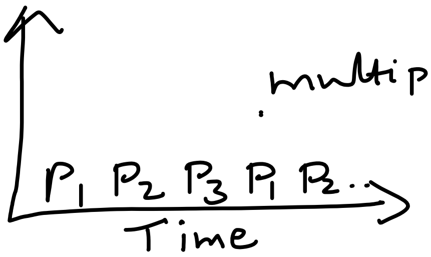
Lecture 4

Goal: Multi-programming

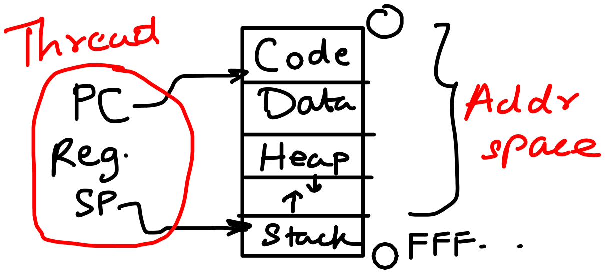
run N programs on M processors
($N >> M$)

suppose $m = 1$

multiplexing - time sharing.

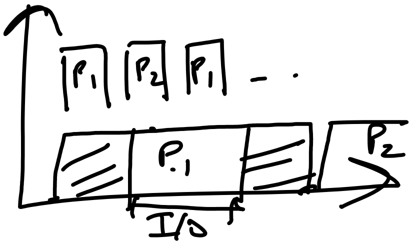


Recall:



(*) What is the benefit of multi-programming?

-



→ Improve utilization of resources

I/O - read a file block

Speed of CPU >> I/O op.

→ Interactive system

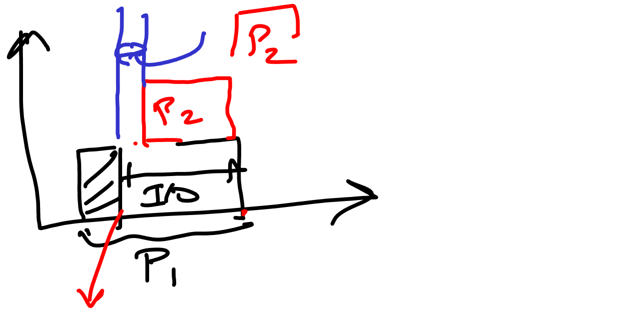
- handle interrupts

(*) The OS performs the switch from 1 prog to another prog.

- {Save the running prog's state & load the state of other prog.}

Important Considerations!

① Efficiency



Switching time ≈ I/O

switching time → 0

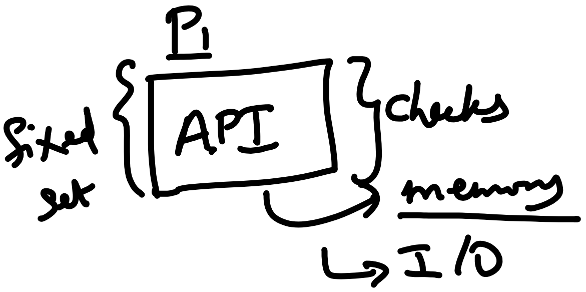
② Safety

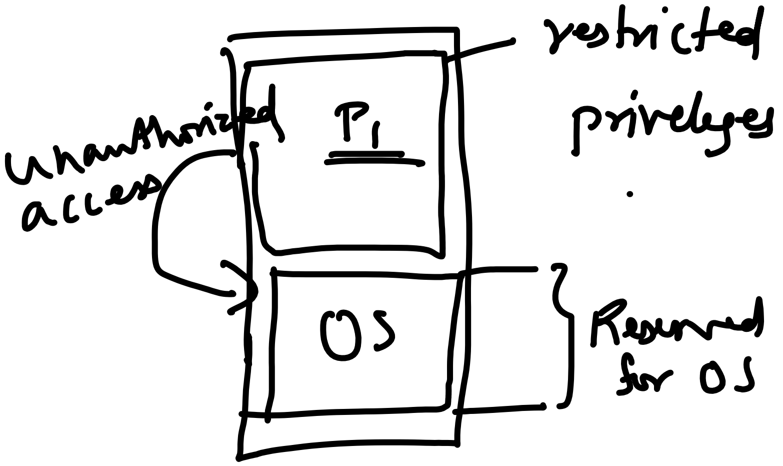
Switching from 1 prog. to another should not corrupt the state of any prog/OS.

