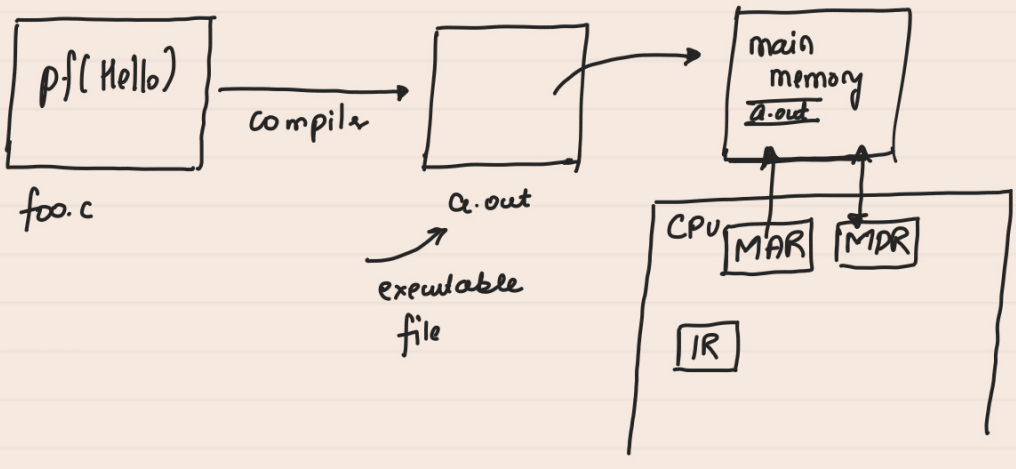
Operating System

Syallabus:

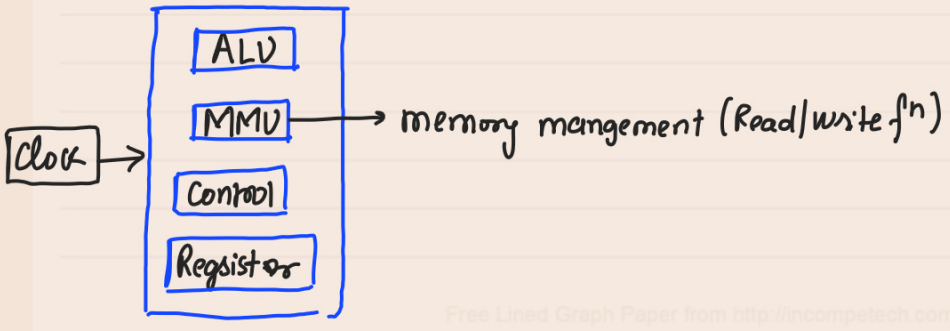
- System calls, processes, threads, intr-praem communication, concurrency and synchronization
- deadlock
- CPU and I/O scheduling
- Memory management and vertual memory
- file systems

lecture 1

pre-requisit: The journey of program (50Mins) COA ∞ OS go together



Inside CPU/processor



CPU [• fetch] ... in loop till the program (set of instructions) gets completed!

- decode
- execute

infinite cycle

Q. Ones you complile, you can run the file ??
Ans: Whenever, you want to run.

running simple program in CPU

```
int a=20, b=30;
int c;
c = a+b
```

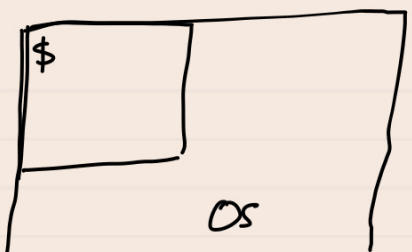
→ PC increment karna hoga manually karna hoga
→ CPU will fetch, load, execute
↓
c ← a+b; (ALU)

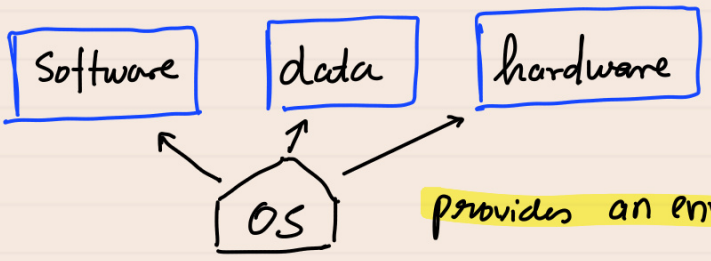
Q. Why "printf" needs OS support!?
→ because CPU doesn't own the monitor
→ monitor is external (called IIo device)
→ CPU has to take permission from monitors

Q. Why need OS?
→ manually too hectic
→ these tasks are very common task
→ even if we ask users to do it manually then first they need to be computer scientist XD.

Q. What OS does?
→ manage Memory
→ manage resources (IIo device)
→ manage file System

What is shell in OS





4 core ≡ 4 CPU

OS- Provids (Virtualization)

decture2

Q. What happens when you switch on the computer?

Q. How does even OS code loaded into memory?

Q. How to create a new procen
↳ how OS load the procen in the main memory?

Q. Is it possible to tweek OS in our personal laptob?

Ans Yes possible

there is one CPU, but it feels like above 2 CPU's are running Simultaneously. CPU Virualization (illusion)

HOW TO CREATE NEW PROCESS

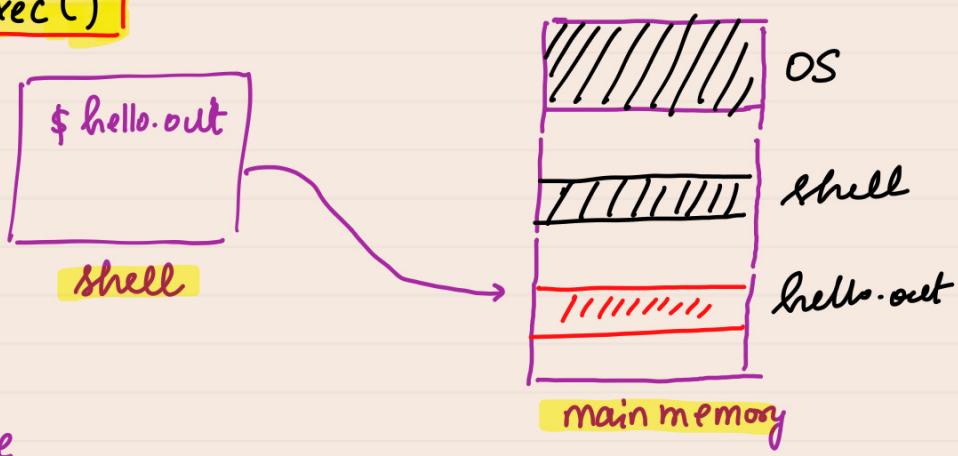
① User writes a program
② Compiler converts the program into executable code (byte code)
③ User requireds the OS to run the byte code.
④ OS "load" the program into memory
⑤ OS sets registrs properly and starts running the code

- OS is always in main memory (loaded by bootloadr at start of computr), but it does not mean that OS is running all the time.

example : printf(" ")
↳ this line can't be executed without the interference of OS !

malloc ()
↳ OS needed !

fork-exec ()



Objective
Understand how these two steps are performed?
① Create Space for hello.out
② Load hello.out in that space.

