling Policies:-

preemptive: willing to stop in middle & large process.

* Scheduler has 2 parts — policy now!
└ mechanism/

→ which process to run next, from a set of processes.

* OS scheduler schedules the CPU requests (bursts) of processes

- CPU burst = CPU time used on a single strech, by a process.

* What are we tryn' to optimize?

- Maximize Utilization (= fraction of time CPU is used)
  eazy. Always.
- Minimize average turnaround-time
  (= time from process arrival to completion)
- Minimize average response time
  (= time from arrival to first execution)
- Fairness: all processes must be treated equally
- Minimize Overheads: run processes long enough to amortize cost of context switch (~ 1 μs)

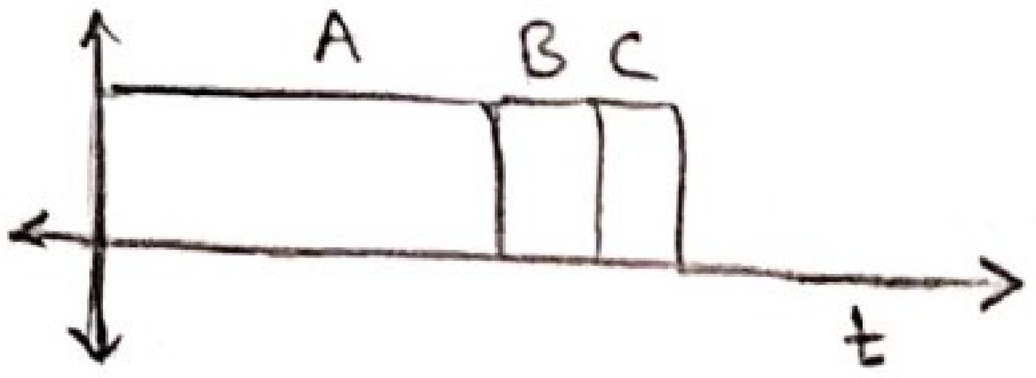
a) FIFO Scheduling:-

* Schedule in the order arrived.

* Problem: convoy effect

- A is too big. B,C mustn't wait

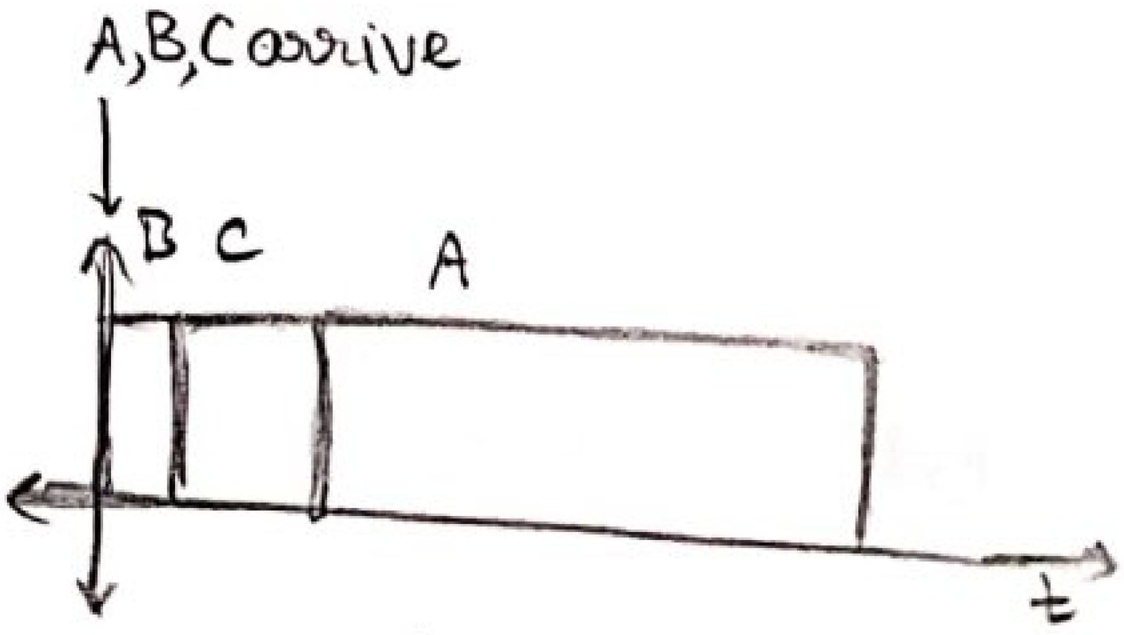
- High turnaround times.
  Also response times!



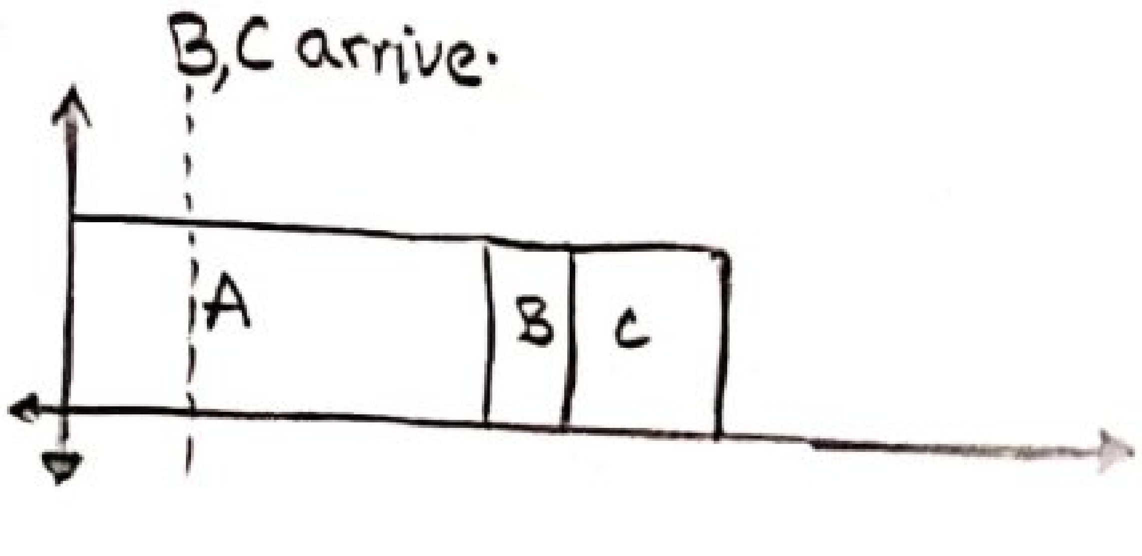
ABC arrive at t=0 in order A,B,C.

b) Shortest Job first:- (SJF)

* provable optimal avg. turnaround time, when all processes arrive together.



* SJF is non-preemptive. So short jobs stuck behind longer ones; if they arrive after longer ones.

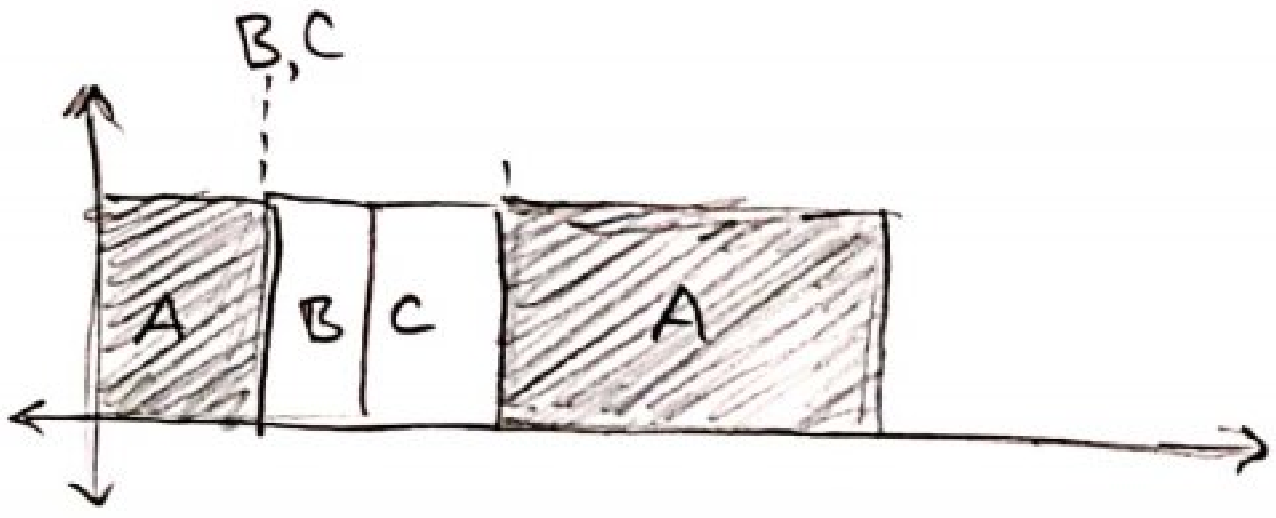


c) Shortest time to completion first :-

(STCF)

* Also, Shortest remaining time first. (SRTF)

- premptive scheduling.
- preempts running task, if time left is more than new arrivals.

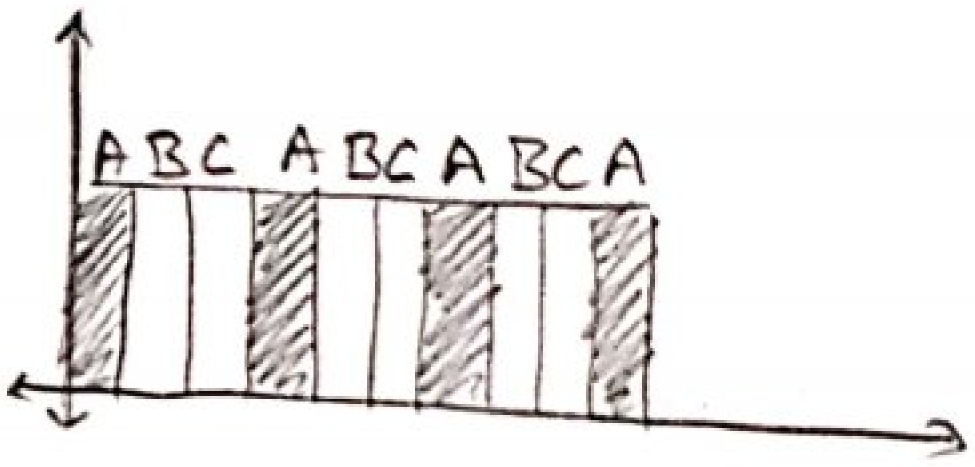


Checking shortest when new procs arrive.

d) Round Robin:-

* every process executed for a fixed quantum slice.

(keep slice big enough to amortize context switch cost)



* preemptive.

* Good for response time & fairness

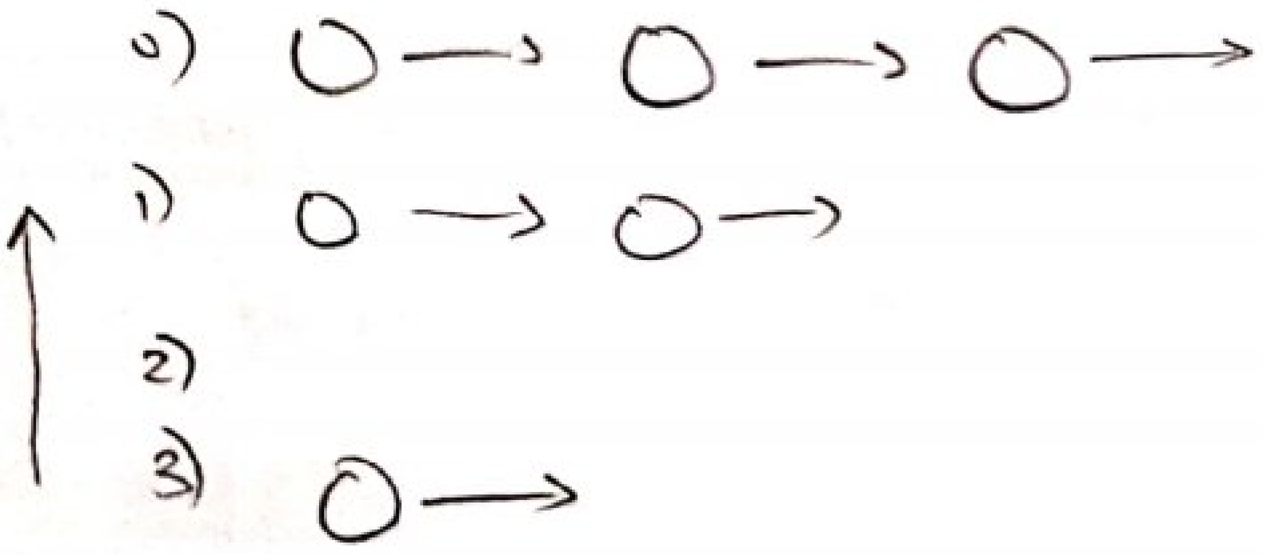
* Big blow on turnaround time.

e) Unix:-

* Schedulers in real system are more complex.

