Operating systems (CS347/CS333)

Theory Lab

— Prof. Purshottam Kulkarni

(a.k.a "puru")

Logistics

office: SIA 404, wed: 2.30 - 3.30pm

(primary)
moodle / piazza: discussion forum

Lab submission: moodle

Course Goals

- what is os?
- why os?
- Design principle of os
- Be the os

Teaching operating system: (not linux, rather a toy os)

Textbook: Operating Systems: Three Easy pieces by Remzi and Andrea Arpaci-Dusseau

Syllabus & Course Text

To do: Paste slide content here

Labs

Linux based tools, programming in C,
xV6 – A simple Unix-like teaching operating systems

Assessment

Theory (CS347) # To do: Put % grade
2-3 scheduled quizzes
In-class surprise SAFE quizzes
Mid term exam
End term exam

Lab (CS333)
4 lab quizzes adding to 100% of the grade
(coding quiz)
Labs are open everything, ungraded
Attendance: mandatory, labs will end at 5pm, so can take course after lab.

Lecture 1

(2/8/23)

- Systems / operating systems
  - Set of concepts/ideas/engineering outputs that enable/capabilities to develop/build programs & applications
    - DBs, Architecture, Compilers, assemblies, IDEs, linkers, networking, OS
  - Application neutral level of software support (systems layer)

If you know how systems work, you can work to squeeze more performance out of applications

- What is an OS?
  - "Manages" all programs (e.g. scheduling, all progs progress)
  - Connects hardware and software components
  - Provide interface between user and hardware
  - Provide security/isolation for programs (failure detection) i.e. safe execution
  - Software to build applications
  - enabler for mass/ubiquitous computing

- What are common actions/services of an OS?

- Virtualizing
- start/stop/pause/... programs
- Scheduling
- Allocation, deallocation of memory
- Transmit/receive packets
- Read/write to disk/file
- I/O to external devices

- Why do you need an OS?

- Management layer needed for orchastration
- Hardware independent source
- Programs are buggy
- Abstraction over hardware

- What is a machine? (architecture sense)

Transistors → boolean logic → chips → cpu → interconnects → i/o devices → systems → applications

- We can't program at that level of detail, would take a v. long time
- So, the secret sauce is abstractions (different layers which people work at)

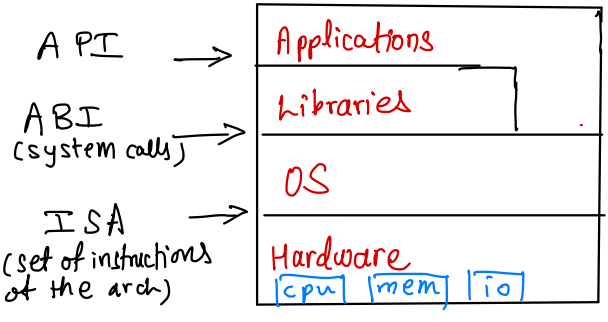
- What is an abstraction ?
  - Conceptual functionality to be consumed/used
  - Details not the concern of user
  - Functionality accessed/consumed via an interface
- What are OS abstractions ?
  - files, socket, process, i/o endpoints

4/8/23

lecture 2

Recap: - Abstractions and interfaces

HW - API vs ABI

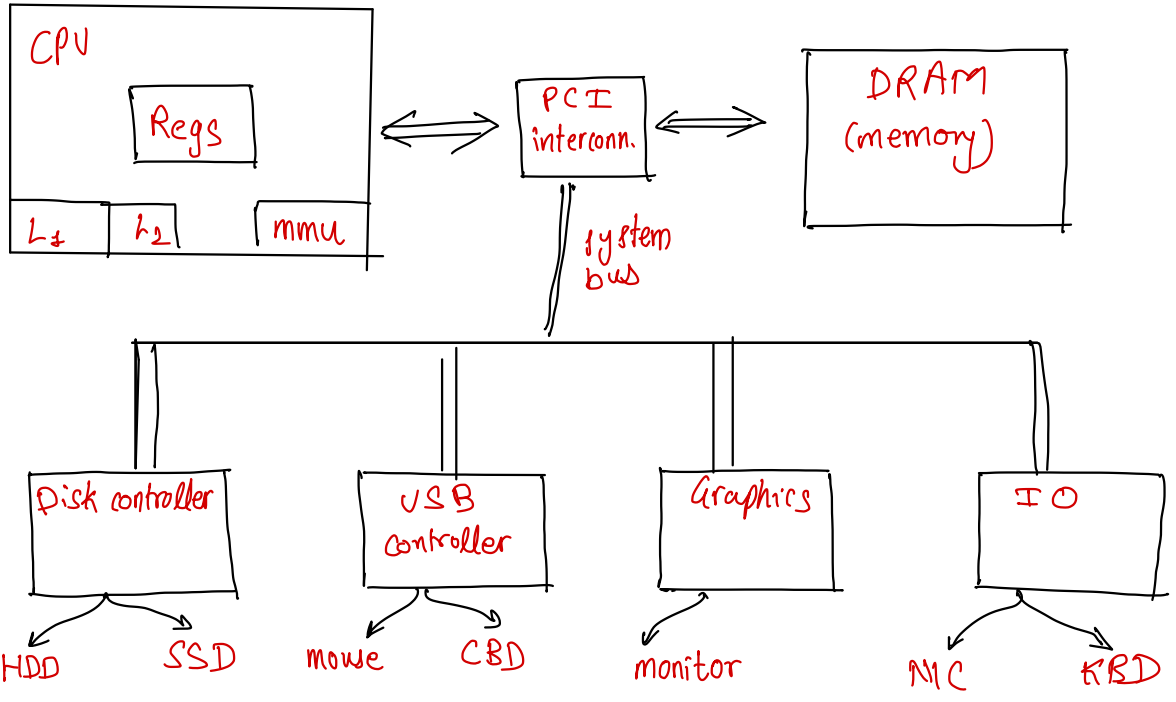


fork, open, close, wait, execv

Simplified view of computing abstraction

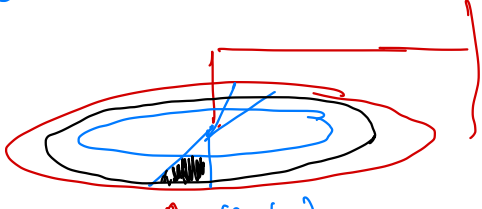
ABI = application binary interface
API = application programming interface