Which component performs the switch?

- S/W or [✓ h/W]

Protection

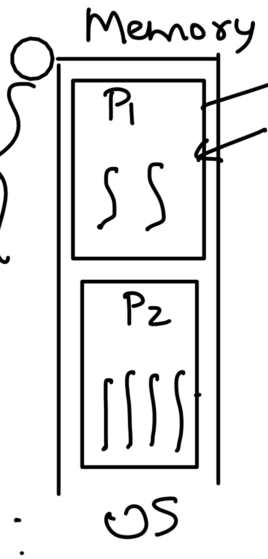




Process Abstraction (*)2.2|2.3

Mechanism to restrict Pi's access to mem.

Base - Bound register (*)2.4-6

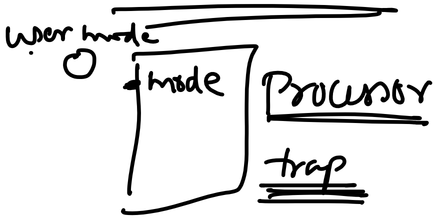


Add A, B write

Addr. space with 2 threads

syscall

→ read a file



Process - Restricted privelges OS - unrestricted privelyes

Issues to be addressed

(I) Process wants to do something restricted?
- eg. I/O read a file

Soln:

different privelge levels + mechanism to change from 1 privelge level to another } sys.call, interrupts, trap (exceptions)

(*)2.7-9

dual-mode of operation

e.g. (*) 2.10-13

fork()

Inst. Trap

1) saves state of running program
 - done by h/w

2) change privelege level from user - kernel

3) run the correct trap handler PC - user code → syscall

Kernel mode { PC - prog. counter
 OS-kernel.

how do you find the sys-call handler? { User prog
 μPC:

RTU - μPC → PC
 1 → 0
 User mode

Kernel boots

- Kernel mode
- Initializes sys.call tables

1
2
3 → def^{n} - open (-Check for access to the file
 -read the block into kernel mem
 - RTU)

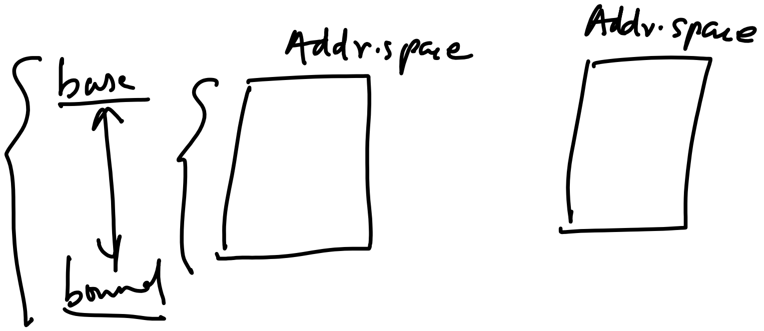
user mode : op^{n}

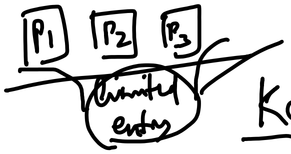
restricted op^{n} - trap → user to kernel - traphandler

Recap

Multi-programming

↓ efficiency ↓ safety





Kernel mode

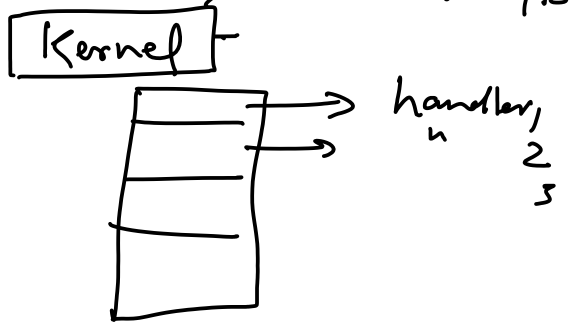
Protection: ↑

- OS provides the process abstraction → (Addr.space)