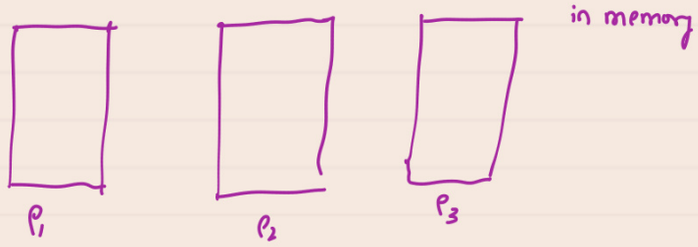
Process
- a program in execution
- process compete for resources
  - context of process
  - Multiple processes at same time

Context of process (pcb)
- every process has context and OS keeps track of that.
- like a school keep tracks of activities of there studies, an OS keep

track of its process.

What all things does OS track about process??

1. PC value
2. general purpose regist
3. file opened et



struct pcb {

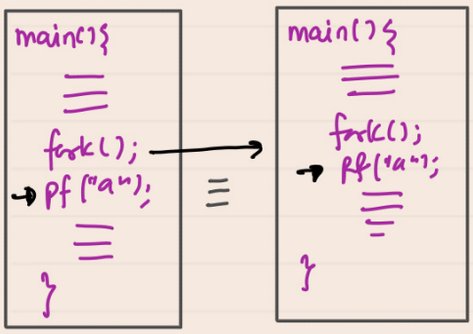
—
—
—
—

}

pcb: a data structure that keep the track of process.

fork() : system call

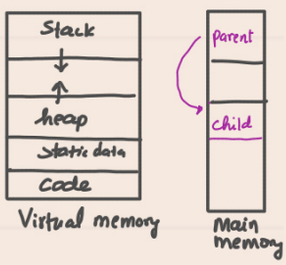
(parent) process 1 | (child) process 2



exact same copy

fork():

1. create EXACT same copy
2. Parent and child will start executing after the line which had fork()
3. fork() will return some integer value to both process
4. fork return "0" to child
5. fork return nonzero to parent
6. fork create exact same copy hence logical address same but physically variables are stored at diffrent location.



- Logical addresn and actual addrn (postponded)

parent

```
main()
{
  int a;
  fork();
  pf(&a);
}
```

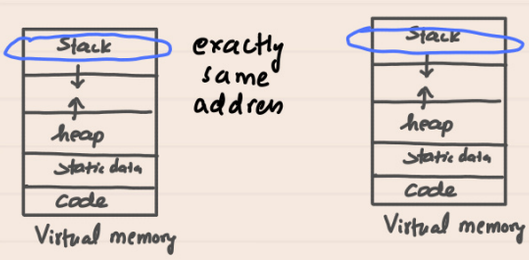
child

```
main()
{
  int a;
  fork();
  pf(&a);
}
```

0xffff001c | 0xffff001c

Q. Will there addrn be same?
Ans: will print exactly same thing!!

M: (Read My Question): Why on earth are we doing so? What are getting by doing this, aren't we using resources wastefully?




```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid = fork();

    if(pid == 0){
        printf("I am child\n");
    }

    else{
        printf("I am parent\n");
    }
    return 0;
}
```

```
(base) sachinmittal@Sachins-MacBook-Pro OSCodes % gcc
(base) sachinmittal@Sachins-MacBook-Pro OSCodes % ./a.
I am parent
I am child
(base) sachinmittal@Sachins-MacBook-Pro OSCodes %
```

Q. how many times each of them will be printed?

```
int main(){
    printf("GATE\n");          — 1 time (ig)
    int pid = fork();
    if(pid==0){
        printf("GATEOverflow!"); — 1 time
    }
    else{
        printf("GO classes\n"); — 1 time
    }
    printf("IISC \n");         ——→ 2 time
}
```

output: { how many patterns / combination we will discuss later }

Q.

```
main()                          parent
{
  int a=5;
  int pid=fork();
  if(pid==0){
      a=a+5;                    } c.s
      pf(a);}
  else{
      a=a+50;                   } c.s
      pf(a);
  }
```

parent will execute

```
main()                          child
{
  int a=5;
  int pid=fork();
  if(pid==0){
      a=a+5;                    } c.s } child will execute this
      pf(a);}
  else{
      a=a+50;                   } c.s
      pf(a);
  }
```

Q. What will child print and what will parent print ??

(10, 60) (60, 55)

child parent child parent

Q. child → parent, order

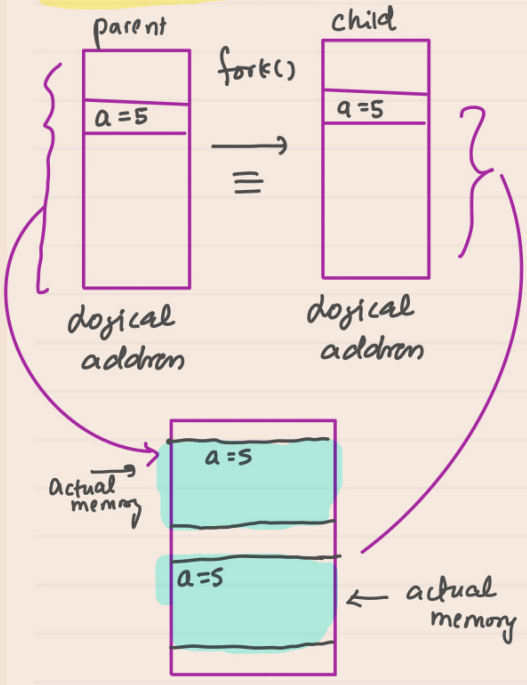
a=5	a=5
$a_c = 10$	$a_c = 10$
$a_p = 55$	$a_p = 60$ (taken a=10;)
(✓)	(✗)

REASON

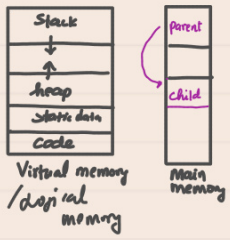
"If doesn't matter what you are doing to child or parent execution order doesn't matter"

Why? how?