Simplified architecture view of a machine



- Compute resides in the CPU & the controllers
- Disk controller :
  - → Provides a linear blocks interface
  - → HDD = 'n' blocks
  - → Hides block # to sector tracking and disk specific signalling
- CPU: 2-von Neumann model (fetch - decode - execute)

{decoupling what to execute and how to execute}



How to r/w here is handled by disk controller, OS just provides block number (on the OS side → driver)

label : fetch from current pc/ip
decode
execute
next PC
jmp label

→ pc may get updated
(instructions are all stored in DRAM {loaded here}, so that pc can point to the right location of right program)
→ orchestrated by the OS (CPU only understands it's own instruction set)

OS Requirements

- multiple execution instances (efficient use of capacity) (*)
- user management
- disk partitioning / provisioning
- useful abstraction
- Correctness :) (*)
- error handling / robustness → execution isolation property. segfaults shouldn't bring down other programs
- extendable, hardware independent! → OS relies on ISA of hardware, so the OS can't be hardware independent. The ABI should on the other hand be hardware independent! (C++ code is the same on windows and mac)
- isolation (*) → Program instances sharing has to be explicitly set-up - R/w access to memory has to be controlled

(* = non-negotiable)

List of bad things (shouldn't happen)

- Peak into other's memory
- Suck up all the network packets
- Monopolize the CPU
  - Program P_1 is hogging up the CPU. Logical: Take control of CPU to do something else.
  - Interval timer = # cpu cycles after which OS intervenes
    - ↳ register on CPU
  - Monopolization = 1. timer is too high, 2. program manipulates timer
  - All critical setup/configuration/allocation has to be via the OS

(building blocks of OS design) ⇓

(i) Priviledged mode of execution (design principle)
(LDE: Limited direct execution)

← implemented via

- Issue: program and OS both have same ISA, so programs can also try to issue instructions to hardware
- Key: OS is the first program that runs, sets up priviledge modes
- ISA/CPU have to support this priviledge level!
- Every instruction of the ISA has a min. priviledge level for current execution, the OS sets current priviledge level (CPL)

Ring (x86)

(eg. SET timer - PL-Ring 0)

3 → Applications / user mode
2
1
0 → OS

Kernel mode

- Currently, we have established boundaries, next half is how to get control back if lower privileged programs try to extend control

Coming up: Process abstractions.

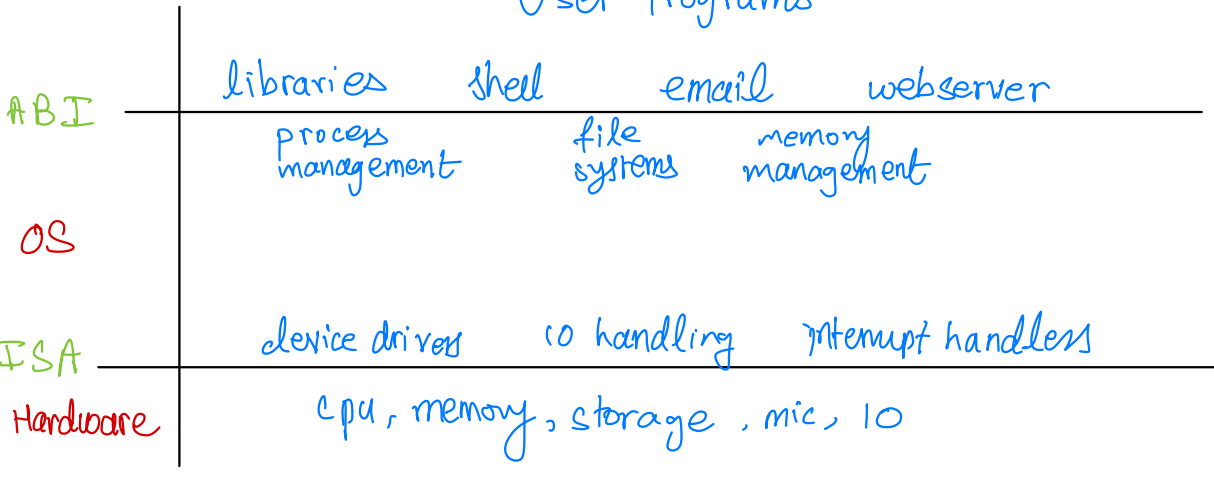
(ii) interrupts / interrupt driven execution

Lecture 3

(09/08/2023)

Recap : OS has to be owner of all (managed) resources for correct/reliable services & abstractions

The programming stack

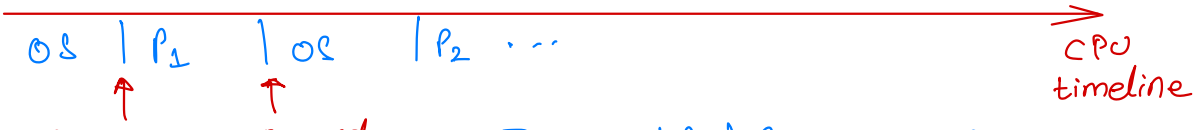


- CPU executes with same privileged level at all time (to change, we need instruction)
- PLs are configurable (update privilege level will be valid only if you're in privilege level 0), e.g. with x86, (last 2 bits of code selector which represents privilege level)
- Each instruction of ISA has an associated min privilege level for current execution
  e.g. SET interval timer cond — PL — ring 0

#homework : OS distribution vs OS vs kernel

Note : Ring 0,1,2,3 are part of hardware spec, OS is built on top of that !

when are privilege levels assigned?