* Unix uses a multilevel feedback Queue (MLFQ)

- many queues, in priority order.
- processes from highest queue scheduled first
- within same priority, any algo like RR
- priority of process decays with age.



L6: Inter-process Communication (IPC):-

- each process has its own unique memoryImage.
- If two processes want to work together, they need to use IPC mechanisms.

1. Shared memory:-
   Shared memory get.
   * shmget() System call.

     int shmget (key_t key, int size, int shmflg).

   - same key means, both processes have same segment of memory.
   - Need to take care one process is not overwriting another.

2. Signals:-
   * Either the OS or a process, can send signal to another process.

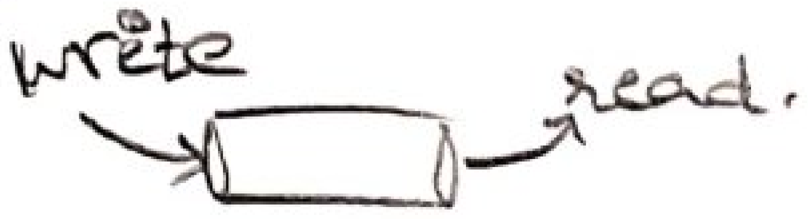
     Ctrl+C → SIGINT.

   * Some signals can't be overridden. Eg: SIGKILL. Some can be.
   * Signal handler: every process has default code for signal handling.

3. Sockets:-
   * can be used by processes on same machine or different machine to communicate.
     - TCP/UDP sockets across PCs
     - Unix sockets in local machine

   * OS transfers data across socket buffers.

4. Pipes:-
   * pipe system call returns 2 file descriptors.



     - read handle & write handle
     - A pipe is a half-duplex communication.

   * Regular pipes: both fds are in same process.
     - parent & child fd after fork.
   * Named pipes: two endpoints can be in different processes.

   * OS buffers pipe data b/w read & write.

5. Message queues:-

* mailbox abstraction

* process can open a mailbox & send/recieve messages from mailbox.

* OS buffers the intermediate mails.

* Each system call read/write have blocking & non-blocking versions.

blocking:
- read from empty queues
- write to full queues.

non-blocking: returns with a value or an error code.

L7: Intro to virtual memory:-

* why virtualize memory:-
  - Bcoz real view of physical memory is messy!
  - Multiple processes are stored, all jumbled up.
  - Need to hide this complexity from user.

* Every process thinks as if it has access to a large space of addresses from 0 to MAX.

* CPU issues load store instructions with virtual addresses.

* MMU memory management unit translaters virtual addresses to real addresses.
  present on CPU itself!

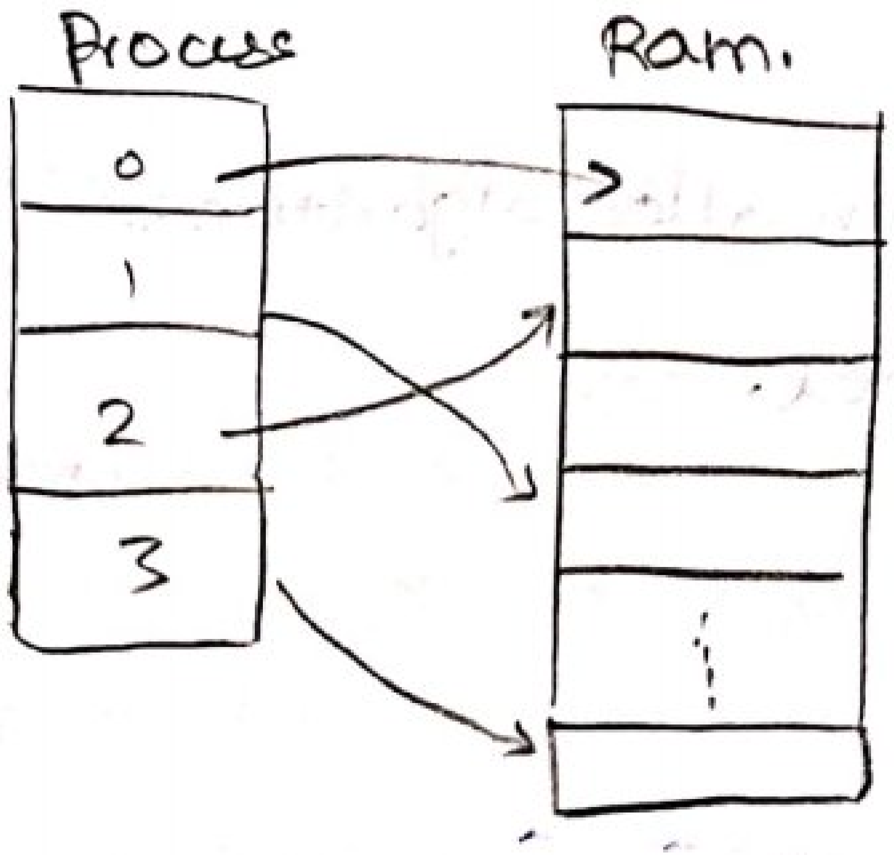
  OS makes the pagetable avaliable to MMU.

→ The concept of paging:-

* The virtual address space is divided into fixed size segments called pages. Also the physical addresses are divided into page frames.

* To allocate memory to a process, pages are mapped to frames.

* The pagetable stores mapping from virtual page number to physical frame number.
  first 12 bits or so.... of address.



4KB pages...
1KB pages...
etc.

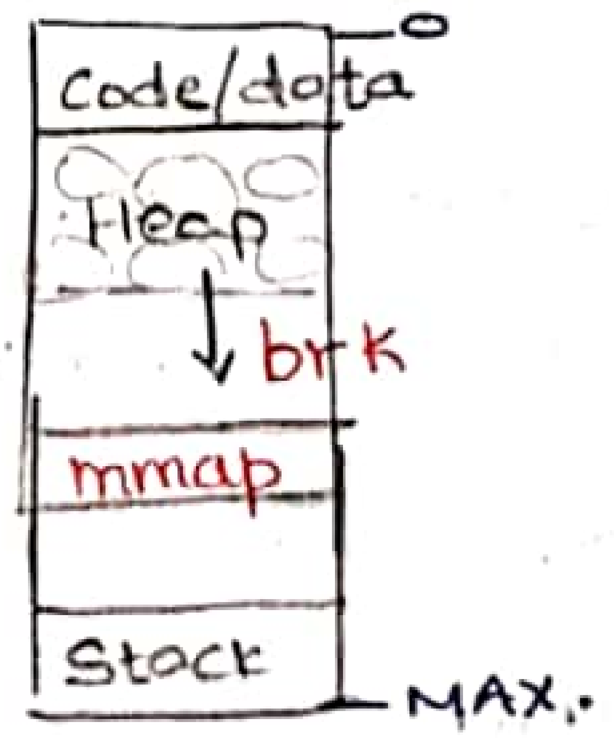
* Goals of memory virtualization:-
  - Transparency: user not aware of messy details
  - Efficency : minimize overheads in terms of memory & access time.
  - Isolation & protection : user should not access anything outside his address space.

Cool!
All 3 are handeled.

whole 0 to MAX isn't alloted at process creation. waste then!

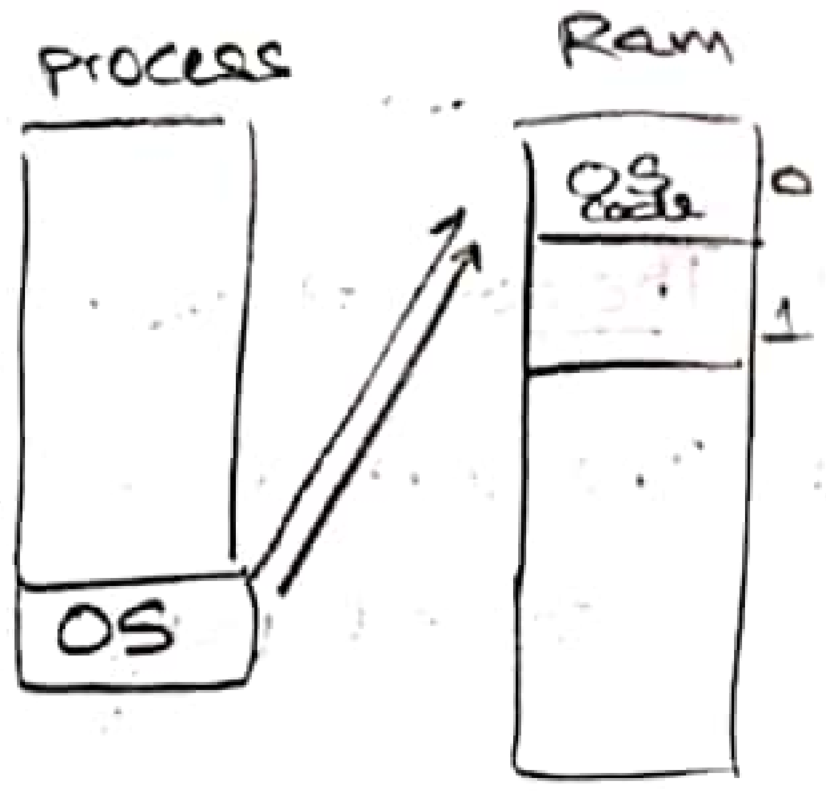
* Memory Allocation:

- * user C-code can allocate more heap memory using malloc().
- * malloc implemented in c library-
  - here we got algos for efficient space management.
- * To grow heap,
  brk / sbrk system call.



- * The user program can also allocate a page sized memory using the mmap() syscall.
  - get "anonymous" page from OS.

→ The address space of OS code in a process:-



- * OS is not a seperate process with its own address space.
- * OS code is part of the address space of all processes.
- * Page table maps OS virtual addresses to real OS code.
  [only 1 real OS code like, for all processes].
  like a headerfile.

* How does OS allot itself memory?

- large allocations; OS takes up a page.
- Small allocations; OS uses various memalloc algorithms.

* can't use libc & malloc in kernel.

who will OS syscall to again....

So, in xv6 kernel code
(which is written in C)
we never have
new ....;
malloc ( )..;
?

hmmm...
wow....

L8: Mechanism of Address Translation:-

* In a simple example of one page translation;

OS tells the MMU, the base (start address)

and the bound (total size of allocation)

- then the MMU does:

  PA = VA + base.

  if (VA ≥ bound) error!

* Hence; OS just says the base, bound once. It is not involved in every translation.

* Hardware role:-

- ISA has priviliged instructions to set translation information. (base bound etc)
- MMU uses this information for every! translation.
- MMU generates fault & traps into OS when access is illegal! VA out of bounds.

  ! Hardware! Interrupt

* role of OS:- (in memory management)

* OS maintains free list of memory. (In a linked list..)

* Allocates frames during process creation (& when requested).

* maintains page table in PCB of a process.

* Set address translation information in hardware! - during context switch ✓

* Handles traps from Hardware. - during Trap? XX.

→ Segmentation:- external fragmentation:- in unallocated RAM.

* say we are using generalized (base, bound) rather than fixed size pages.

* variable sized allocations leads to external fragmentation:

- small holes left out in RAM, between segments.
- no such issue with fixed size segments.

L9: Paging:-

* lets allocate memory in fixed sized chunks. → makes mem management easy.

- Avoids external fragmentation.
- Has internal fragmentation (partially filled pages)

→ Page table:-

* per-process datastructure

* Array stores mapping from

virtual page number to Physical frame number
VPN PFN

fair enough.

Index is virtual page number!
not the starting address of page!

* MMU has access to page table.

* OS updates page table of MMU upon context switch.
but not during traping into kernel.

→ Page table entry (PTE):-

The OS class scene from "The social network". Hehe!

* page table entry is one per virtual page.

* VPN is the index into Page table, for its PTE.
array of PTES.

* Each page table entry contains

→ PFN (physical frame number) & few other bits

Status bits

→ Valid bit: Is the page used by process?

→ Protection bits read/wr. permission.

→ Present bit: is this page in memory? or swap? demand paging!