- Logical address and actual address (postponded)



"Exactly same logical memory but different actual memory"



MOV R1 1000 → Logical address

- Why "logical" memory (postponded) (logical memory)

a.out → 5 year

GATE

main memory

```
Consider the following code fragment:
if (fork() == 0)
{
   a = a + 5;
   printf("%d, %p n", a, &a);
}
else
{
   a = a - 5;
   printf ("%d, %p n", a,& a);
}
```

u,v - parent process
x,y - child process

child: u = a + 5; x = a - 5; u - 5 = x + 5; u = x + 10

parent: x = a + 5; u = a - 5; x - 5 = u + 5; x = u + 10 (✓) (X)

Wrong!

Same, V=y

Let u, v be the values printed by the parent process and x, y be the values printed by the child process. Which one of the following is TRUE?

A. $u = x + 10$ and $v = y$
B. $u = x + 10$ and $v! = y$
C. $u + 10 = x$ and $v = y$ (correct)
D. $u + 10 = x$ and $v! = y$

even if global variable

do you think Compiler will think

main memory

Where in main memory Space will be free?? NOPE. It logically think there is complete continous memory free

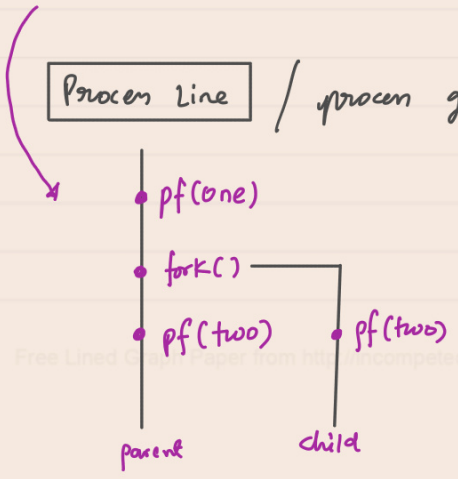
address space / logical memory

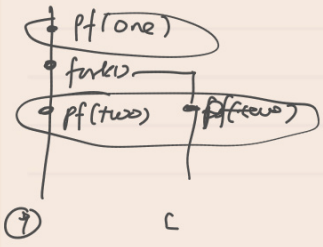
Q. What will be output of the following program?

```
int main(){
    printf("one\n");
    fork();
    printf("two\n");
    return 0;
}
```

Ans: one
two ← parent/child (order!)
two ← child/parent

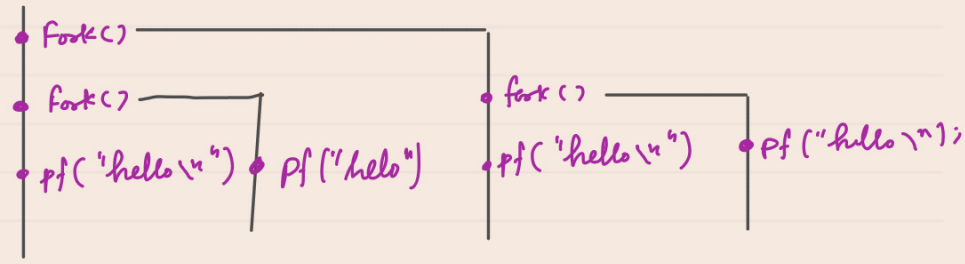
Process Line / process graph





Q. int main(){
    Fork();
    Fork();
    printf("hello\n");
    exist(0);
}
output??

output — process line / process graph



output:
hello
hello
hello
hello

total process: 4

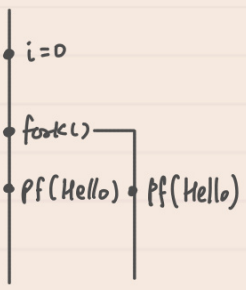
make them from down to up

* Q. how many Hello's printed. (Assume fork does not fail.)

```
int main(){
    int i;
    for(i=0; i<2; i++){
        fork();
        printf("Hello\n");
    }
    return 0;
}
```

a. 2
b. 4
✓c. 6
d. 8

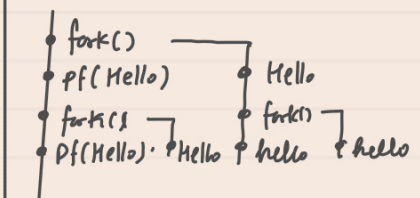
trying:-



Wrong!

output: Hello

since method



Correct

Hello
Hello
Hello

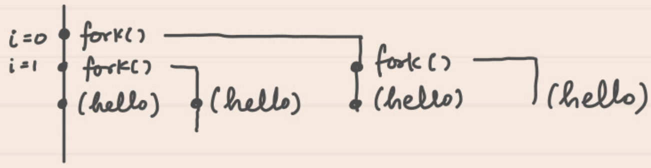
Hello
Hello
Hello
Hello
Hello
Hello

Process: 6

Q. How many Hello's printed? (Assume fork() do not fail)

```
int main(){
    int i;
    for(i=0; i<2; i++){
        fork(); }
    printf("Hello \n");
    return0;
}
```

a. 2
b. 4
c. 6
d. 8

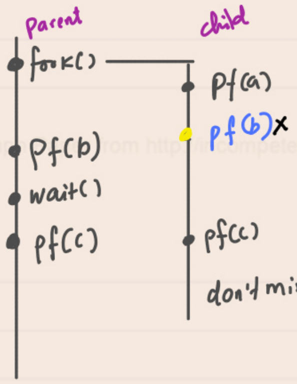


4 times

hello
hello
hello
hello

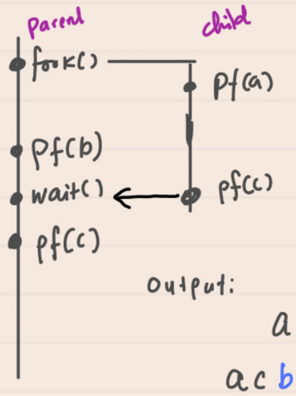
** Q4 Write down all the possible patterns printed by following code

```
int main(){
    if (Fork()==0){
        printf("a");   } child
    }
    else{
        printf("b");   } parent
        wait();
    }
    printf("C");
    exit(0);
}
```