- PL changes
- OS sets reduced PL

- PL reset to ring 0

Is OS | P1 | P2 possible?
Difficult. we need to point to mem where P2 is, but P1 cannot know that, hence we need OS intervention.

Digression
- How does OS adapt to what's happening in programs?
→ It can't code all this in its instructions, so we decouple this case using a "context" and the OS just needs a way to handle and manage those contexts.

(ii) interrupt driven execution (hardware assisted)

Some voltage levels (pulse) trigger interrupt on hardware. So, we use a CPU interface

What is an interrupt?
- Event to stop/pause CPU
- change PL to ring 0 (x86 specific)

How is an interrupt generated?
- change in a voltage levels based on a pin/pins
- Instruction in x86 int/INT ____ (interrupt number)

Q. How does OS put intr. in correct place (can't write code?)

e.g. keyboard, mouse, syscalls

The world is not deterministic: failure in memory allocation, waiting for input (all external I/O)

Can you interrupt OS indefinitely by taking input @ OS freq.?
OS can use polling instead. if not OS can be screwed dealing with keyboard interrupts (DOS attack). (if high freq / hardware

Why interrupts?
- World is not deterministic
- All io events are non-deterministic (key press, disk read completion, arrival of new packets)
- Used as a building block for the ABI/system call interface

interrupts

- hardware : i/o devices
- software
  - implicit
    - exceptions (null pt deref, div by zero : CPU can't deal with situation)
  - explicit
    - Program wants OS to perform a service (ABI or syscall)

what do interrupts do ?

- we need to do who caused the interrupt ? (interrupt number/vector) keyboard press, then we need to land in OS code handling I/O. (how? will follow)

(e.g. key → interface → CPU → data on pins for sthere to read)
(flow for keyboard interrupt)

- OS can instruct to disable interrupts on certain lines to prevent hardware based DOS attack sabotage

Program (binary)

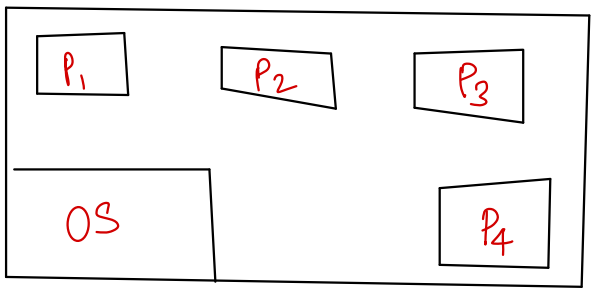
source code + libraries → executable binary program

- Software paper-weight (supposedly encodes some instructions, does nothing)
- ELF format

Process

~ As an instance of a program in execution (new abstraction by OS)

~ base unit to deliver & manage services.



OS is also a program, hence loaded into memory

OS needs to know where programs are in memory

. metadata of a process is stored an entity PCB (process controller block)

e.g. of metadata of processes

- PID : process identifier, PPID

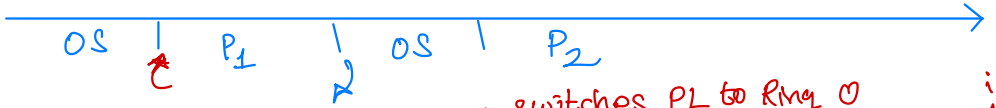
(Duplicate proc. : fork, Load program into process : exec)

- Memory alloc into / bounds
- Open files
- CPU registers (context)
- Kernel stack

Lecture 4 (10/08/2023)

Recap : 1. You are yourself
2. The world(?) is not deterministic
3. World peace needs I/O

- LDE
- interrupts, process abstraction



interrupt → switches PL to Ring 0
→ jumps to OS code/program

interrupt handler (ISR) (interrupt service routine)

P_1 wants to hog the CPU, no software interrupt, how to interrupt P_1 ? Timer interrupt !

(e.g. 10 ms it fires)

10 ms value is configurable, but just a default value.

1. Allow OS to get back control
2. Only way to keep track of time for the OS

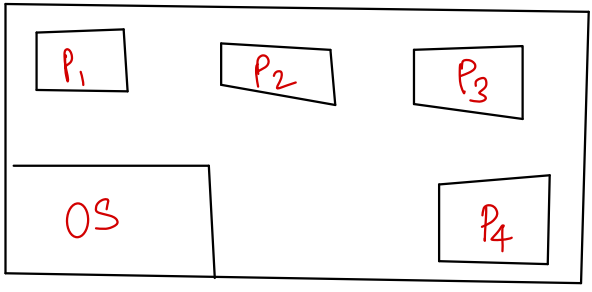
why ?

#cpu cycles = 3 GHz, 10 ms is good enough for "long enough"

1. OS schedule 10 times / min (e.g.)
2. disk read / process termination after a long time

Will I/O interrupt reset timer counter ? usually, no.

what happens to a process after termination ? context !