→ Dirty bit: has this page been modified

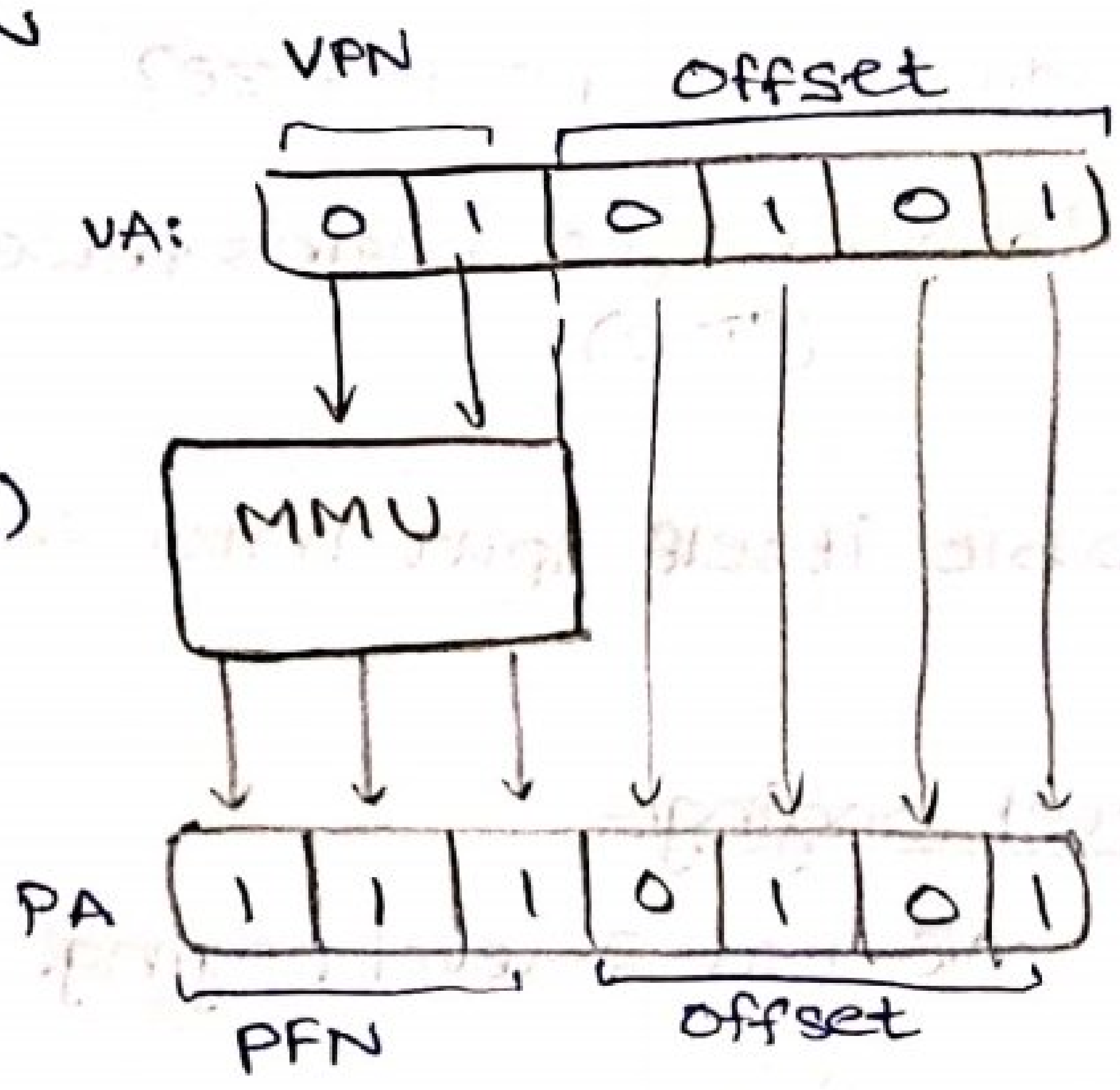
→ Accessed bit: has this page been recently accessed? (even for reading!)

* most significant bits of VA give VPN

* the MMU stores the addrese of the start of page table.

  Need to add (or rather append!) offset.

  hehe..



* MMU needs to translate EVERY address from CPU.

- Paging adds overhead to translation. (could be multi-level page table).

* Hence use a cache for VAPA translation.

→ Translation lookaside Buffer TLB :- inside MMU

* A cache of recent VA-PA mappings accessed. (since Memory access is slow...)

* MMU first looks inside TLB.

- if TLB hit, PA directly accessed.

- if TLB miss, then MMU walks the page table.

* TLB misses are expensive (multiple memory accesses)

  * locality of reference has a high hit rate.

    code usually asks lw/sw in similar locations.

    - for loops etc.

*TLB enteries become invalid on context switch & change of pagetables.

* Page tables typical size:

  4KB → first 20 bits of VA is our VPN.

  12 bits offset

  * say each PTE is 4 bytes; then total page table is

    4 x 2^20 B = 4MB (too large) for single process.

* How to reduce page table size?

- larger pages means fewer PTE.
  (Size)

or
⋮

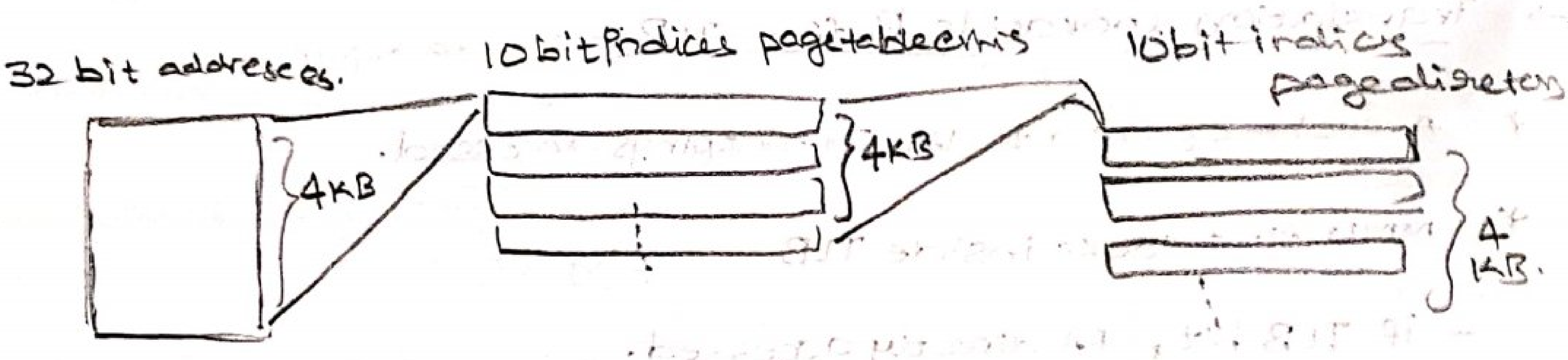
Page table itself split into smaller chunks
(Pages) haha.

→ Multilevel paging:-

XV6 has 2-level paging.

* A page table itself spread over many pages. (All need not be allocated at process creation)

* An "outer" pagetable or page directory tracks PFNs of page table pages.



VA:

10 bits	10 bits	12 bits
page dir index	page table index	offset

∴ 2^{32} values } 4×2^{20} pagetable Bytes } 4×2^{10} Pagedirectory Bytes.

4GB　　4MB　　4KB

all 2^{10} pagetable pages need not be alloted. This saves space on pagetables per process.

* we could need even higher level pagetables.

- 64bit arch, – 7 levels.

* Like this; in case of TLB miss, we'd need heavy cost.

L10- Demand Paging:-

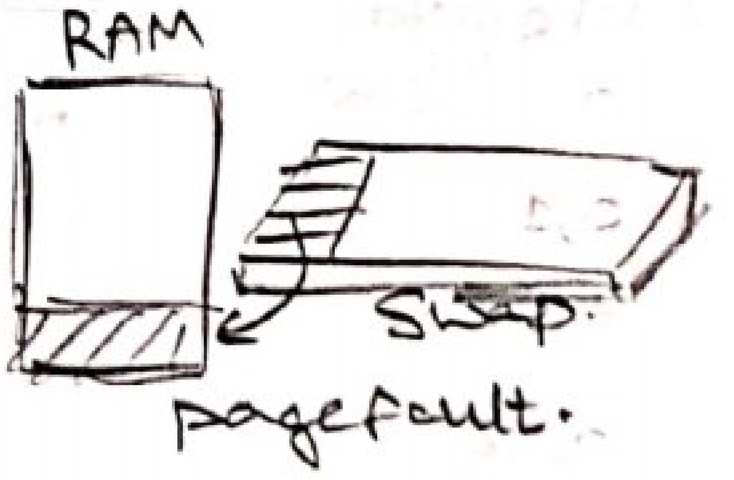
* Not all pages of all active processes are present in RAM/memory.

- OS uses a part of disk (swaparea) to store pages not in use.

→ Page fault:-

* present bit indicates if the page is in meomory or not:

- if page not present; MMU raises a trap to OS - page fault. not error!.
- in Kernel mode (after traping); OS issues read to disk to bring back the page.
- OS switches to another process. (Disk has a small CPU which works independent of our computer CPU).
- After disk read completes; disk raises an interrupt & OS updates the page table of our old process & marks it ready.
- When old process is scheduled again, OS restarts the instruction that caused page fault. not the next instruct^n!



Summary:- What happens on memory access by MMU:-

1) CPU issues load to VA.
   - checks CPU cache first Ooo....
   - goes to main memory in case of cache miss.

2) MMU looks up TLB for VA =
   - TLB hit : obtains PA, access memory.
   - TLB miss: walks page table & obtains PTE.
     - If present bit set, access memory
     - If not present, but valid, raise pagefault. OS handles page fault; sets the present bit; restarts instruct^n.
     - If invalid page, trap to OS for error.

When moving a page to swap; clear TLB entry!!.

* more complications in page fault:-

- what if OS finds no space to swap-in the faulting page.
- OS will readily swapout pages to keep a list of free pages.

→ Page replacement policy:-

1. Optimal: replace page not needed for the longest time in future. (not practical... we dunno future).

2. FIFO: replace page, bought in the earliest! (but, could be a popular page)

3. LRU:/LFU: replace the page that was least recently (or frequently) used.
least recently used

Eg: 3 frames & 4 pages...

* first few access are cold miss.
compulsary:

Optimal:-

Access	Hit/Miss	Remove	Cache (after page fault)
0	miss		0
1	miss	cold misses.	0,1
2	miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	miss	2	0,1,3
0	Hit		
3	hit		
1	hit		
2	miss	3	0,1,2
1	hit		

net $\underbrace{3}_{\text{cold}}+\underline{2}$ misses

FIFO: usually worse than optimal.

Access	hit/miss	Remove	Cache
0	miss		
1	miss		
2	miss		0,1,2
0	hit		
1	hit		
3	miss	0	1,2,3
0	miss	1	2,3,0
3	hit		
1	miss	2	3,0,1
2	miss	3	0,1,2
1	hit.		

net = 3+ 4 misses
Bad.

Belady's anamaly:-

performance gets worse, when memory size increases.

LRU policy:-

Access	hit/miss	evict	cache.
0	miss		0
1	miss		0,1
2	miss		0,1,2
0	hit		1,2,0
1	hit		2,0,1
3	miss	2	0,1,3
0	hit		1,3,0
3	hit		1,0,3
1	hit		0,3,1
2	miss	0	3,1,2
1	hit		3,2,1

keep 0 at last; since most recent.

∴ hit = 3 + 2 Same as optimal.

* works well due to locality of reference.

- recently used pages are more popular.
- Hence evict least recently used ones.

* How is LRU implemented:-

- OS is not involved in each & every memory access. How to implement LRU?
- Hardware help & some approximation.
- MMU sets a bit in PTE. "accessed bit" when page is accessed.
- OS periodically looks at this bit; to estimate active & inactive pages. classify into hot/cold pages.
- To replace; OS tries to find a page that has low access bit (0).
    - may also look for page with dirty bit unset. (to avoid swapping out to disk). not sure here...

*So OS. only swaps out pages of current process ?

* will TLB be cleared, if page swapped out?

→ Linked list with headers

L11: memory allocation Algorithms   Data Structure + Algorithm.

↳ first, best, worst.

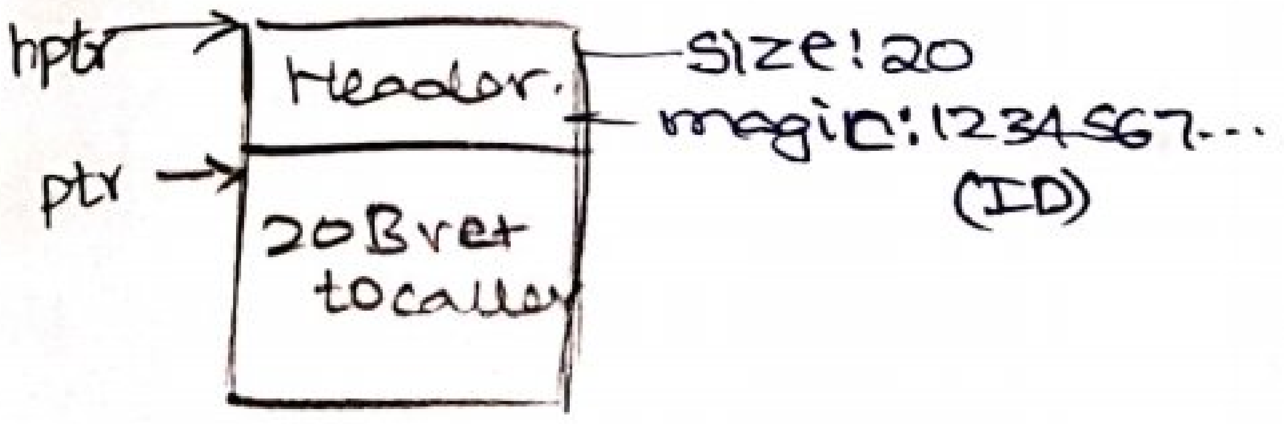
* Fixed size allocation is straightforward. lets see for variable sized allocations.