done need to write this?
no! cause if-else statements

don't mix them up



wait(): parent will be waiting till child have completed its process

content switch

Output:

a → c

acbc
abcc
bacc

_a_ c_c

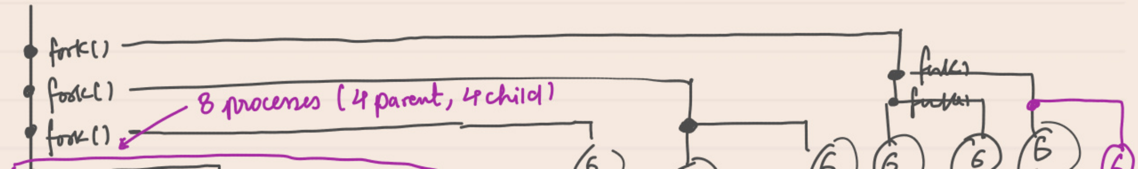
$^3C_1$ = 3 ways

Q.8 How many processes are created with these fork() Statements? HOMEWORK (Wrong!)

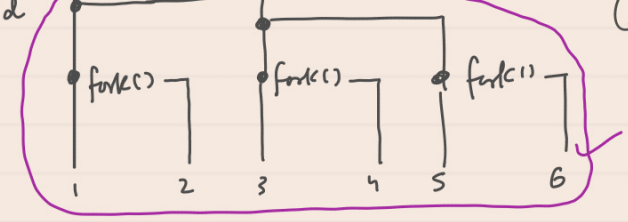
```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(void){
    pid-t pid = fork();
    pid = fork();
    pid = fork();
    if (pid == 0)
```

parent

pid-t pid = fork()



```
    fork();
  }
  fork();
  return 0;
}
```



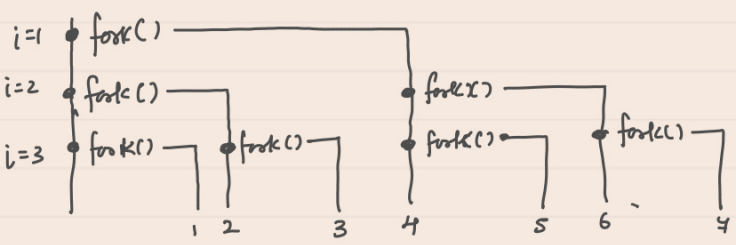
total: $6 \times 7 = 42$ process

Lecture 3

Q. How many child process will create?

```
for(i=1; i<=3; i++)
{
  fork();
}
```

Trying



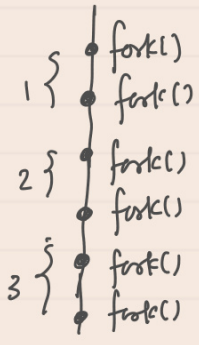
Ans: 7 child processes
1 parent process
Total process: 8

"fork-tree"

2^n process, n = no. of fork()

Q. How many child process will create?

```
for(i=1; i<=3; i++){
    fork();
    fork();
}
```



$2^6 = 2^2 \times 2^2 \times 2^2$
$4 \times 4 \times 4$
$16 \times 4 = 64$

No. of child process: 63

Q. `for(i=0; i<n; i++) fork();`
Total no. of child process?

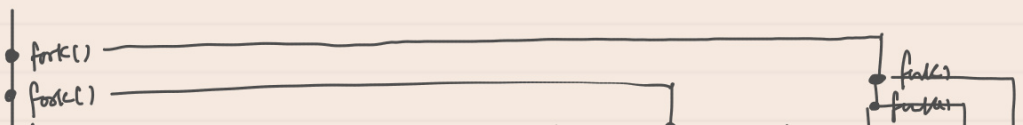
"n times" → 2^n process

$2^n - 1$ child process

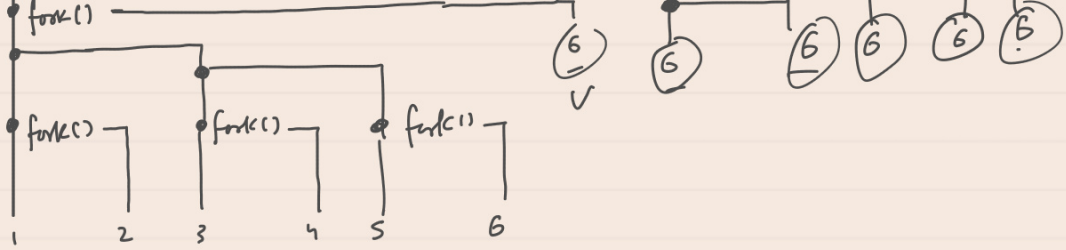
Q.8 How many process are created with these fork() statements? HOMEWORK (Wrong!)

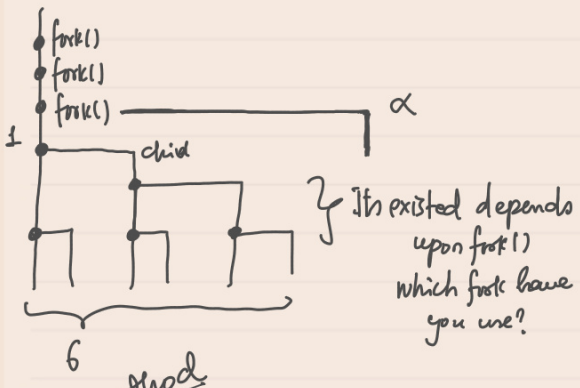
```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(void){
  pid_t pid = fork();
  pid = fork();
```

parent `pid_t pid = fork()`

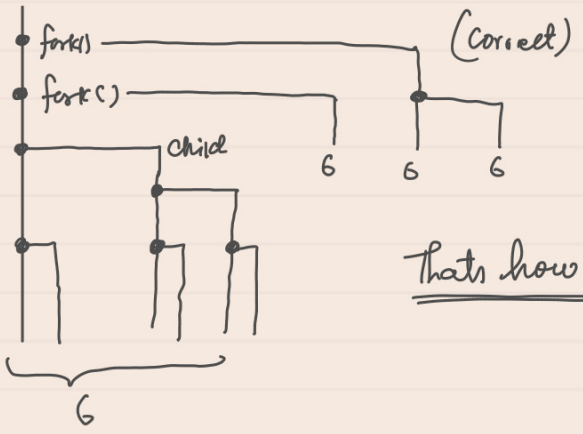


```
    pid = fork();
    if (pid == 0)
    {
        fork();
    } } child
    fork();
}
```





} Its existed depends upon fork() which fork have you use?