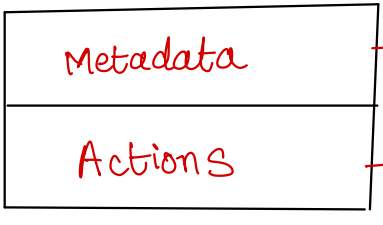
- Executing programs is decoupled from writing them - (this is the OS promise, setting up execution of program)

- Load program in memory
- Allocate memory to program entity/process
- schedule → PC points to instructions in memory

OS abstraction

(all done by the OS!)
(if not, the burden would fall on users).

Two block description of all things OS



PCB : Process Control Block
(Linux : task_struct, xv6 :)
(variable name that linux uses in C code for the PCB)

→ scheduling
mem. allocation

PCB content
- pid, ppid
- list of files
- memory allocation information
- space of registers
- Kernel stack pointer
- Usage information

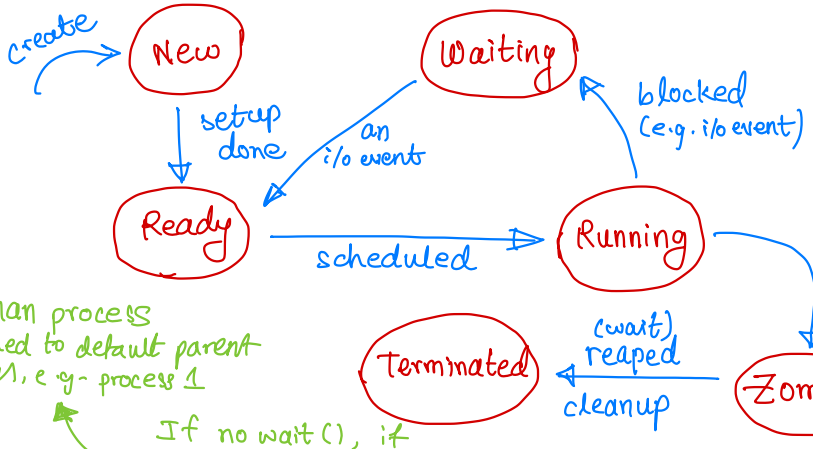
- when you create a process, first thing is creating the PCB, then allocating memory, point PC to start of the memory & then dump it on CPU.
- If process is over, OS needs to reclaim the PCB, i.e. reclaim memory, other resources.

Reasons to wait (process)?
- CPU is being hogged by another process
- waiting for input or sleeping

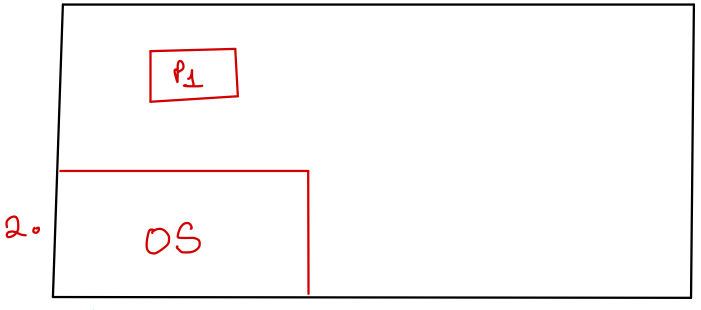
So, situation/status of all processes may be different, this is important because OS will schedule only processes which are ready to run!
where do you store this information? → PCB

Process state transition diagram

multi-core machine, then you can't have 1 process on 2 cpus.



Memory



1. BIOS
2. OS bootloader
3. Loads process 1

(this box provides abstractions, needs someone to use it)
(unless there's a user, it's a paperweight)
(process is the entity that uses the OS!)

- Game plan for OS to have a purpose
  - step 1 : As part of bootup, load itself into memory
  - step 2 : Handcraft a user-level process & jump to process, instructions start of. (init process pid 1)
  - step 3 : Consumer of services (that os has advertised) is in execution (now we have one process)

Since we are in user land, we can now request for more processes
OS doesn't want to worry about what to start, so asking
user to build the init program.

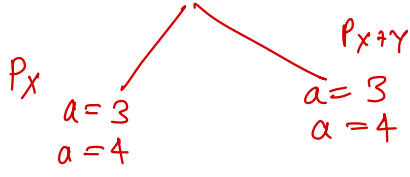
Some system calls,

fork	exec	wait
• Creates (duplicate) a process (next instruction is the one after fork for both) • Creates a new PCB • Populates PCB entries	• Load program into memory & makes it part of a process	• return with a return value (gets pushed out of a process, and sent to parent process)

```
int a = 3;
fork();
a++
```

in the parent process, returns pid of child

in the child process, returns zero.



e.g.
```
int a=3;
if (fork()==0) { // child
    a=a+1;
    print(a);
}
else {
    a=a-1;
    print(a);
}
```

← at this pt, copies state of all vars to child's PCB

→ 4 (child process)

→ 2 (parent process)

← could start another process i.e. load another program via execl()
+ family of system calls

Q. How is wait() retired? OS magic!

missed lec (16/08/23) ; refer ostep chapter 5

Lecture 6 — Signals and Scheduling (18/08/23)

Recap : fork, exec, copy-on-write

(i) Signals

- OS-assisted / software defined mechanism to deliver events/interrupts/control signals to processes.
- a signal / pending signal are part of OS state
- signals are processed before a process is scheduled on CPU.