* This problem must be solved in C-library in malloc().

& also in Kernel. Kernel must allocate memory of its internal data structures.

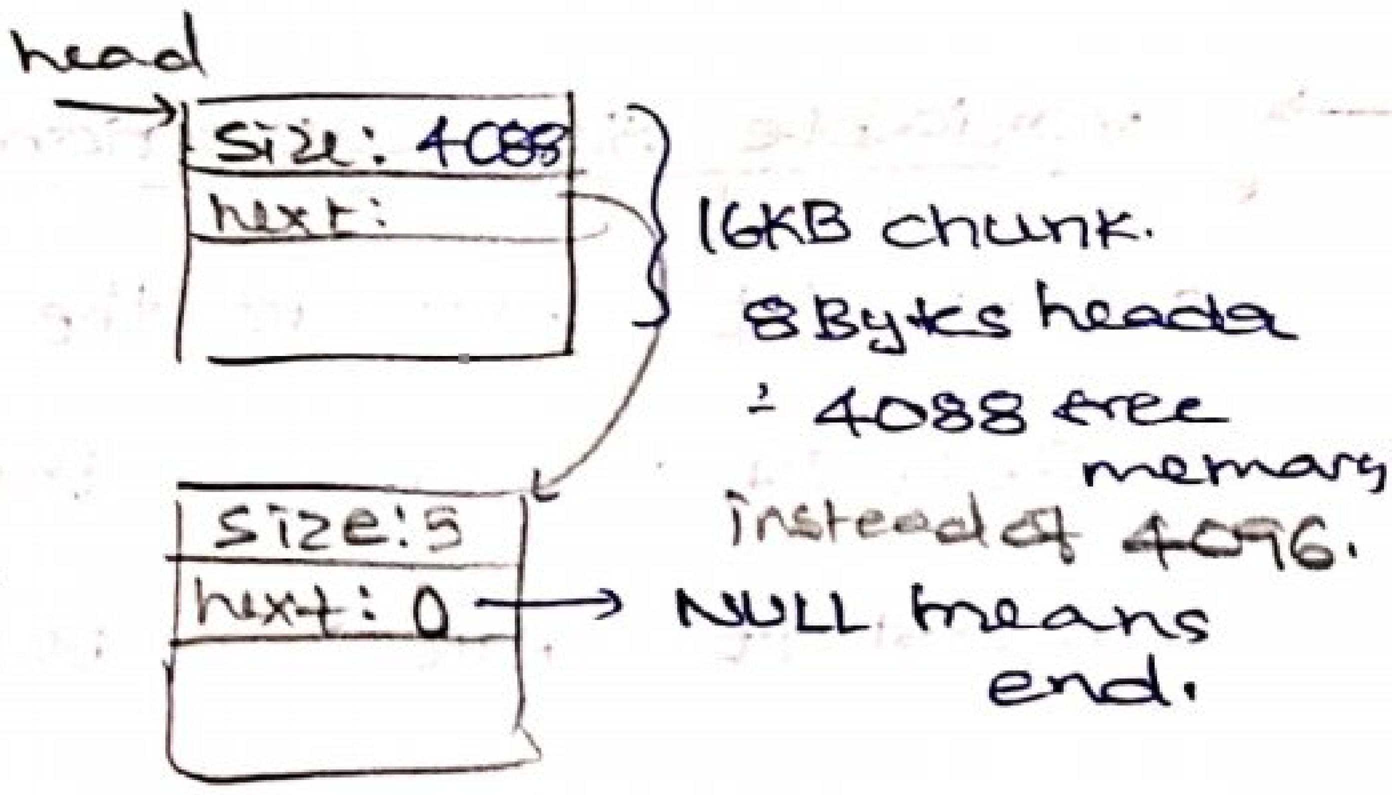
→ **Headers:**

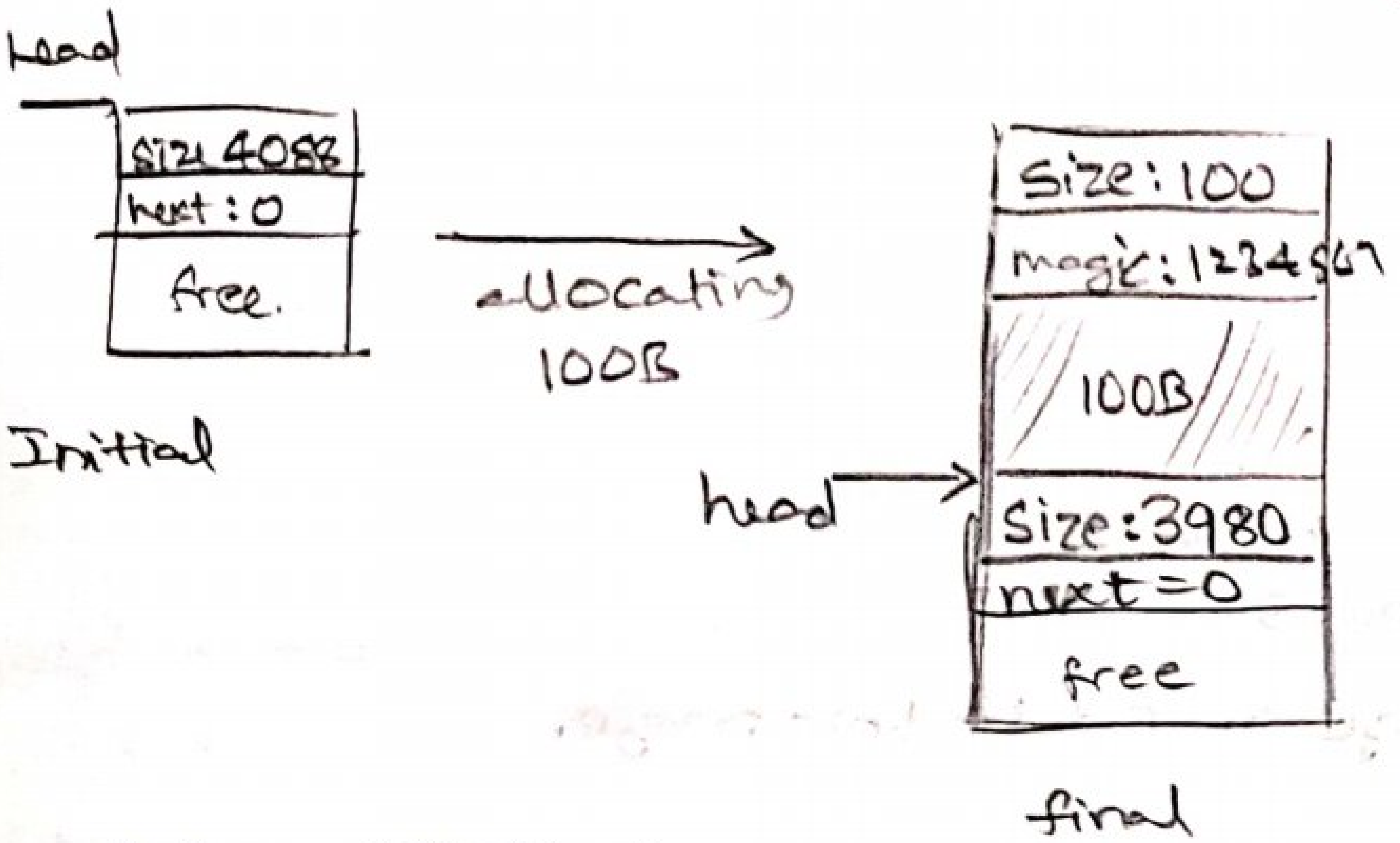
- consider simple case of malloc()
  - every allocated chunk has a header with info like size of chunk.



* Size is needed; since when free() called; we'll know how much to free.

* Free space is managed as a freelist.
  - pointer to next chunk; mentioned in current chunk.
  - Allocations happen from head. C-Library tracks the head.





see how
4088 free → 100 occupied + 3980 free

header loss = 8 Bytes

Nice!

* external fragmentation:-

* Say we have 3 100 blocks & 1 3764 block. Now freed middle 100 Block. Total we got 3864 B free space

But fragmented!

* So, we can't allocate a 3800B request!

Since continuous 3800B unavailable.

* Spliting & Coalesce:-

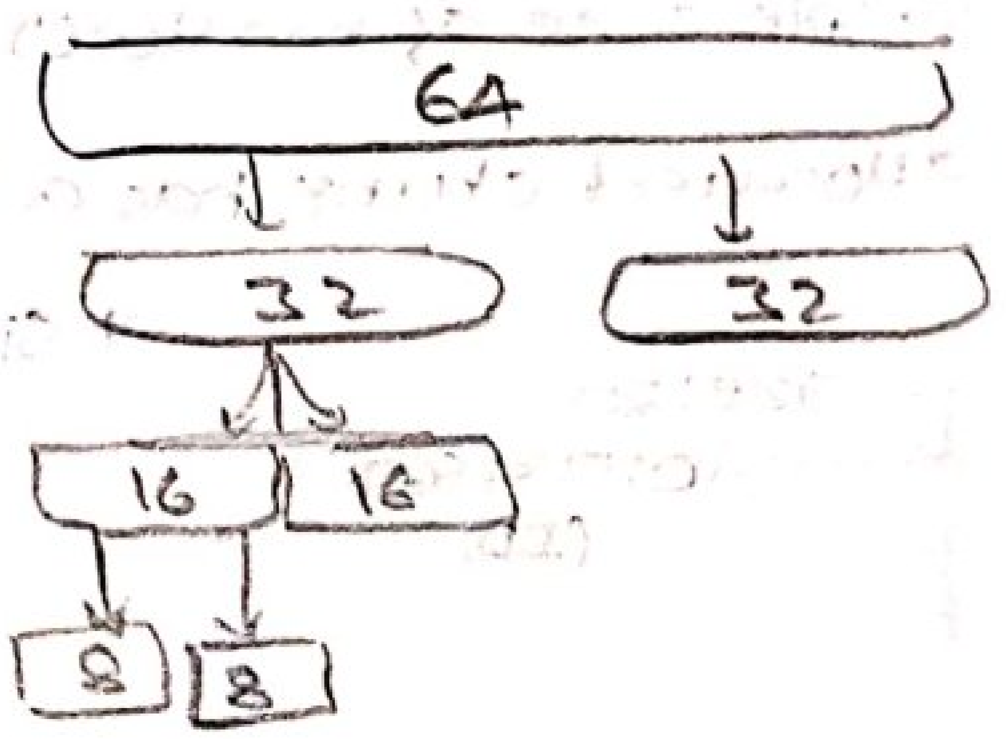
- * A smart algorithm must coalesce adjacent chunks.
- * Must split while alloting requests & coalesce while freeing requests.

coalescing also reduces headers overhead.

→ Buddy allocation for easy coalscing:-

- Allocate memory in powers of 2.

Eg: for 7KB request,
allocate 8KB.



Cause; if
this chunk & its buddy
are free; Coalesce
into bigger
chunk.

→ variable size allocation strategice:-

* the policies!

- First fit: Allocate the first chunk thats sufficent.
- Best fit: Allocate the chunk, closest in size.
- Worst fit: Allocate the largest chunk in size.

→ fixed size allocations:-

*A Bit easier.

- has free list of pages
- pointer to next page stored in this page.

} need a header na?

I dunno...
maybe not needed.

* for smaller allocations (like PCBs), Kernel uses a slab allocator.

- object caches for each type (size) of object.
- within a cache, only fixed size allocations.
- each cache; made up of one or more "slabs".

* we could use fixed size allocations in C (instead of malloc...).
But meh!

L27: Paging in xv6

- 32 bit OS → 4GB virtual address space per-process.

  page size → 4KB

  no demand paging.