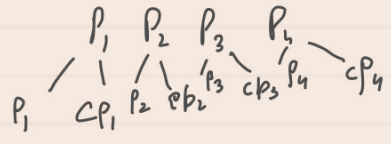
That's how 24

Sir's method

```
int main(void){
    pid_t pid = fork();
    pid = fork();
    pid = fork();
    if (pid == 0)
    {
        fork();
    }
    fork();
    return 0;
}
```

how many processes? = 4 $P_1\ P_2\ P_3\ P_4$

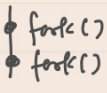
how many processes? = 8



$P_1\ P_2\ P_3\ P_4$, $c(P_1\ P_2\ P_3\ P_4)$ → zero pid

non-zero pid

$4 \times 2 = 8$



$2^2 = 4$

$4 \times 4 = 16$

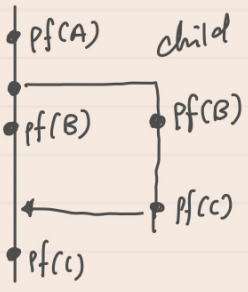
$\therefore$ Total = $8 + 16 = 24$

Q. Consider the following example program. List all legal outputs this program may produce when executed on a unix system. The output consists of strings made up of multiple letters.

```
#include <unistd.h>
#include <sys/wait.h>
// W(A) means write(1, "A", sizeof "A")
#define W(x) write(1, #x, sizeof #x)
int main()
{
    W(A);
    int child = fork();
    W(B);
    if (child)
        wait(NULL);
    W(C);
}
```

pf(A)

pf(B)



A _ B _ C _ C

A B B C C

A B B c c

A C B C

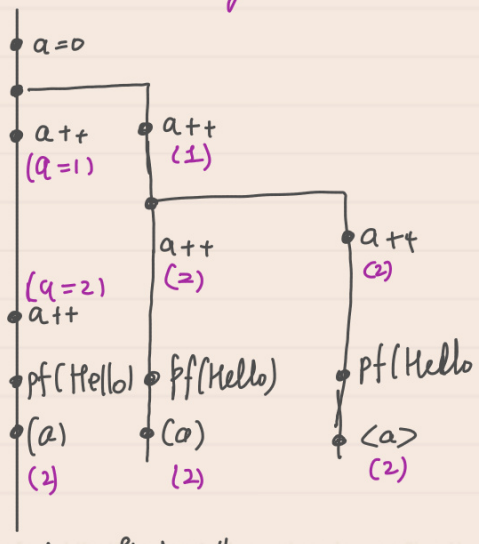
$\therefore$ total 2 patterns!

if (non-zero) ≡ TRUE

if (zero) ≡ false

Q. how many times does the following program print "Hello!"? also value of a?

```
int a=0;
int rc = fork();
a++;
if (rc==0) {
        rc = fork();
        a++;
}
else {
        a++;
}
printf("Hello!\n");
printf("a is %d\n", a);
```



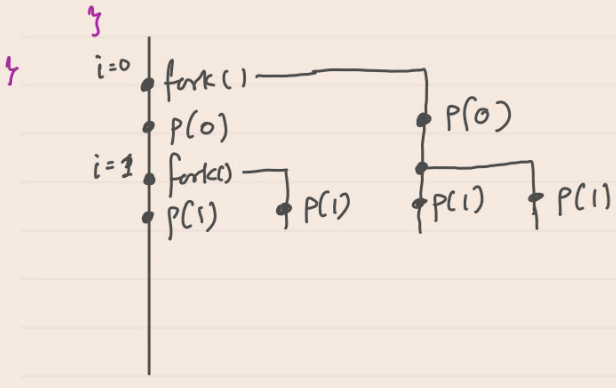
note: $1 doesn't

Ans 3 times "Hello"!

2 max value?

Q.
```
int main() {
    int i = 0;
    while (i<2) {
        fork();
        printf("%d", i);
        i = i+1;
    }
}
```

X A: 0 1 0 1
✓ B: 0 0 1 1 1 1
✓ C: 0 1 0 1 1 1
X D: 0 1 1 1 0 1 → need to access 1's
✓ E: 0 1 1 0 1 1



Q.
```
int main()
{
    int c=0;
    for(int i=0; i<3; i++) {
        if(fork()==0) {
            printf("%d", c+i);
            fflush(stdout);
        }else {
            c = c+i+3;
        }
    }
    return 0;
}
```

(H·W)

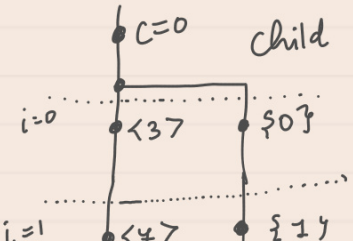
ⓐ how many times fork() involved?

ⓑ correct Outputs

a. 0 1 4 6 5 9 2
b. 0 1 5 4 9 6 2
c. 0 4 2 6 1 5 2
d. 9 4 0 5 1 2 6
e. 4 0 1 6 9 2 5
f. 6 4 9 5 0 1 2

(Later)