↳ Privelyed inst. - non-privelyed inst. → user mode

↳ Memory protection

↳ Timer interrupts



divide by zero user mode

↓ exception

→ kernel mode

→ saves the prog state

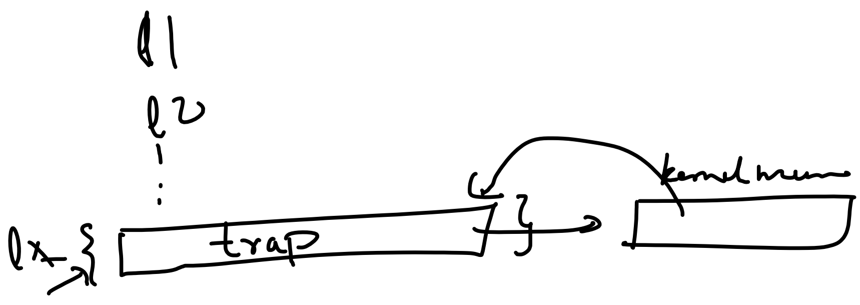
→ handler -

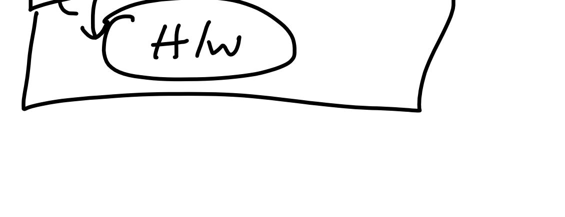
illusion

gdb - breakpoints

exceptions

Privelyed





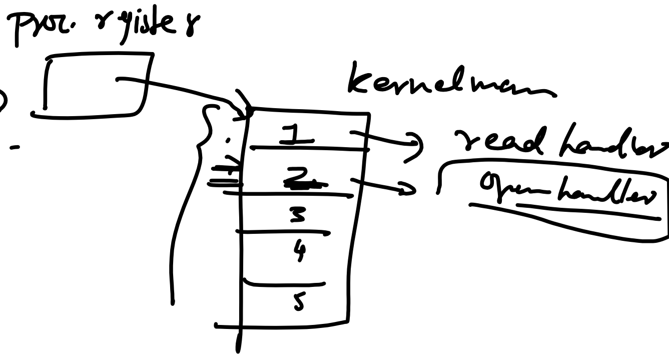
(II) Context switching - how does OS stop 1 process & load another process?

(*) 2.14 - 19

Limited entry pt.

syscalls - read, open,

#syscall



main() {
  func(2,3);
}

func(a, b) {

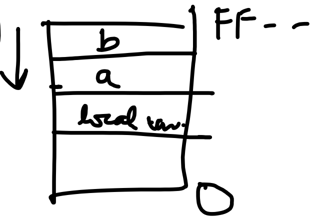
syscall file_open() -

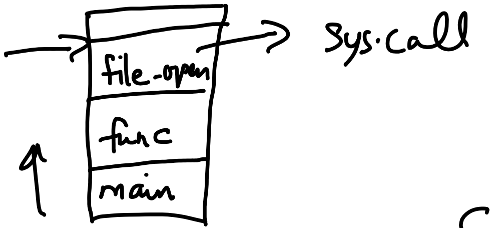
}

User Stack

kernel stack

Stack - func(a,b)

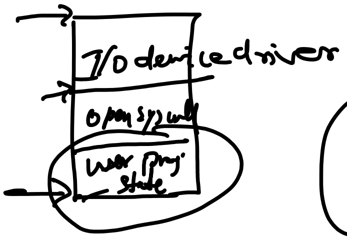




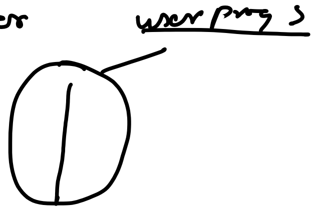
kernel should NOT user user-level stack

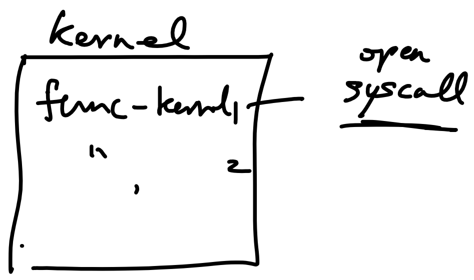
Security
1. SP may be invalid
2. stack can be manipulated by some thread