- each PTE has : Pagetable is indexed using 20bit index.
  - 20 bit PFN (physical frame no.)
  - some flages:-

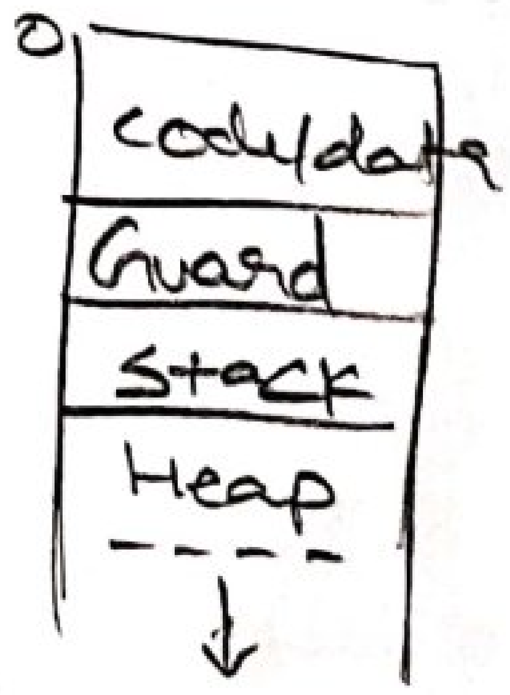
    PTE_P : present. if not set → pagefault.
    PTE_W : writable. if unset → only read permited
    PTE_U : user : if unset → only Kernel can access page.

- 2^{20} PTE can't be stored simultaneously.

  Two-level page table.
  - 2^{10} inner page-table pages.
  - Outer page directory is 4KB in size.
  - physical address of outer page directory in CPU's cr3 register.
    used by MMU

→ Process virtual address space:-

- from 0:
  - code/data
  - Fixed size stack (with guardpage)
    ! xv6 only.
  - expandable heap



- Kernel code/data:

  from 2GB onwards

  KERNBASE.
  - kernel code/data
  - kstack? * freepages maintaind by kernel
  - Some space reserved for I/O devices.

- Pagetable
  - user PTEs } maps low virtual addresses
  - kernel PTEs } maps high VAs to real OS code.
    (identical in all processes)

→ OS page table mappings:-

* maps

KERNBASE

(2GB, 2GB+PHYSTOP) to (0, PHYSTOP)

VA

PA ↳ in physical memory.

- mapping identical in all processes for kernel code.

* During trap; we'll use the same pagetable. for OS code...

* [0, PHYSTOP) has code for

- Kernel code/data
- I/O devices
- mostly free pages. in physical Addresses.

∴ in kernel VAs;

Phy. frame P has virtual address P+2GB.

∴ } same frame has 2 virtual addresses. → then 2 PTEs (one user, one kernel) will point to same PFN.

can be mapped to user pages.

* every RAM byte has 2 bytes in process.

∴ XV6 process can't use more than 2GB.

---

* freelist: maintaind by OS.

- just a linked list. ←
- kernel maintains a head pointer

```
struct run{
 struct run*next;
}
```

lol.

* assigns pages to user memory & page tables of user procs.

```
struct {
   struct spinlock lock;
   int use_lock;
   struct run* freelist;
} kmem;
```

→ Alloc & free:-

- who needs a new page: Kalloc()
  - returns first free page on freelist.
- who needs to free a page: Kfree()
  - Add freepage to head of freelist.

Simple!

L-28: memory management of user process:-

* New virtual address space for a process during

- init creation
- fork()
- exec()

Hmmm....

* existing VA space; modified in Sbrk systemcall.

→ Building pagetable for a process:-

- start with one page for directory.
- Allocate inner pages on-demand.

* begins with setupkvm(), (outer directory allocated)

| Add kernel mappings

mappages() on kmap[ ].

after kernel mappings; user page mappings added.

* page table enteries added using

mappages (pgdir, va, uint size, pa, int perm)

permisions.

Howmany Bytes.

- for each page,

walks page table; gets pointer to PTE using walkpgdir(....)

& fills it with pa, permissions.

mappages(pgdir, va, size, pa, perm)

walkpgdir (pgdir, va, alloc)

searches for Pgtab in Pgdir & returns PTE in Pgtab

if alloc = 1; allocate PTE corresponding to va if not present.

PDX, PTX are macros.

pagedirectory index

page tabel index.

→ **Fork**: copy memory image] implemented in OS. not the user process. Hence, gotta work with PAs. not VAs

* copyuvm() called by parent to copy.

- create a new pgtable for child. } using setupkvm():
- walk through parent VM page by page;
  copy to child; add PTE in child.

* for each page in parent:

- fetch PTE, get physical address, permissions.
  ↳ memmove()
- kalloc() new page for child.
  memmove() into new page.
- Add PTE from va to pa in child
  using mappage().

needs physical addresses.
not virtual
(-_-)
in both arguments!

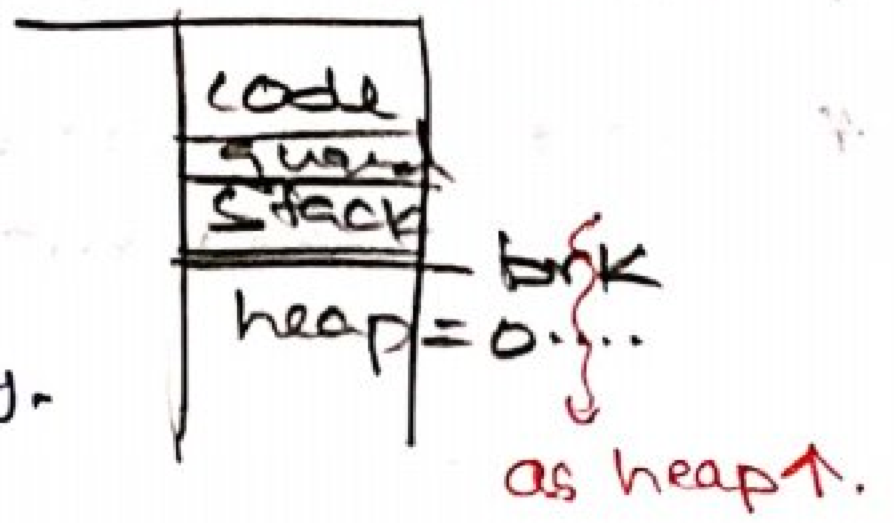
*virtual address means nothing in OS code.

* real OS do copy-on write:

initially child also points to same PA as parent.
duplication occurs if one of them modifies the page in their code.

→ Growing memory image : sbrk().

- * initially heap is empty.
  program "break" is at endofstack.
  - sbrk() is invoked by malloc() internally.



- * to grow memory; allocuvm() allocates new page, adds mappings into Pgdir.
- * whenever page table updated, must update Cr3 reg & TLB
  switchuvm()
  also done during context switch.

growproc(int n)

```
├→ allocuvm (pgdir, sz, sz+n)
├→ deallocuvm if n<0
└ switchuvm(currproc).} refreshing
```

Allocuvm():

- * walk through new VAS to be added
  - * Alloc new page kalloc()
    add to page table mappages().
  - * similarly deallocuvm() to kfree().

Allocuvm()

```
│ for (a < newsz; a+=pg siz)
├→ mem=kalloc()
├→ mappages (a, mem).
     okay.
```

→ exec system call:-

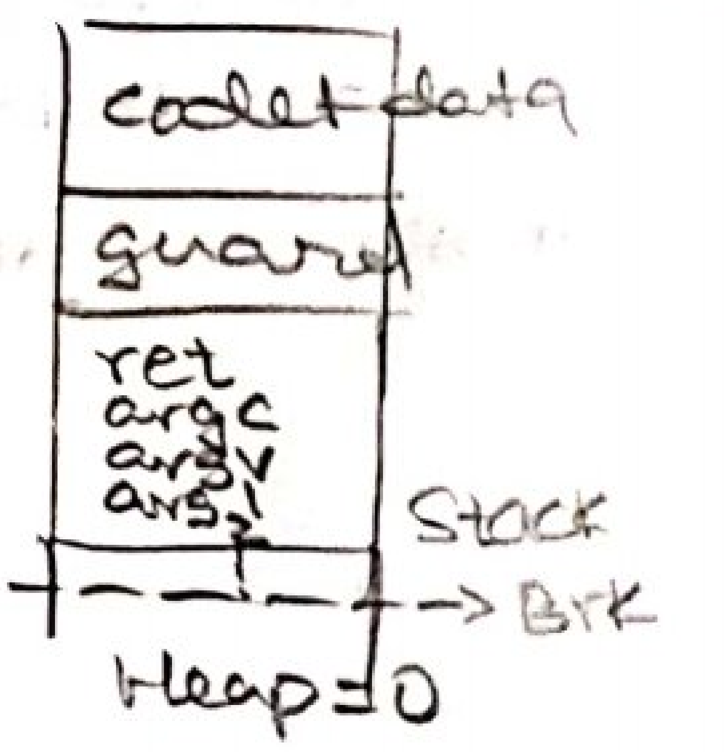
- read ELF binary file from disk to memory.
  - ELF: executable & linkable format.
- start with new pg dir.
  - Add mappings to new executable pages & grow virtual addresses.

exec
- setupkvm
- readi (ELF)
- allocuvm

- After executable is copied (only code + data na!)
  - Allocate 2 pages - Guard + Stack.
    - Guard: permissions cleared. Accessing this will trap!.
  - push exec() arguments onto user stack for main function of new program. (command line args).
  - Stack now has return address, args, argv array (pointers) & the arguments themselves. !!

exec
- allocuvm (2PGSIZE)
- pushing arguments onto userstack.

new memIng



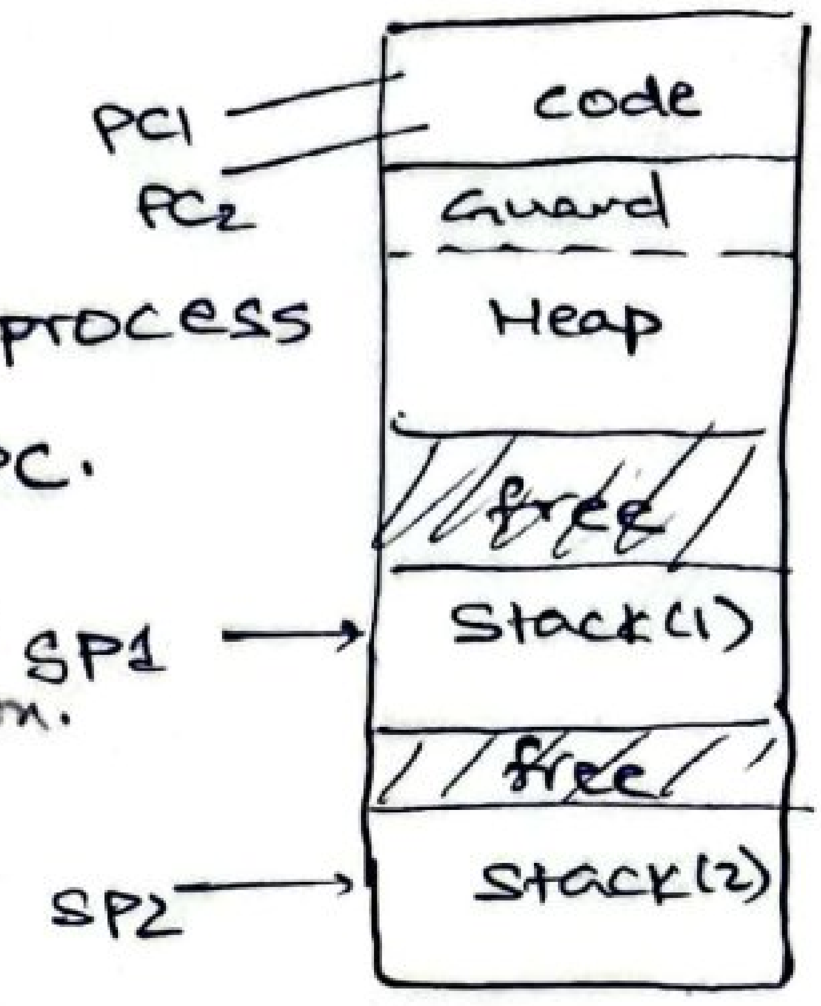
- if no errors so far; switch to new pgdir.
  - if any error; don't switch. return exec() with error.
- Set EIP in trap frame to entry of new program.
  - returning from trap; new code will run.

exec
- curproc→pgdir = pgdir
- curproc → tf → eip = elf.entry
- switchuvm (curproc) } update CR3 & TLB

L12 : Threads and concurrency:-

* Thread is like another copy of a process, that executes independently.

- Threads share the same address space (code, heap...)
- Each thread has seperate stack, for independent function calls.



* For communication among threads of same process use Global variables rather some IPC.

interprocess communication.

* concurrency vs parallelism:-

concurrency → multiple threads/processes at same time; even on single core; by interleaving their execution.

parallelism → multiple processes/threads in parallel over multiple cores.

xv6 has no threads.
But supports multi core

* why threads?