(need c)



attempted but unable to understand

EXEC SYSTEM CALL

exec( ) → parameter → any executable file

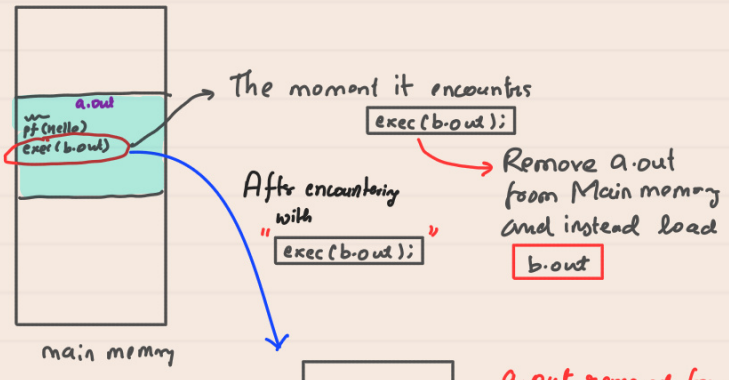
Suppose

a.out

```
int main(){
    ~~~
    pf(Hi Adib);
    ~~~
    exec(b.out);
    ~~~
    pf(Adib failed);
    ~~~
}
```

b.out

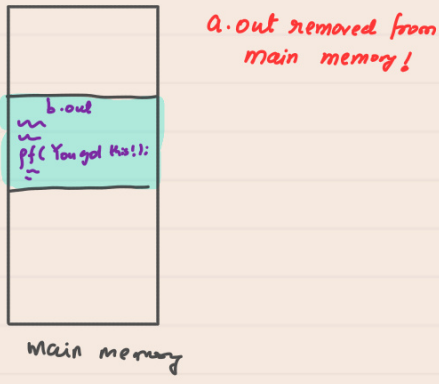
```
int main(){
    ~~~
    pf(You got this!);
}
```



What is the output if we ran a.out file ??

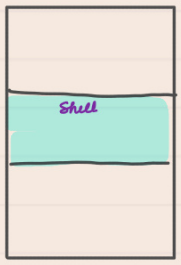
Hi Adib
You got this!
(✓)

Hi Adib
You got this!
Adib failed → never printed!!
(✗)



a.out removed from main memory!

Process Creation



① Suppose, shell is running in main memory, and in cmd you entered ./a.out, then what?

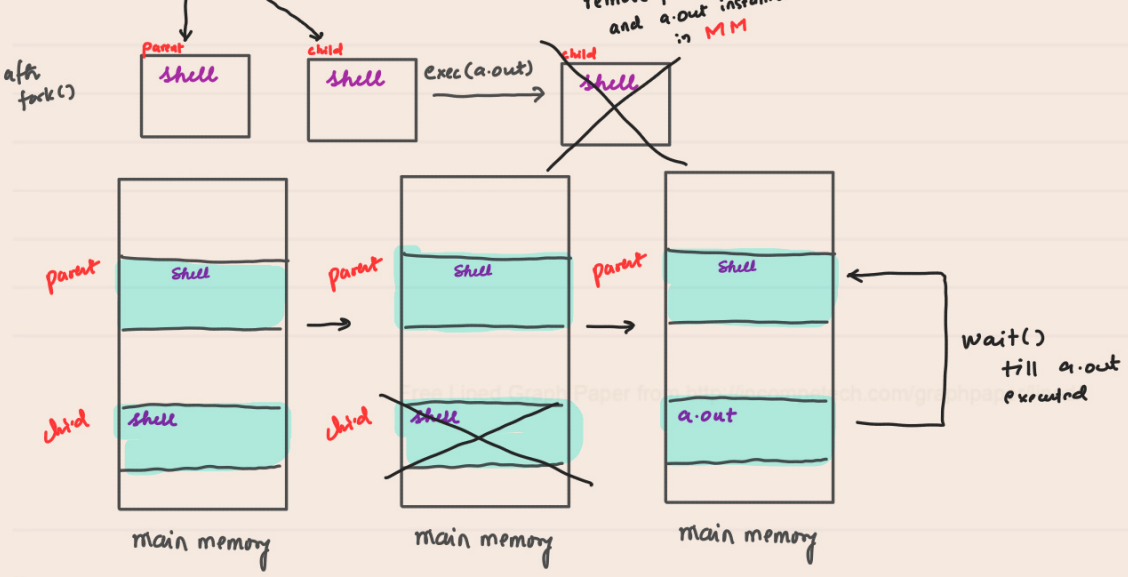
Shell code (somewhat)

```
While (TRUE){
    scanf("%s", &t);        // Some command
    pid = fork()
    if (pid == 0)           // child
        exec(t);
    else
        wait();
}
```

(Not the only, UNIX does this way, other OS may do other way too!)

(discussig Idea!)

remove from MM

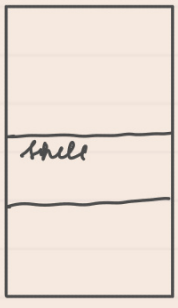


You write a UNIX shell, but instead of calling fork() then exec() to launch a new job, you instead insert a subtle difference: the code first calls exec() and then calls fork() like the following:

```
shell (..) {
  .. ..
  exec (cmd, args);
  fork();
  .. ..
}
```

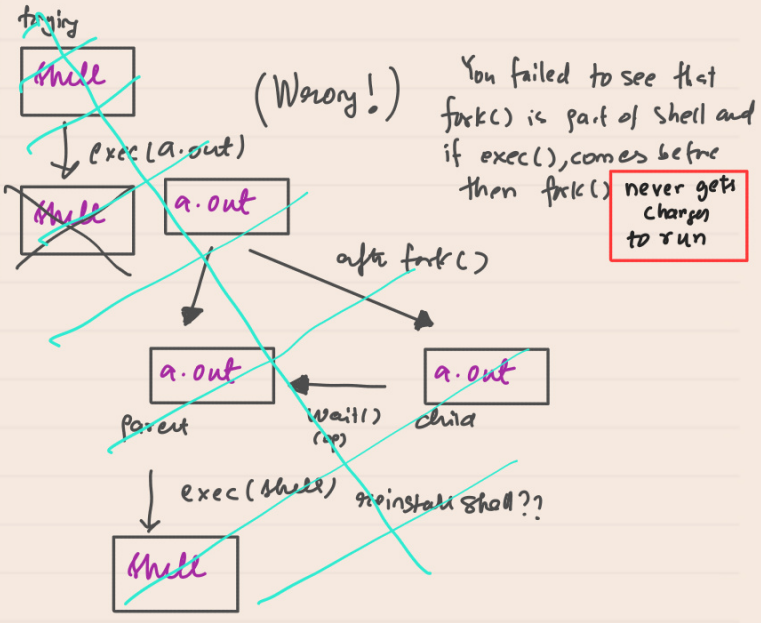
Does it work? What is the impact of this change to the shell, if any? (Explain)

→ Means working just like "earlier"??



Who creates Shell inside main memory??

init (Bootload ke time) of Shell



[process tree of unix]

↳ data

(Correct)

Ans: No.

→ we are destroying the shell.

→ we are just replacing shell with a.out

→ fork() never gets executed!!