wait for I/O



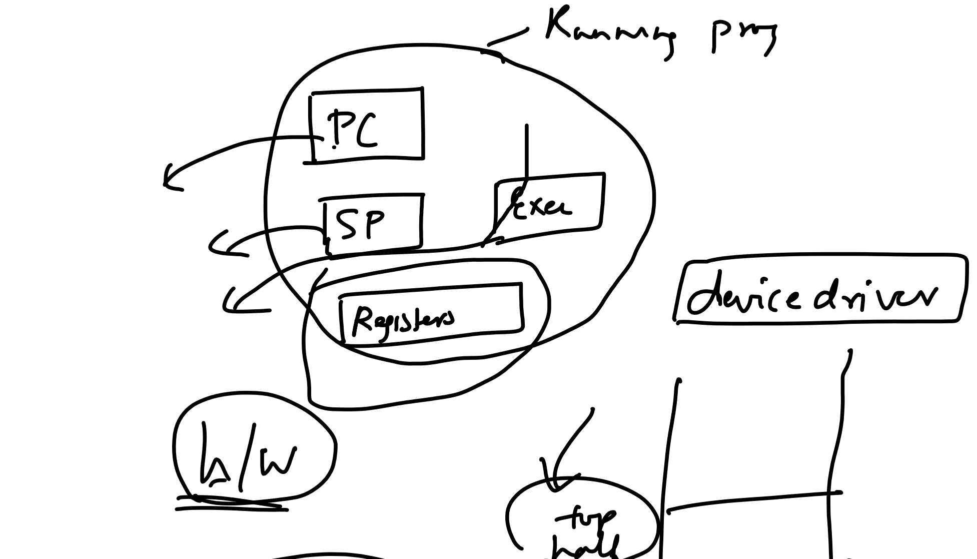
Kernel interrupt stack per process



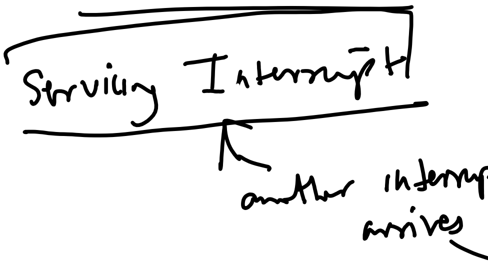


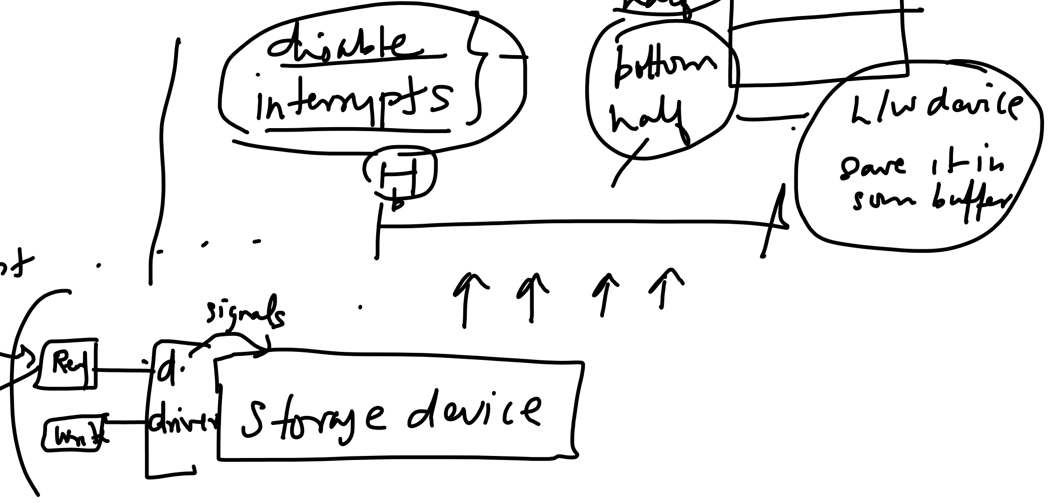
user stub

```
file_open ( ) {
    push ~~#~~sys call
    trap
}
```