Signal handling

- Default OS action
- User mode handler per signal (via registration, e.g. lab 3)

Types of signals

- Override or not (i.e. can you write a handler for it)
- Type of default action
  1. Terminate
  2. dump core (memory content of process or file)

SIGKILL	9	terminate, can't override
SIG INT	2	interrupt, terminate, overridable
SIG SEGV	11	overridable, dump core
SIG STOP	19	cannot override change process state to blocking
SIG CONT	18	Cannot override change status
SIG CHILD		can override (signal on child process terminate), ignore

Note : System delivery depends on the system calls (kill)

OS doesn't kill p2 right away, it just parks that signal in memory, gets killed next time it is scheduled

Note : SIGSEGV can be overriden to give more information about segfaults. But if you return without exiting program, you return to same instruction causing the signal, hence no point of doing so.

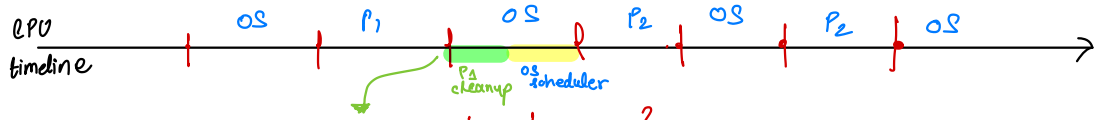
exit() system call tells the OS that program is done.

```
SIGCHILD
  handle_child_exit(){
      pid = waitpid(-1, &status, WNOHONGUP);
  }
  main(){
      signal(SIGCHILD, handle_child_exit);
      fork();
      fork();
      // long running code
  }
```

(sent when zombie child, current parent will get signal)

As part of clean up of PCB of child, OS sends this signal.
Child DOES NOT send this signal, it just calls exit() and tells OS that it's done.

(ii) scheduling



- When does this transition happen ?
- How does this transition happen ?
- what does the OS do in its time over the CPU ?

Say exit() syscall causes $P_1 \rightarrow OS$

1. Deal with exit(), i.e. cleanup of P_1
2. Handle which process to schedule next (i.e. go back to user land), i.e. OS scheduler.
   (can do other stuff in OS land as well)

Q. What does zombie process retain in its PCB?
→ exit status, pid information (much smaller memory footprint)

List of actions scheduler has to do to switch from P_1 to P_2.

(i) An event that triggers the scheduler (interrupt or system call)

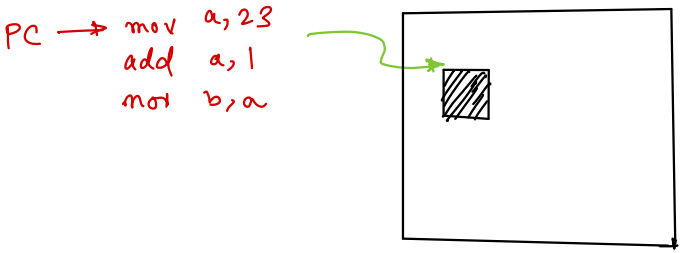
(iii) Context switch (execution "state" of a process)

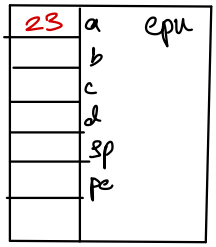
- Save context of outgoing process
- restore context of incoming process

context = registers on cpu during execution (cpu state)

Note: If program uses stack or heap, the memory allocated on RAM is pointer is stored in PCB, so that's already stored!

Scam: Make every process believe they have their own cpu
1. Switch process so fast that illusion is everything runs simul.
2. cpu runs on same instructions it left (i.e. quick save & restores)





Where to store the context ? ← PCB (kernel stack)

(ii) Choose the next process to execute (before context switch)

scheduling policy → TBD, will focus on mechanism for now

Note: When interrupt comes, & switch out, OS instructions run on cpu! So how does context get stored exactly? (handwavy arguement, make 2 copies of everything on stack)

Scheduling action is a function of the OS state.

- Ready queue (schedulable processes)
- multi-CPU setup (scheduling multiple processes while dequeue)

- Single ready queue (needs synchronization)
- Have multiple ready queues (per cpu ready queue)
  (issue: which ready queue to go to, if empty ready queue, underutilisation)
  — rebalancing.

Lecture 7

(23/08/23)

Recap: Signals & scheduling mechanism (policy)

System call interface

- open, close, fork, read, write, exec, exit, wait, dup, pipe, waitpid, kill
- system call is an OS mechanism to request for OS services/functionality.

(i) system call interface is the list/collection of all these calls.

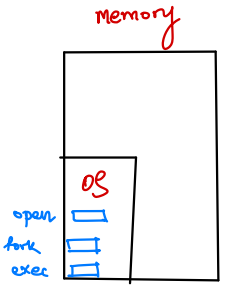
(ii) ABI vs API

source/binary files (object files)

system call + binary loaded to perform instructions

binary (executable)

(syscall & ABI)



fd = open (filename, RDWR, 0666);
pid = fork ();

(iii) a few things need to be in place for system calls

- How to invoke a system call?
- mechanism to switch privilege levels.
- context save & restore
- mechanism to handle arguements/ret. value
- specify the system call fn.
- where in binary is required functionality

(iv) all system calls are interrupts!

interrupts
- hardware : external IO
- software
  - (implicit) failures & exceptions
  - (explicit) int (x86 instruction) {part of ISA}

int 0x80
(interrupt id)

Explicit software interrupt (e.g. int 0x80)

- software CPU to highest privilege level
- save context of running process (all regs of CPU) in the kernel stack
- switches to kernel stack
- jump to the interrupt handler

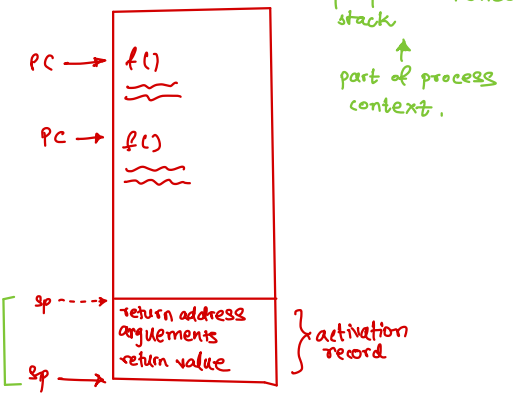
↳ part of the PCB (location of stack = register)

Kernel stack = Stack memory of OS
(OS has this reserved memory)

* Unless $sp of cpu is changed, you can store sp in PCBs with no effect.