- single process can effictively use multiple cores parally.
- Even if no paralelism, concurrency of threads ensure one thread runs, when other thread is blocked... imp!

* Scheduling threads:-

- The context of a thread (PC, register) is in Thread Control Block. TCB.
  - each PCB has a list of TCBs.
- The threads those are scheduled independently by Kernel are Kernel-threads.

Linux pthreads.h is a Kernel thread.

pthreads_create( )
pthreads_join( )

* Some libraries provide user threads. wooo!

- user sees many threads
- Library multiplexes large no. of user threads over small no. of K.thread
- low overhead of switching in user threads.

→ Race Condition:- concurrent execution leads to different results.

* We really can't predict the execution order of multiple threads.
at assembly level.

Say

```
counter=0

thread 1 :                      thread 2 :

for(i=0; i<1000; i++)           for(i=0, i<1000, i++)
    counter++;                      counter++;
```

By end of both threads; we expect the counter to be 2000.
not anymore...
But we get less sometimes

counter++; in assembly:-

```
0)  mov   0x8abc, $1
4)  add   $1, 0x01
8)  mov   $1, 0x8abc
```

Thread 1: executes 0,4 & then interrupt!

Thread2: executes 0,4,8

Thread1: executes 8.

Now, we finally got (+1) instead of (+2).

lol....

* Critical Section : portion of code, that can lead to race condition.

- What shall we do? Mutual exclusion! only one thread executes critical section at once.
- What we need? Atomic Instructions from ISA & Atomicity of critical section

Achieve by using Locks. 😄

L13: Locks:-

pthreads library in linux provides such lock.

* We need to update a shared variable...

Use a lock: only a thread which holds the lock can change it.

- Goals of building a lock
  - Mutual exclusiveness: lol...
  - Fairness
  - Low overhead: acquring, releasing, waiting for a lock; shouldn't consume many resources.

- locks needed for user programs & kernel programs.
  (pthreads)

Implementation of locks needs support from microarchitecture & OS.
(atomic instructns))

→ Disabling interrupts:-

really? mightbe... to send signals...

* Haha; such an implementation isn't nice.

good! - Disabling Interrupts is a priviledged instruction & cannot give this power to user code.

- In multicores; another thread on another CPU can access critical section.
  (unless u disable on all processors...) which is a big waste.

* Can use this in single processor systems inside OS.

→ Lock Implementation:-

* we'll use a flag to track whether lock is taken/available. But simple instructions wont do. Race condition moves to lock acquisition code.

Use atomic Instructions:-

1) test-and-set : update a variable & return old value. All in one instruction.

& so

```
• struct lock_t { int flag=0;}
• void lock ( lock_t* lock) {
      while (TestAndSet(&lock->flag, 1)==1){
          ;
      } Spin wait. do nothing till condition true.
  }
```

Spin lock

Spinning until lock is acquired...

```
2) compareAndSwap (int* ptr, int expected, int new) {

        int actual = *ptr,
         if (actual = expected)
               *ptr = new;
        return actual;
    }
```

Again, can implement a spinlock.

→ Alternative to spinning:- sleeping mulex.

* A contending thread yeilds the CPU.

```
while ( testAndSet(&lock → flag, 1) ==1)
      yield(); // give up CPU.
```

nice!

* Most userspace locks → sleeping mutex kind. Saves lots of resources...

inside, OS ——→ spinlocks, only. why?

Who will OS yeild to? itself na? Doubt!

* When OS acquires a spinlock!

nice.

1. It must disable interrupts, when the lock is held. Cuz interrupt handler could also request for the same lock & spin forever.

no! maybe scheduler itself needs that lock... & again it itself yeilds & ! recursive loop...

2. Must not perform any blocking operation. "never go to sleep with a lock". You are O.S! who'll wake you up?

---

* When to use locks:

*1. A lock should be used before acquiring any shared data structures. "thread-safe data structures"

2. All Shared Kernel DS. should be accessed after locking.

3. Coarse-grained locks vs fine-grained locks:-

one big lock for all shared data

Seperate locks...
- allows more parallelism.
- Harder to manage.

L14: Condition Variables:- Pthreads provide CV for user prog.

* we done mutual-exclusion. lets see waiting-signalling.

different from yielding... → ready to run.

→ Condition Variables:-

* A CV is a queue, a thread/process can put itself into; waiting for a condition.

- Another thread signals CV to wake up a waiting thread.

Signal → wakeup one thread.

signal broadcast → wakeup all.

- no flag inside Condition variable just wait(), signal() methods.
- wait() is different from yield() & yield() makes status runnable; where wait() makes status sleeping.

Eg:

```
Shared value  int done=0
Lock  mutex_t m = init_mutex;
CV:   condvar_t c = init_condvar;

      void  thr_exit(){
            lock(&m);
            done = 1;
            Signal(&c);
            unlock(&m);
      }
      void* child {
            thr_exit();
            return NULL;
      }
      Void thr_join(){
            Lock(&m);
            while(done==0)
                  wait(&C, &m);
            unlock(&m);
      }
      main {
            pthread_t p;
            pthread_create(&p, child);
            thr_join();
            return 0;
      }
```

use a 'while' loop; rather than if-condition; to account for wrong waking of CV.

use lock, while accesing a shared variable. Otherwise here; race condition could send parent to permanent sleep.

IMP "Lock must be held while using wait, signal on conditional variables"

and so;

The wait(.,.) releases the lock; before putting thread to sleep. Never sleep with lock

Similarly; when thread wakes up; it would already be holding the lock.

1. CV is a queue
2. Update shared vars inside a lock. (Read)
3. Use lock; before using wait, signal on conditional variables...

→ Producer - Consumer Problem:-

* A common pattern in multi-threaded programs:

Setup: one or more producer threads
one or more consumer threads
a fixed buffer for everyone.

* Using 2 CVs:-

```
int count =0;
cond_t  empty, fill;
mutex_t  mutex;

void* producer (void*args) {
        int i;
        for (i = 0; i < loops; i++) {
            lock(&mutex);
            while (count = MAX)
                wait (&empty, &mutex);
            put(i);
            signal (&full);
            unlock(& mutex);
        }
}
```

→ no need lock here.
locks already stored in queue of CV. Those need to be returned to processes.

```
void* consumer (void){
        Int i;
        for (i=0; i<loops; i++){
            lock (&mutex);
            while (count ==0)
                wait(&full,&mutex);
            get(i);
            Signal (&empty);
            unlock(&mutex);
        }
}
```

Again we note:-

1) Lock must be held, during wait, signal on conditional variables.
2) Pass the lock to wait(.,.) which releases the lock.
3) Never sleep with a lock.

L15. Semaphores:-

- * Semaphore is also a synchronizat^n primitive like CVs.
  - Semaphore has an underlying counter.
  - up/post increments the counter.
  - down/wait decrements the counter & blocks the calling thread if the resulting value is negative.

- * Semaphore with value 1, acts like a lock.

Binary semaphore = mutex.

```
sem_t m;
sem_init (&m, X);
wait (&m);
  //critical section
post(&m)
```

X = 1 for lock.

* Semaphores for ordering;

```
void* child() {
  sempost(&s);
}
main {
thread (&P, child);
sem.wait (&s);
}
```

See how we don't use lock beforehand...
Since if() condition is atomically implemented inside sema

Here no lock is passed in wait(); unlike C.V. wait.

→ Producer Consumer problem again:-

- * 2 semaphores for signalling:
  - one to track empty slots
  - one to track full slots

* 1 Semaphore as mutex for shared buffer.

```
sem_t empty; sem_init(&empty, 0, MAX);
sem_t full;  sem_init(&full,0,0);
sem_t mutex; sem_init(&mutex,0,1); } a lock...

void* producer() {
    int i;
    for (i=0; i<loops, i++){
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}

void * consumer() {
    int i;
    for(i=0; i<loops,i ++){
        sem_wait(&full);
        sem_wait(&mutex);
        get(i);
        sem_post(&mutex);
        sem_post (& empty);
    }
}
```

loaded

some other arg...

] mind this order!

never sleep with a lock.

"Acquire lock after signalling".

A different conclusion; when CV are used...

since cv_wait are able to free locks; where as sem_wait are not.

no if() ---- here; since inbuilt inside semaphore.

L16:

(Non-deterministic).

Bugs in concurrent programming:-

Deadlock bugs:-

cannot go further

classic: T_1 has L_1, needs L_2

T_2 has L_2, needs L_1

Condition for deadlock:

See next page.

> deadlock dependency grph.

Non deadlock bugs:-

wrong results obtained

- **Atomicity Bugs:**
  - fix is: use locks for mutual exclu^n.
- **Order violation Bugs:**
  - use CV to impose ordering among various processes. Signal in 1st process wait in 2nd process & a bool variable.

when our assump^n about atomicity of instructions is violated.

when our assumption abt order of exec. of 2 or more threads is wrong. can't just assume such stuff.

→ conditions for deadlock to occur

1. Mutual exclusion: a thread claims exclusive control over a resource.

2. Hold-and-wait: thread holds a resource & is waiting for another

3. No preemption: Thread cannot be made to give up its resources.

4. Circular wait: There is cycle in resource dependancy graph.

* All four above must hold for deadlock.

* preventing circular wait:-

- Acquire locks in a fixed order, everywhere.
- Impose total ordering.

Eg: Acquire the least address lock...

```
if (m1<m2){
        lock(&m1);
}       lock(&m2);
else {
     lock(&m2);
     lock(&m1);
}
```

reboot system or
kill deadlocked proce

Doubt

will killing work?
lock is still held...

* preventing hold-and-wait:

- Have a big master lock; instead of smallones.
  & then acquire smaller ones.
- may reduce concurrent execution.

* OS can schedule s.t. deadlocks won't occur... impractical!

Banker's Algorithm   OS won't know na!.

L29: Locking in XV6 :-

- * XV6 has no threads. But supports multiple cores.

  So; we need locking in OS code.

- * Use spinlocks to protect critical sections.

→ Spinlocks in XV6 :-

- * xchg (&lk->locked, 1) x86 atomic instruction

  similar to testAndSet.

- * must disable interrupts; before spinning for lock.
- * Interrupts stay disabled; until ALL locks are released.

  Maintain a counter for no. of locks held.

  mycpu() → ncli;

```
acquire () {
    pushcli(); → disable interrupts.
    while (xchg (&lk->locked, 1) =! 0)
      ;
}
```

```
* pushcli()
  {
    disable interrupts
    if mycpu()->ncli==0;
    mycpu()->ncli++;
  }

* popcli()
  {
    decrement ncli;
    if ncli == 0 : enable interrupts
                    sti();
  }
```

→ ptable.lock :-

- * Any access to ptable must be done with ptable.lock held.

  But in code, sometimes...

- * Normally a process :-

  acquire lock → modify ptable → release lock

  But during context switch :- ptable is changing throughout.

  P$_1$ holds lock → goes to scheduler → P$_2$ releases lock.

IMP

- * Every function that calls sched(), does so, while holding the lock.

  yield()
  sleep()
  exit()

  yield(), sleep(), forkret() — for a newly created process.

  Every function switch(;·) returns to, releases the ptable lock immediately.

* Subtle points about Scheduler();

  - scheduler runs a loop with lock held.
  - Periodically; when a loop through all the processes is over; ptable.lock is released & interrupts are enabled.
    - what if no process is runnable & interrupts are all disabled....

    To avoid this; we enable interrupts periodically.

* Interrupts during lockheld time are queued up.(not discarded).

```
void
scheduler(void)
{
  for(;;){//∞ LOOP
    sti(); //turn on interrupts
    acquire(&ptable.lock);
    for(over all processes){

      swtch(0,0);

    }
    release(&ptable.lock);
  }
}
```

Kernel functions... not userspace...

L-30: Sleep and Wakeup in XV6 :-

How interrupts work... Similar to condition variables in user codes.

→ **Basis** :-

- * process P1 blocks for an event – disk read.

  Invokes sleep() function.

- * while P2 is running; say disk interrupt occurs and wakeup() is run.

  How to know which proc? use channel. (Stored in struct proc 'chan')

  * common value known to both Sleep process & wakeup process
  : locat^n of struct proc
  : address of disk read
  etc...

- * Also; have to use same lock for sleeping & wakeup...
  - to bypass missed wakeup problem. ← wakeup sent, even before proc. went to sleep... lol.

So;

```
P1:                    P2:
lock l;                lock l;
if (!done)               wakeup(c);
   sleep(c, l)         unlock l;
   → released l
unlock l;
```

① called with lock held. return with lock held

```
void
sleep(void*chan, spinlock *lk){
    p= curproc();

    if ( lk!= &ptable.lock){
       acquire(&ptable.lock);
       release (&lk); for wakeup fun.
    }

    p→chan= chan; //stored in struct proc
    ---
    sche();

    p→chan=0;
    if ( lk!= &ptable.lock){
       release(&ptable.lock);
       acquire(lk);
    }
}
```

release the lk given; only if its NOT ptable.lock.