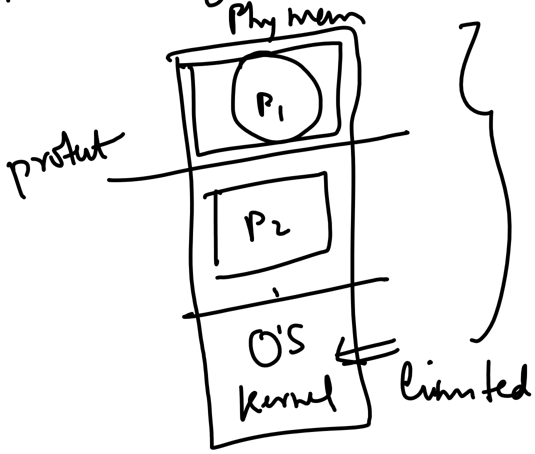


Interrupt





Multiprogramming → improve utilization



— protection

↳ privelyed inst.

↳ based bound mem protection

↳ Timer Interrupt

2002 - Apple's OS

reboot the Mac X yield() program — voluntarily relinquishes its CPU

Timer Interrupt → OS has to decide which proc. to run

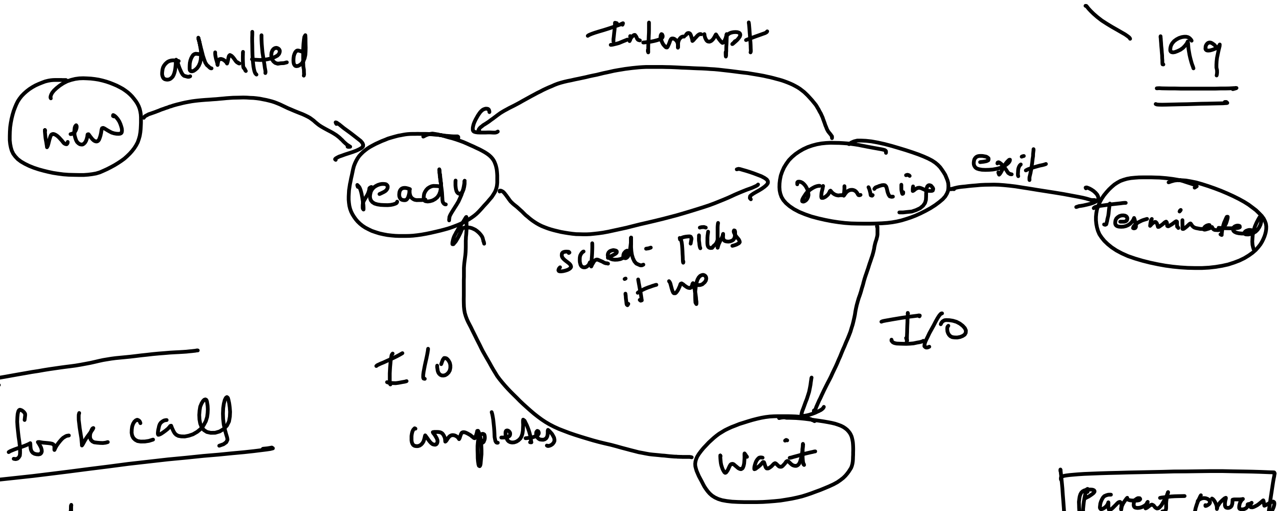
$P_1 \xrightarrow{I/O}$ I/O not complete

State of a process

fork() → creating a new process

1 processor

- 200 processes
- 1 process running
- 199



On fork call

- Create

PCB$_1$ in the kernel

- create/initialize the addr space
- Inherits the exec context of the parent
- Add the PCB$_1$ to the ready list