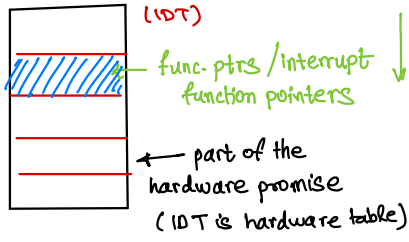
(x86) iret (opposite of int)

- restores context of CPU regs
- changes PL to user mode
- jumps to PC in user mode / code
  (location is unclear)



(v) Interrupt Handler
(ISR – interrupt service routine)

- where is the interrupt descriptor table (IDT)

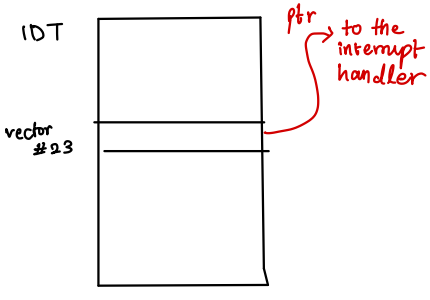


- Where is the IDT? hardware provides IDTR register which points to start of IDT
  who populates IDTR? OS copies in boot-time (lidt ← instruction of ISA, copies into arg register)

Lecture

25/08/23

Recap : ABI, API, requirements of system call mechanism, int, IDT, IDTR



System call ⇒ explicit software interrupt

System call identifier & arguements

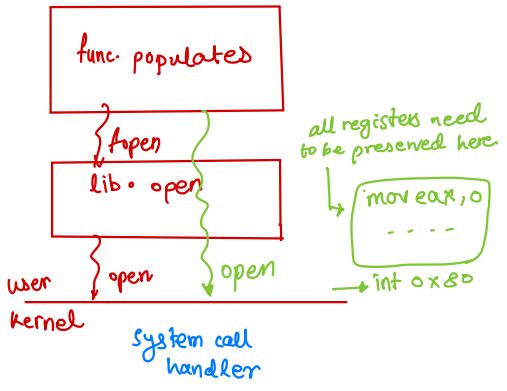
- via CPU registers (eax, ebx, ...)
  ↑ stores system call number
- via memory/user stack (sp)

(user) interrupt → hardware ↑ OS

- $0, data abt interrupt must be stored in hardware.

↑ populating registers. not the OS, someone in user space handles this.

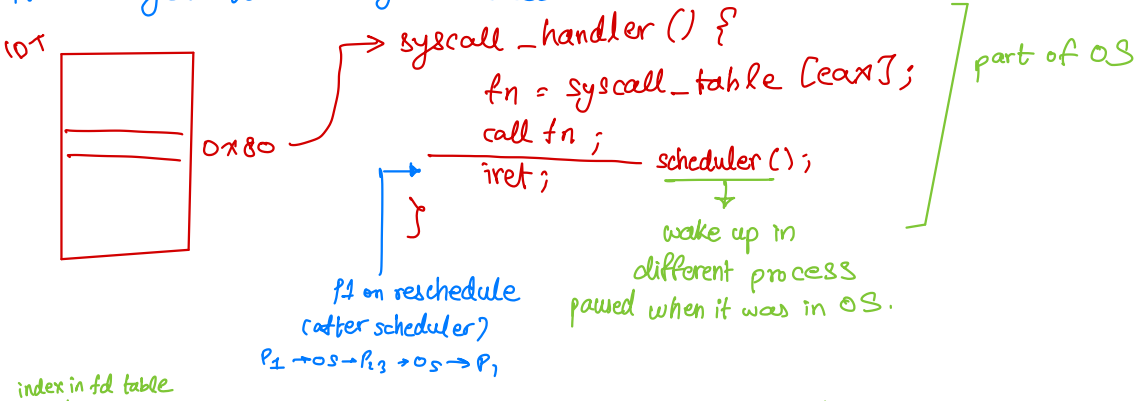
(1)



(2) will save CPU regs on the kernel stack

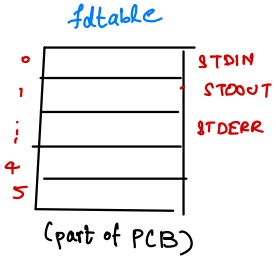
Software PLS → jump to handler.

How to get to THE system call.



index in fd table

```
int fd = open();
```



syscall in two processes gives diff results, e.g. open() for same file in diff processes - {context dependent}

XV6 Introduction

- teaching OS MIT
- xv6 the source code ⇒ implements the OS logic
  ⇒ generates the OS program (binary)
  ⇒ xv6.img (1)
- qemu ⇒ machine emulator (converts one ISA to other)
  - xv6.img – kernel/OS to boot the machine into
  - fs.img – root/default file system/files available on boot up.