Part (a) [3 MARKS] whatA

```
int main() {
        int i = 0;
        printf("%d ", i);
        i = i + 1;
        if (i >= 2)
                exit(0);
        execv("./whatA");
}
```

A: 0 1 2
B: 0 0 0
C: 0 0 0 0 0 0 0 ... (forever) (✓)
D: 0 1 2 0 1 2 ... (forever)
E: none of the above

initially "0" printed, i = 1 (updated)
as i < 2 ∴ goes to next line exec(whatA),
which means that it will load WhatA in place
of WhatA, and again i=0, "0" printed

∴

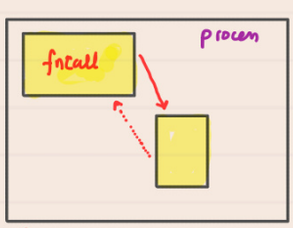
0 0 0 0 0 . . . (forever)

System Calls

>What is System call?
>Why they are always in limelight?

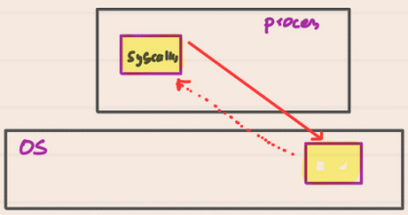
function call vs System call

| millions of lines of OS code | OS code |

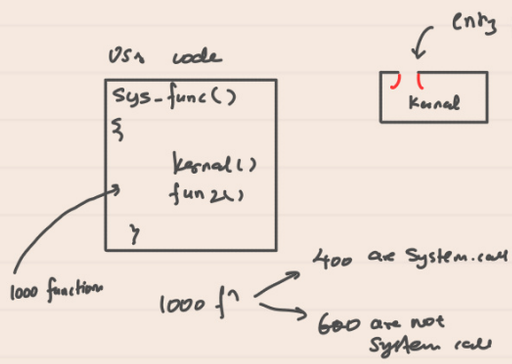


function call

Caller and callee are in the same process
- Same user
- Same "domain of trust"

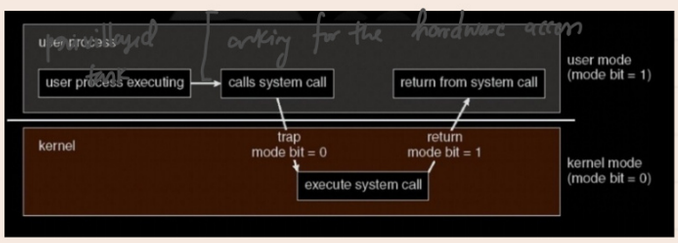


- OS is trusted. User is not
- OS have supa priviles. User Not
- must take measure to prevent abuse



- normal task [a = b+c ; ← can be done without permission of OS

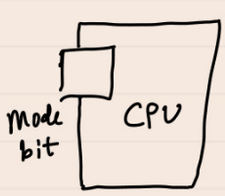
Q. how user ask for OS service?
- using system call



User Trust Issue

- Most of the users are not experts in computers and do not understand how hardware works. Thus they may not know how to properly manage the resources.
- An user (expert or not) can make incorrect operations while managing the resources (like handling h/w)
- An user can be malicious i.e. he/she can deliberately make incorrect operations. This can lead to system failure.

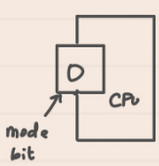
Solution from OS side: Never trust a user while dealing with critical tasks



Suppose user wants to access both

User mode — Normal task [a = a+b-c ; mode bit = 1

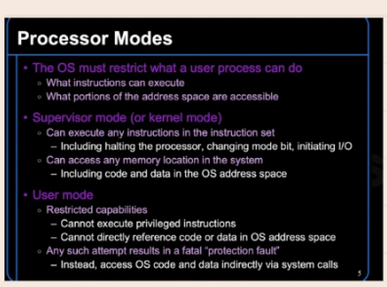
Kernal mode / privilaged mode — Privilated task [access some hardware ; mode bit = 0



User Mode
- Compute operations like addition, Sub, etc
- Reading and writing from memory

Kernal mode
- CPU and memory management
- I/O handling

Using System calls



API (Application programming interface)

User processes usually do not directly involve system calls. Instead they use something called an **Application Programming Interface (API)**. It is set of functions, variable and other software construction that allows to perform the most needed operations and also to cooperate with the operating system

Q. System calls must be used to

- ✗ (a) modify global variable
- ✗ (b) call a user-written fn
- ✓ (c) write a file
- ✗ (d) All of the above

Lecture

Q. What is a System call?

Various possible answers

request to kernal to perform a service

- Open file
- Execute a new file
- Create a new process
- Send a message to another process

(Programmer Perspective) "call a function"

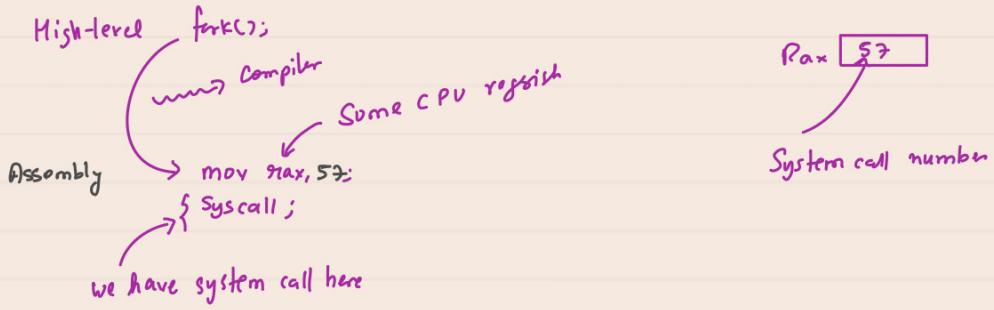
- fd = open("myfile.txt");
- System call looks like any other function call

entry point providing controlled mechanism to execute kernal code

System call execution

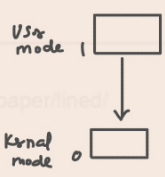
- Each system call has unique numeric identifier
- OS has a system call table that maps number to functionality required.

eg:



- When invoking a systemcall, user places system call number and associated parameter in a specified register, then execute the syscall instruction.