(called when lk is ptable.lock)

② wakes up all processes with given chan.

```
void
wakeup1 (void*chan)
{
    struct proc* p;
    for( p in proc[] & p→state=SLEEP&
       p→chan== chan)
        P→state== RUNNABLE;
}
```

③ if lk is not ptable lock.

```
void
wakeup(void*chan){
    acquire(ptable.lock);
    wakeup1(chan);
    release (ptable.lock);
}
```

A wakeup cannot run in b/w sleep; as sleep holds lk or ptable.lock.

* Examples are pipes:-

```
pipewrite ( pipe*p; char*addr;
                        int n);

piperead (pipe*p; char*addr;
                        int n);
```

```
struct pipe{
    struct spinlock lock;
    char data[PIPESIZE];
    uint nread;
    uint nwrite;
    int readopen;
    int writeopen;
}
```

- one process writes into pipe; another reads from the pipe.
- reader sleeps, if the nread == nwrite; & writer wakes it.
- writer sleeps if nwrite = nread + PIPESIZE; & reader wakes it.
- Channel is addresses of member variables. fine.

* Example is wait and exit:- Here the lock held by both is ptable.lock. Yay! lets see!

* Parent calls wait when child is running

```
wait()  | //wait for children
        | sleep(curproc, &ptable.lock);
```

In child; exit() has lock & wakes up parent

```
exit()  | // parent could be sleeping
        | wakeup1 (curproc->parent)
```

directly this called ☺

chan = addr. of struct proc of parent process.

* child proc. by itself can't cleanup its complete memory... & its struct proc.
cr3, kstack are in use always

So; wait() by parent needs to do this.

lecture 17: Communication with IO devices:-

* IO devices connect to CPU and memory via bus.

Block devices:-

* Store a set of numbered blocks.

Eg: Disks

character devices:

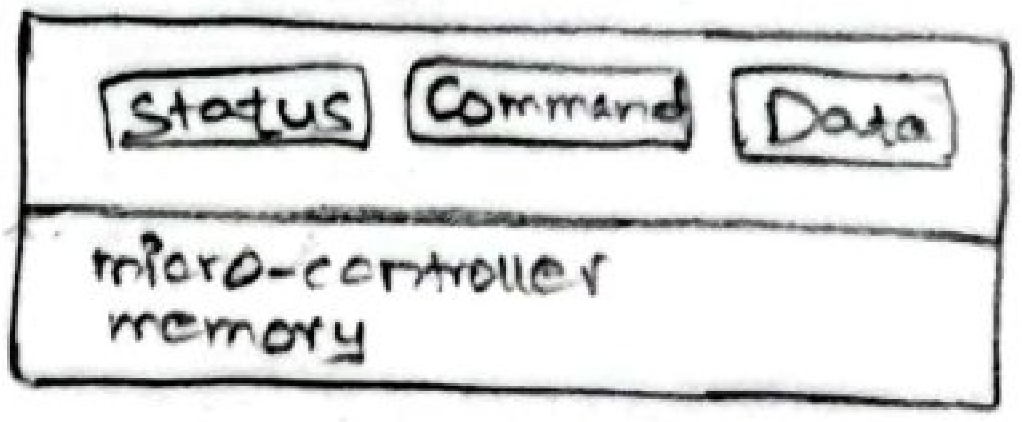
* produce/consume stream of bytes

Eg: keyboard

* Expose an interface of registers : status, command, data

* How OS reads/writes these device regs:-

1) Explicit IO instructions
   - in, out are two such, on x86.
   - priviledged instructions for OS only.



2) Memory mapped IO:-
   - Device regs. appear like memory locations.
   - Memory hardware routes links these special memory addresses to device regs.

→ Execution of IO requests:-

```
while (status == BUSY)
    ;
write data to data reg
write command to command reg
while (Status == BUSY)
    ;
```

1) Polling status to see if device's ready:-
   - → wastes CPU cycles
   - → use interrupts. Send process to sleep & device at completion raises interrupt.
   - → IRQ number identifies which interrupt handler function to call to…
   - → wakes up blocked Process & starts next pending IO request.
   - → If device is fast, polling better than interrupts as we avoid kernel mode transition overheads. Monitor?

2) programmed I/O: explicit copying of data

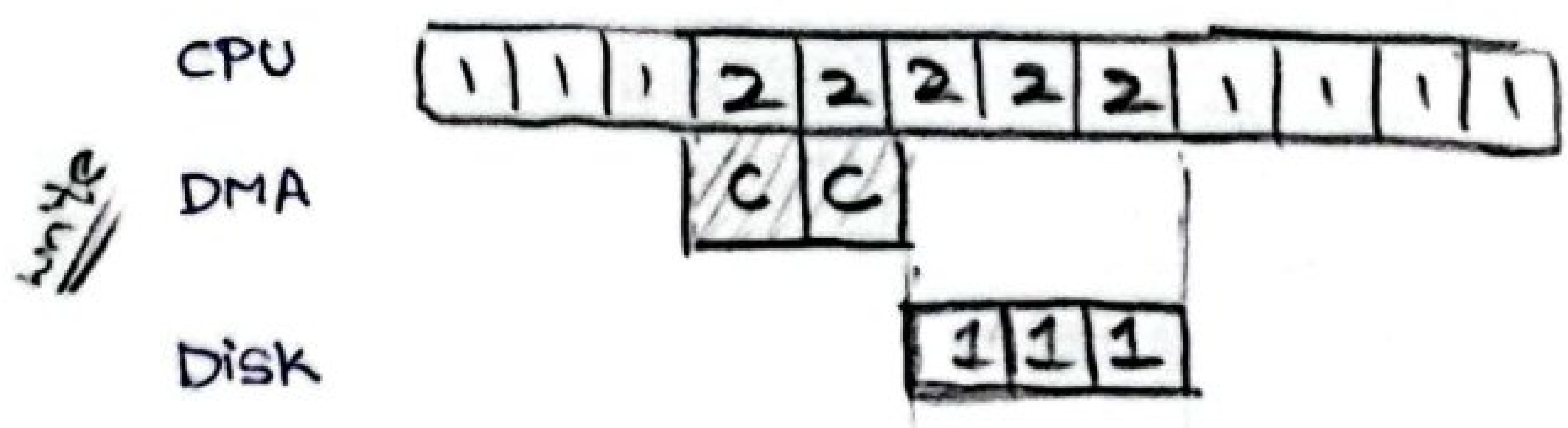
Direct memory Access (DMA):

-> CPU wastes time in copying data

-> Instead, a special hardware DMA engine copies from main memory to device and back.
CPU provides memory location to DMA.

read → DMA raises interrupt after copying to mem.

write → disk starts writing after DMA copies to register.

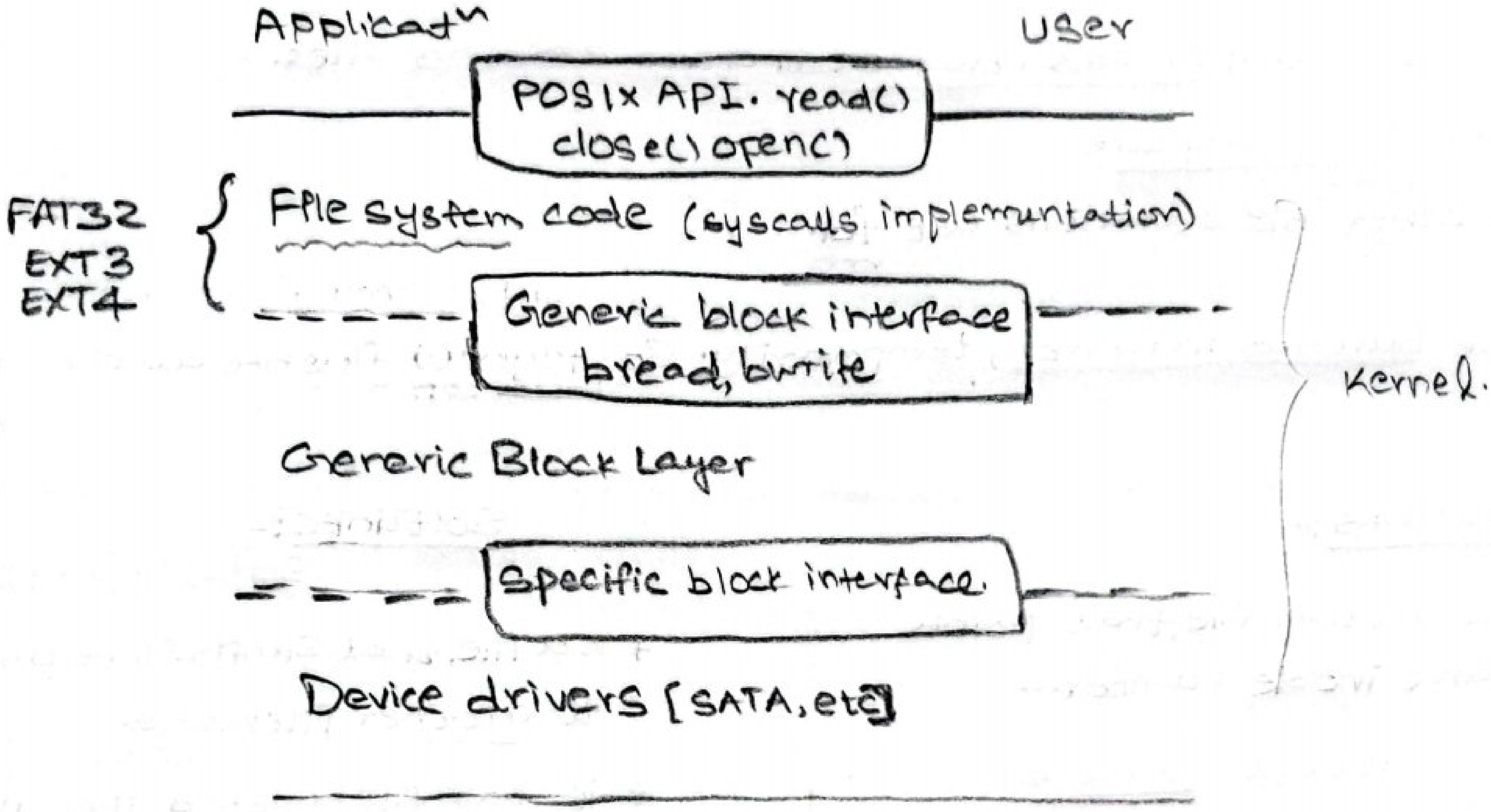


→ Device drivers:- Software.

* part of OS code, that talks to device, handles its interrupts etc.

* Most OS code abstracts the device details.

Eg: file system code is written on top of a generic block interface.



L18: Files and Directories:

inode → index node.

- File: stored persistently.
  identified with filename (human read)
  & inode number (OS-level)
  for each & every file? wow!

Directory: A kind of file, whose contents are filename-inode mappings which it contains.

- open() system call creates new files & opens existing files.
  Returns file descriptor.
  All other file operations use fd.

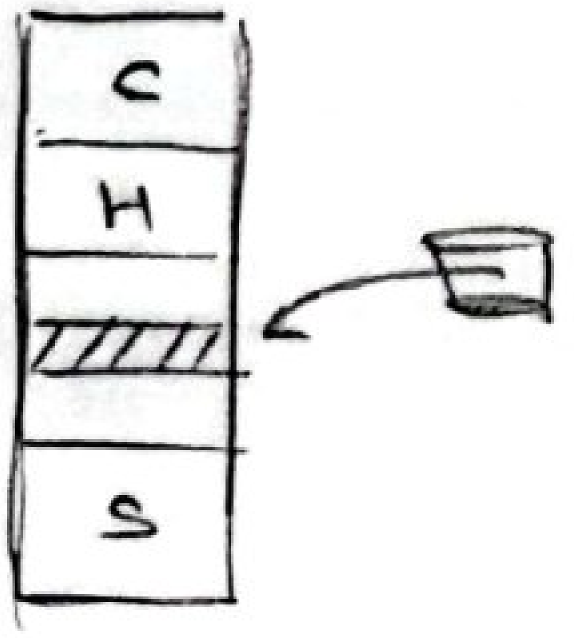
- Files are buffered in memory temporarily. So fsync() flushes all changes to disk.

Hardlinks:-	SoftLinks:- Symbolic links
* creates another file, that points to same inode number...	* is a file, that simply stores pointer to another filename.
* when one deleted, we access inode through another.	* if main file deleted, then inode gone.
* inode maintains a link count. Deletes when Links = 0.	