Parent process fork

→ exec

→ kernel

Shell - ls

PCB

Pid, user, privileges
- open fd, status

→ kernel memory -

} - OS kernel

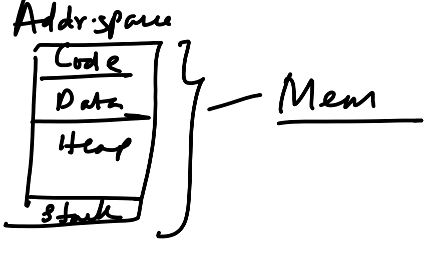
signals

→ User

↓ sys call

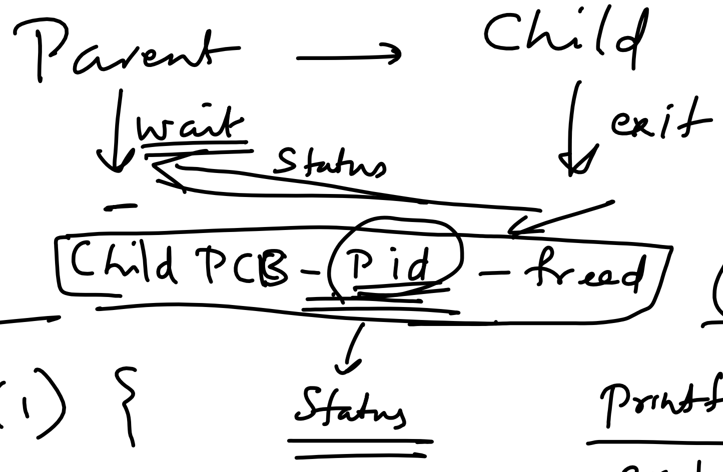
kernel

Addr space



Zombie process

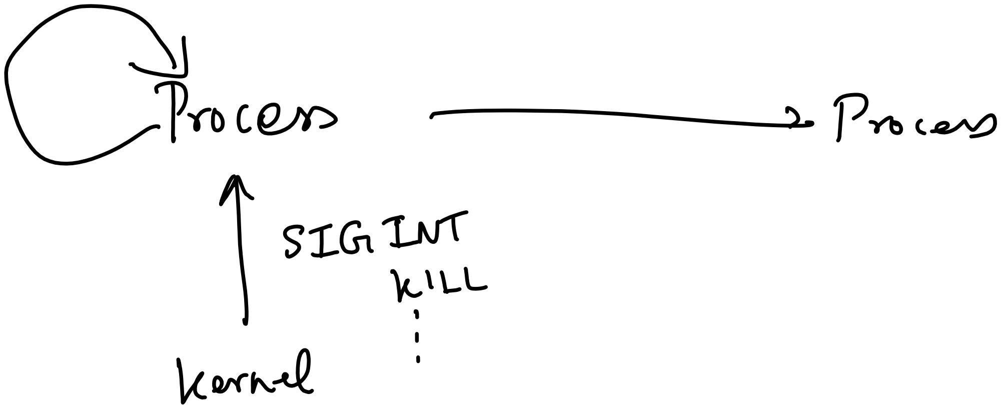
```
id=fork();
if(id==0)
    Child
    printf("I am done");
    exit(0);
else
    Parent
    while(1) {
    }
    wait(&status);
```



defunct => zombie process

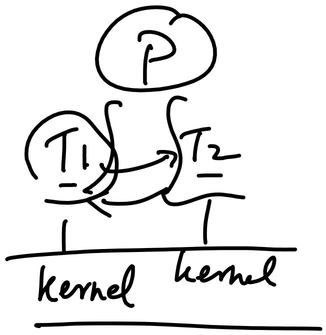
fork bomb

```
while(1)
    fork()
```



Database

Concurrency — threads — T_1, T_2, T_3, ⋮ —



Database

Own thread library

/API

thread Create

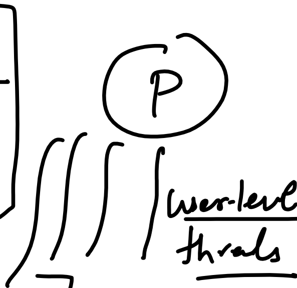
⋮

OS — Thread library

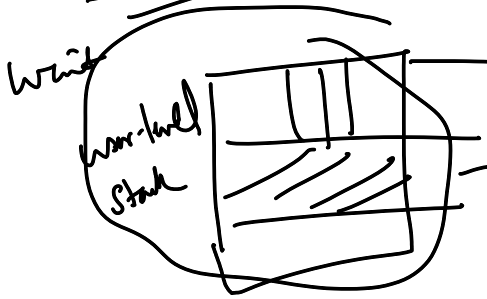
API — thread Create ⋮

Kernel threads

1s 2s



Signals



→ T_1 — syscall read file block

→ T_2

PCB — TCB₁ · — TCB₂

Kernel interrupt stack