makefile → xv6.img, fs.img, qemu

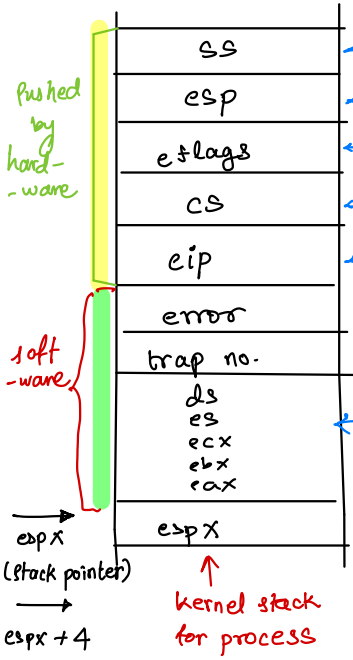
30/8/23

lecture 9

Recap: xv6 intro

- makefile
  - xv6.img : kernel
  - fs.img : file system

traps.h → list of all interrupts
vectors.S → IDT entry for each trap/interrupt vector
trapasm.S → contains the entry point of all traps (trap handling setup)
trap.C → generic trap handler
syscall.C → syscall handler



int instruction has these interrupt mechanisms in-place.

mixture of control & exec. state of flag ≡ state of current context.

where instruction?

- set via hardware action (or) interrupt instruction.

e.g.
```
mov 0, eax
int 0x64
```
issued by user mode process

1. change in PL (privilege level) (user → kernel)
2. push regs on kernel stack
3. jump to IDT pointer address

int <> interrupt number, eax register holds arguement.

≡ process context in which interrupt gets handled

≡ trapframe (xv6 lingo)

1. check which interrupt (i.e. trapno), if syscall, handle system call
2. If some other interrupt (e.g. timer, I/O, etc.), handle them accordingly.
3. If an exception, (e.g. memory restricted, div by zero), etc.

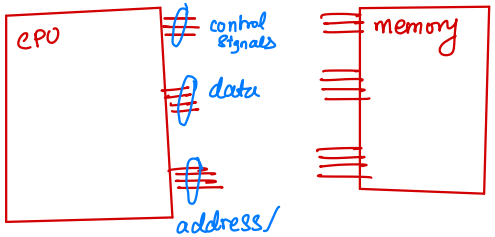
Syscall : eax register in trapframe pointer, then do cases depending on which system call.
After return, update value in eax to return value (i.e. update eax in trapframe).

popal : dumps values from stack to registers. Hence, useful for restoring context.
iret instruction cleans up only hardware components of kernel stack, the software ones need to be popped from stack into registers.

Memory virtualization

Hardware

- CPU is the main entity which accesses the memory



Function of pins :
1. Read / write ?
2. where to read ?
3. what to read / write ?

address/ location (address bus) ⇒ 8 bit vs 16 bit vs 32 bit vs 64 bit CPU. (i.e. can do arithmetic in that size like 32 bit arithmetic)

↳ width of the address bus

32 bit → 2^{32} addresses.

byte memory ⇒ 2^{32} bytes = 4 GB memory.

Lecture 17

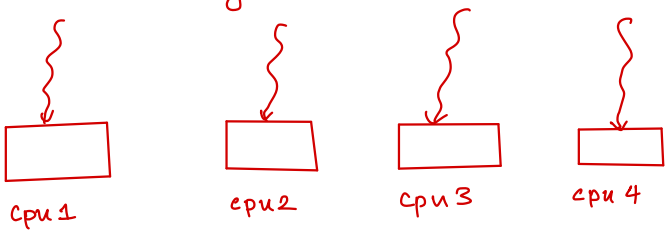
(4/10/23)

xv6 help session
cc 103 (Wednesday)

1. Lab 6 : scheduling in xv6
2. Lab 7 : synchronization
3. Lab 8 → LabQuiz → .... → LabQuiz 4.

Announcements

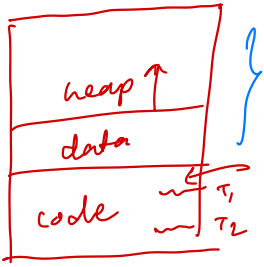
Synchronization/Concurrency



instances
threads of execution.

- multi-process (multiple indep exec)
- multi-threaded : thread
  - execution context in a process
  - shared process address space
    - data, code, heap, stack
      - unique stack

(multiple exec shared data)



* Threads can run in parallel or sequentially
multiple context = multi-threading ≠ multi-CPU.

(e.g. of multi-threading : web-server)

Issues :

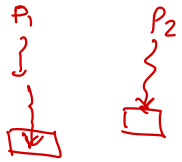
1. Race conditions :

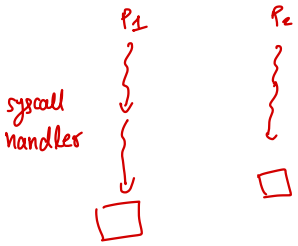
then, a gets incremented thrice! 24, 25, 26

If all T_1, T_2, T_3 read 23 together then a ← 24

Shared state : e.g. ptable

Syscall handler





read() can have access to shared state (ref file)

(i) P1 & P2 exit() → p->state = ZOMBIE → yield() → sched() → scheduler(); → np = getnextprocess() (both choose P3)

proc->state = RUNNABLE ← get a process from the ready queue

(2 instances of P3 on 2 diff CPUs) → Chaos.

(2 inst without fork is chaos)

(ii) P1 & P2 kalloc() {simultaneous}

shared free list : they get same page but need different pages.

(iii) fork()

child may have same pid for two threads, might land in same proc table entry (same PCB for two diff childs)

1. shared PCB list/state

— same PCB is a problem

— same pid is a problem

#race condition

(i) multiple instances of execution with shared data/state

(ii) read-modify-update actions on a shared state.