- mounting a filesystem & devices using mount command. in linux.

→ Memory mapping a file:-

- Alternate way of file access without fd, read(), write().
- mmap() allocates a page in virtual address space.
  - "Anonymous" page for program data
  - file-backed page contains file data.
    (filename argument to mmap())



- Access file data like any other memory locat^n.

L19: File system Implementation:-

Filesystem ≠ device driver.

I think....

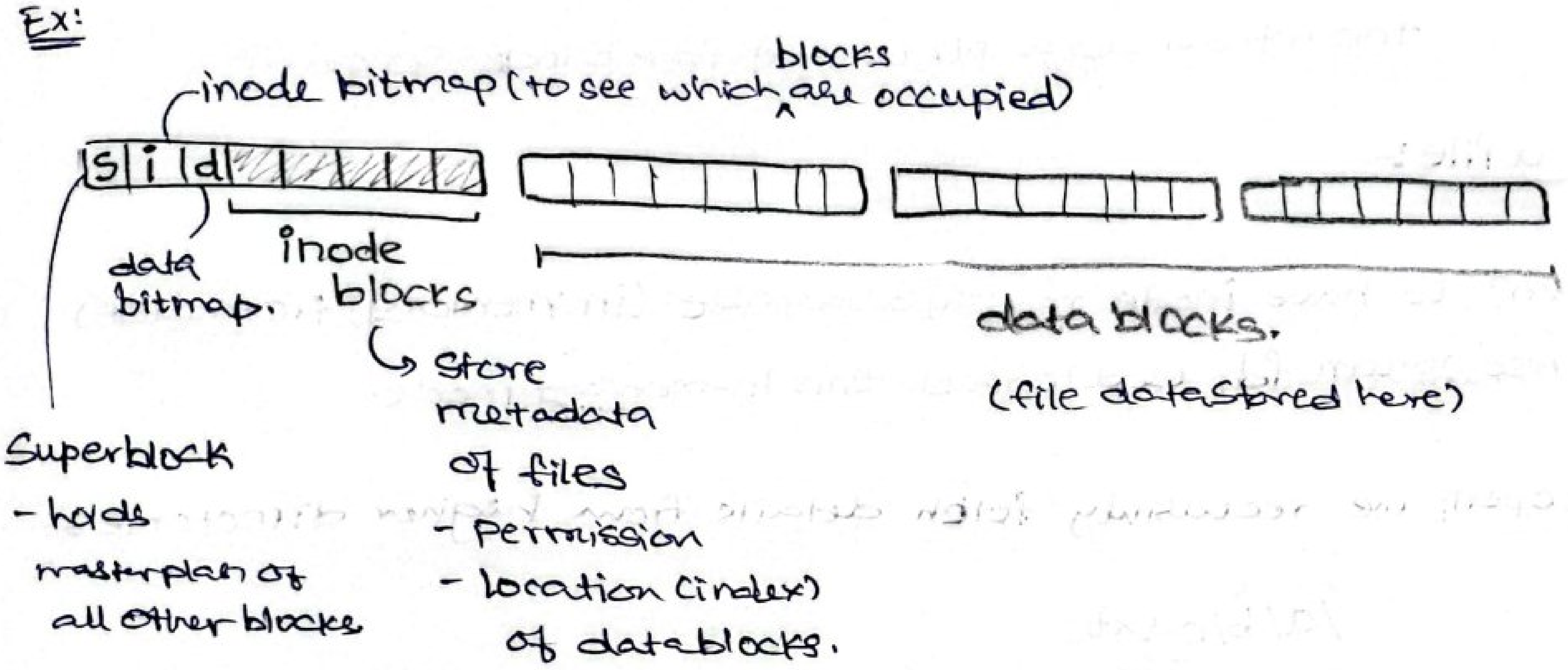
Its 2nd layer i.e. OS code.

* Filesystem is an organisation of files and directories on disk. !

* Two main aspects
- Datastructures to organize
- System call implementations to read, delete etc.

* Disks expose a set of blocks. Say 512B. Filesystem organize files onto these blocks.

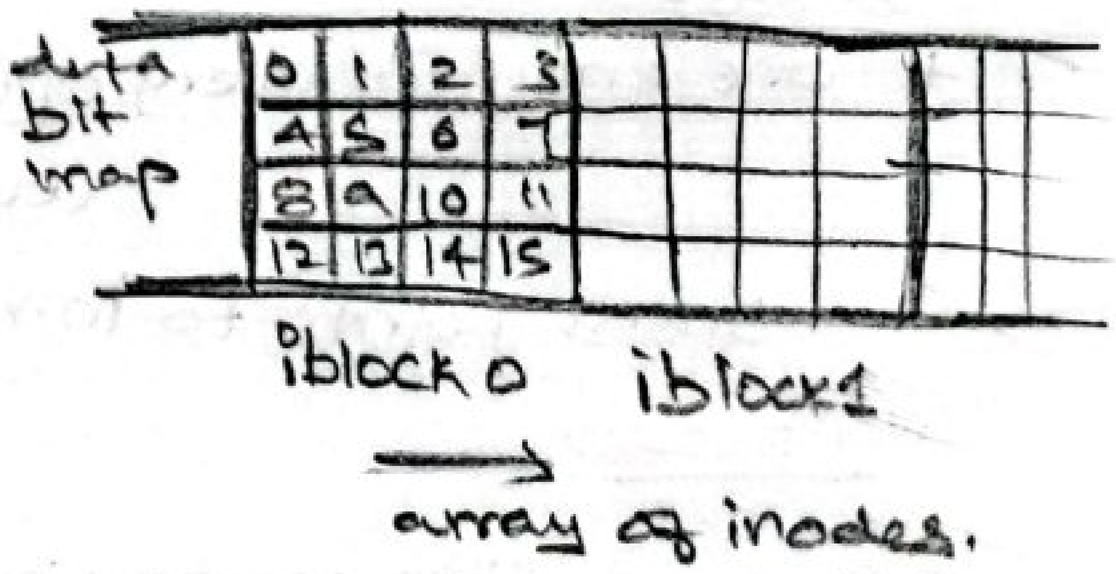
Ex:



→ Inode table:-

* Inode no. is index into this table.

each inode block is:-



* Inode stores:-

1) file metadata - name, access time.

2) pointers to file data. (block numbers)

* file not stored contiguously on disk. Have to track block numbers. So inside inode...

- Direct pointers: no. of first few blocks stored in inode itself. (enough for small files)
- Indirect pointers: inode stores no. of block, which inturn has block numbers of file.

* we similarly have double and triple indirect blocks.

multilevel indexing.

* Or, use File allocation table (FAT); where each [illegible] [illegible] block [illegible] has pointer to next block.

First block addr. stored in inode.

file system !

a seperate filesystem na...

* tracking free blocks: (both inode blocks and datablocks)

- Bitmaps

    Store one bit per block to indicate free/used.

- Freelist

    Superblock stores pointer to 1st free block.

    this inturn stores ptr to next free block so on...

→ Opening a file:-

* why open? to have inode readily available (in memory, from disk)

- Also return fd; used to reach this in-memory inode.

* during open; we recursively fetch details from higher directories

↓↓ ↓

/a/b/c.txt

create
or
load inode

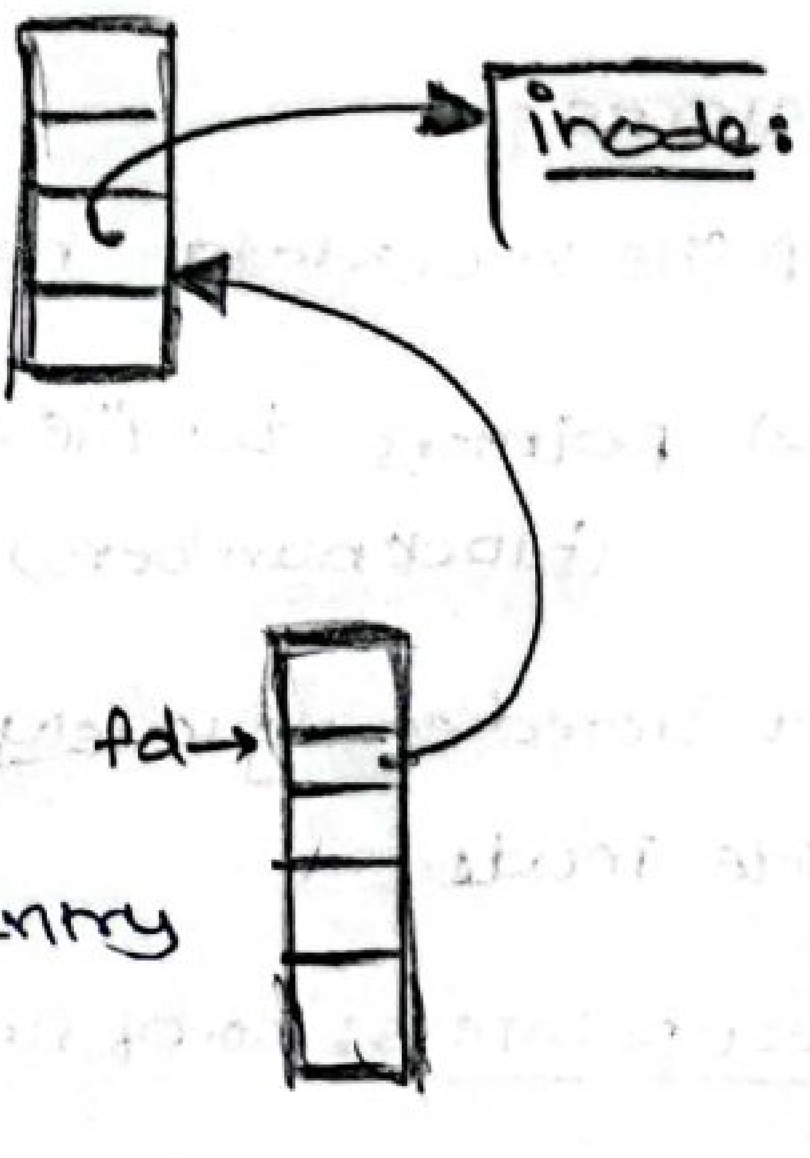
* open file table:-

Global:-

* one entry for every open file. across all processes.

(even pipes, sockets)

- Entry points to in-memory copy of inode.



Per-process:-

* Array of files opened by process.

- fd is index into this array

- entry points to global-open-file table entry

* open() creates entries in both file tables & returns fd.

good.

→ Virtual File System :- (abstraction)

- File systems differ in implementation of datastructures.
- VFS looks at a file system as objects & operations.
- Syscall logic is written on VFS.
- To develope new file system; simply implement functions on VFS objects & provide these to kernel.

* syscall implementation doesnot have to change with FS implementat

Spirit of SW.Dev.

→ Disk buffer cache :-

* recently fetched disk blocks are cached.
  - FS issued read/write are passed onto buffer first.
  - while writing;
    - Synchronous/write-through cache : write to disk imm.
    - Asynchronous/write-back cache : have a dirty bit set & write back when evicted.

* Benefits:
  - Improved performance due to no disk I/O
  - single copy of block in memory (no inconsistency)

* Some applications like databases, avoid caching altogether. to avoid inconsistencies due to crash.

L31,32: XV6 filesystem

Abstractions:

System Call	open() link() read()
operation on fs-data structs	struct dinode{} → disk structure. struct dirent{} struct file{} struct ftable{} for open files. struct inode{} → inmemory inode. struct icache{}
Block I/O layer (disk buffer cache * buffers disk & synchronizes process access to disk.	struct buf{} → disk block buffer. stuct bcache is buf[ ] bread() bwrite() bget() → lock! exclusive access.) brelse() } use below, driver functions...
* use in,out to read/write blocks device drivers (communicate with harddisk)	iderw() → read/write to disk. idestart() → many assembly code. ideintr() → interrupt handler → in, out instructions

logging in disk: we want atomicity on disk changes. (when crash...)

* logging groups disk changes into transaction.
- later, installs the changes into disk, one by one.
- if crash happen after logging; the entries are replayed on restart of disk.

* link count of inode = no. of directory entries pointing to the inode.

diskinode

* for in-memory structures:-

- two processes P,Q open same file, use different file entries in ftable.
  - points to same inode
  - different offsets
- two parent-child process use same ftable entry...
  - Shared offset.

reference no. in struct file is how many ftable entries point to it.

* on disk:-

inodes, datablocks, free bitmap, logs.

in-memory:-

file descriptor array (per process) ⟶ open file table array (global) ⟶ in-memory inode in icache. (global)

* updates to disk happen via buffercache.

- changes to all blocks in a systemcall are wrapped into log for atomicity.

we either want all or none, in case of crash.

Nothing like, inode is updated but data block is trash.