(race cond^{n} ≡ non-det. behaviour ≠ issue but issue may happen)

egs of race conditions in kernel mode

P1 { read

P2 { kill(P1) — can screw over P1

single CPU + multi CPU

(i) syscall + syscall

(ii) syscall + interrupt handler

→ P1 → Blocked

→ P1 → RUNNABLE

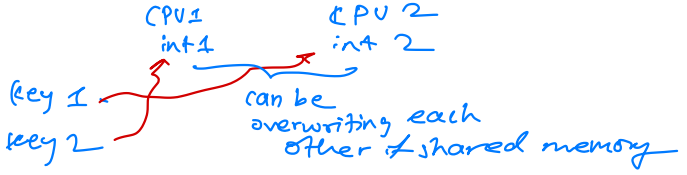
→ P1 → KILLED

} simultaneous before completion.

(#ques : so what? And how sync. techniques help?)

common in multi CPU

(iii) interrupt handler + interrupt handler



synchronization techniques to avoid undesirable outcomes due to race cond^n

(i) syscall + syscall

* Disable preemption, run system calls to completion (no halfway shut)

→ Blocking cpu till anything happens

multi-cpu (other cpu are still an issue !) P1, P2 both syscall. Issue !

(ii) system call + interrupts.

* Disable interrupts – works for single cpu system (if interrupt all cpu, very inefficient)

issue: 1. inefficiency

2. semantics, interrupts coming in are dropped ! Queued up, eventually queue is full)

disable time of size of buffer.

3. multiple cpus are still a problem

(3) locking/mutual exclusion

Critical section

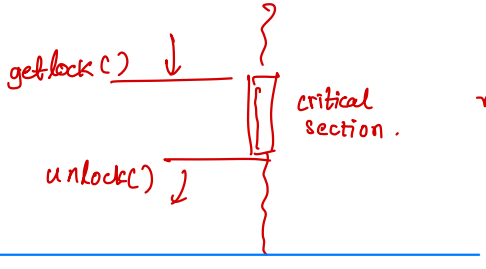
↑

disallow parallelism (mutual exclusion)

Lecture #18

Syncronization Primitives

05/08/23



read - modify - update.

Candidates for a "key"

1. What is a lock in programming sense

↳ variable / object

Machine: memory address.

```
int lock;
lock = 0    // lock available
lock = 1    // lock taken.
```

} now add conditional execution

```
int getlock ( lock=1 ) {
while (a( l -> locked == 0) {//available
        b  l -> locked = = 1 ;
    }
    return 1;
}
```

can occur many sec.

```
struct lock {
    int locked ;
    int id;
}
```

Issue : 1. Many threads can get lock at same time
2. return can't happen, others have to wait.

Bad → $T_1 a \; T_2 a \; T_1 b \; T_2 b$
0 0 1 1

— Status of lock = non-atomic

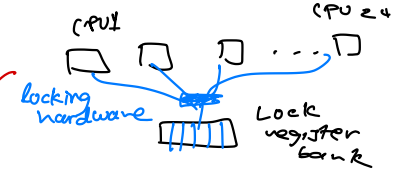
① multiple inst./ c-statements involved.

② single c-statement ← doesn't mean anything, can } lock atomicity
still have multiple instructions.

③ Single ISA instruction ← pipeline R – D – E in parallel.
pipelining of a set of instructions. So, wel pipelining can cause both reads before exec.

inc %mem → read mem value, update, write.
Hence, can still be parallelized.

Unless atomicity, no possibility of a lock.

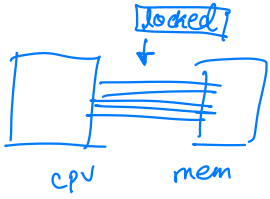


radical hardware (costly)

we want a single instruction that can do both of these things.

lock : inc %mem not good enough. we need check and set

prefix x86 : locks the address bus



* Locks depend on ISA instr involving check and set both. / compare and swap.

E.g. TSL LOCK, R0 — atomic
test & set
xchng
- old value = LOCK
- LOCK = R0
- return old value

#spin lock

```
getlock : mov  R0, 1
          TSL  LOCK, R0
          CMP  R0, 0
          JNZ  getlock
          RET

unlock : mov LOCK, 0
         RET
```

assuming return value is in R0.

Implicit assumption

Two TSL can't run at same time.

② mutex / sleeplock

- If lock not available sleep (proc → state = BLOCKED) till lock is available & recheck
- On unlock wakeup all threads waiting on this lock.
  - ↳ proc → state = RUNNABLE;

```
mutex lock : mov R0, 1
             TSL LOCK, R0
             CMP R0, 0
             JZ  OK
             CALL YIELD
             JMP mutex lock
OK : RET
```

YIELD: proc → state = BLOCKED; proc → chan = &LOCK;

← wakes up here

→ after waking up to continue checking

```
mutex unlock :
    mov LOCK, 0
    call wakeup (& LOCK)
```

wakeup: iterates over all blocked processes

if proc → chan == & LOCK

proc → state = RUNNABLE

Q. who gets the CPU first?

* Re-implement scheduler logic in wakeup (Redundant).
* So, just wake everyone up, & let scheduler decide.

③

```
struct mutex {
    int id;
    int locked;   <  0 ~ available
                  <  1 ~ locked.
}

mutex lock (mutex *m) {
while  if (m → locked == 1)
           sleep (m → id);
    m → locked = 1;
}
```

not atomic.

(xchng (m → locked, 1) != 0)

compiler converts this to appropriate inst.

#define xchng as a macro.

this is what XV6 does!

```
mutex unlock (mutex *m) {
    m->locked = 0;
    wakeup (m->id);
}
```

mutex ⊆ cond^n vars

But cond^n could be anything, not just 0 or 1 →

So we can't have just a static check.

sleep – wakeup

wait – signal

(*) sleeplock – sleepunlock.