1. Progam calls Wrapper function in C Library
2. Wrapper function
   - package syscall arguements into register / Stacks
   - Puts (unique) syscall number into a register
3. Wrapper flips CPU to Kernal Mode (User-mode → Kernal-mode)
   - Execute special machine instruction (eg Syscall)



- Main effect: CPU now can touch memory marked as accessible only in kernal mode

④ Kernal execute Syscall handler:

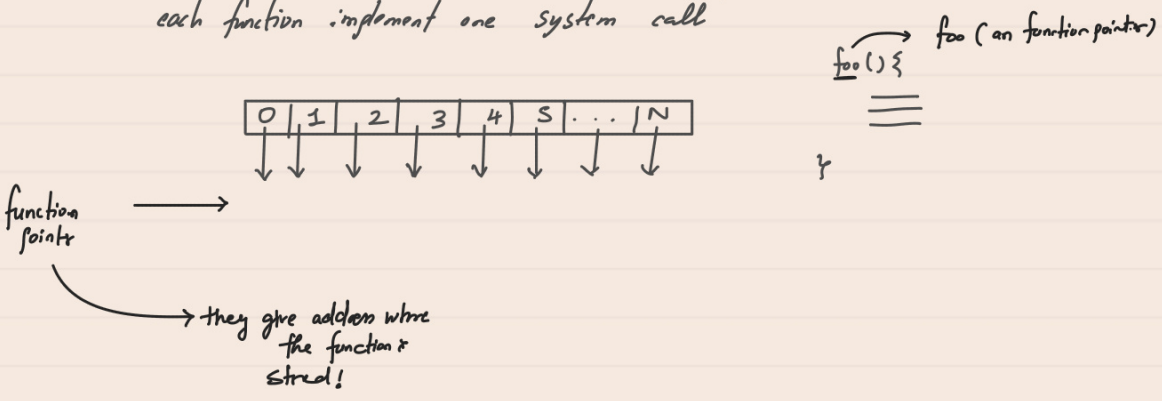
invokes service routine, corresponding to syscall number

- Do the real work, generate result status

- Places return values from service routine in a register
- switch back to user mode, passing control back to wrapper
  - (Kernal mode ⟶ user-mode)

System Call table

- Syscall table is usually implemented as an **array of function pointer**, where each function implement one system call



Q. Whenever we see "Syscall" instruction then what are the steps??

Q. True or false? System calls are as fast as function calls? Explain briefly??

False. System calls have to store syscall (unique) number and appropiate arguements then switch mode bit then travels through arrays of function pointr, then go to address execute and return value and switch back. takes hundreds of cycle. Whereas (assume) functioncal, just need to follow function point which is available in user mode address only.

Types of System call

COA interpet Q's ⟷ Syscall
overlap!!

process control
- create process, teriminate process
- end, abort
- load, execute
- get process attributes, set process attributes

file management
- create file, delete file
- open, close file
- read, write

Device management
- request device, release device

information maintaince
- get time or date, set time or date

communication

protection

Threads

Sometimes, a sigle application may required to perform similar tasks, in such cases we don't want to have several process.

Why?
* context switching
* diffrent address space

Solution? Threads.

They share! unlike diffrent process they want everything for themselves!

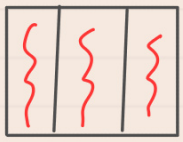
example: word process → spell checker, write something

for similar tasks, if we have diffrent processes (instead of threads) then
- IPC (interprocess communication) is required

Types of thread process

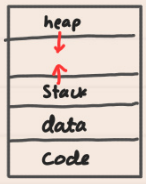
- process context switching is expensive than thread context switching

Single threaded process



multithread process

Q. What do (things) process have??

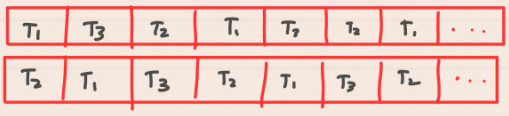


content
→ might have opened some file
→ Some register value PC, R_1, R_3

Concurrency :

"concurrent"

parallel (need) >1 CPU

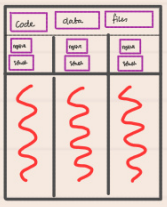


"parallel"

Q. Why do need to create single thread?

It could be you want just perform one similar task, 1 Process 1 thread ??

single threaded process



multithread process

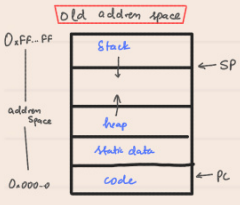
WAIT! WAIT! threads are like process they run individually! in single CPU! so Now question?

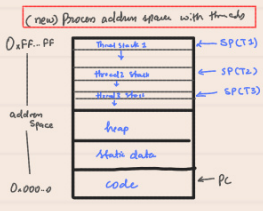
What can you can't share??

Stack

register values

What abt heap?
What abt it??
its space occupied at runtime!

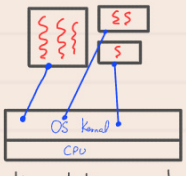




Types of thread

User level

- When threads are created using function call
- have TCB (Thread Control Block)

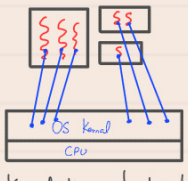


Kernal knows only three process. Kernal don't know how many threads inside process they are

OS will schedule the process if doesn't know about threads, hence effect

Kernal Level

- When threads are created using System call
- have TCB (Thread Control Block)



Kernal know about internal threads.

OS could Schedule threads. More effective use of hardware.

} Threads in User level

Can we take help from kernal scheduler to schedule the threads??

⇒ No

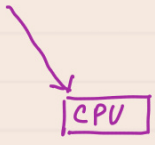
> i(user) might be having my own preference to schedule

Q. I can not take help from OS scheduler then how am I supposed to schedule user level threads??

Ans: Thread schedular implemented in user space. (Repeative task)
we have thread libraries, which has everything

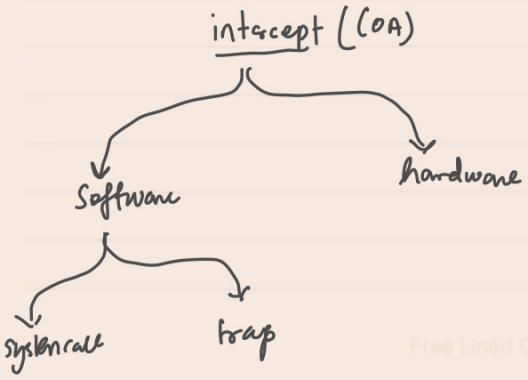
Q. Can user level threads run kernal code?
Ans: Yes, we can definitately have systemcall

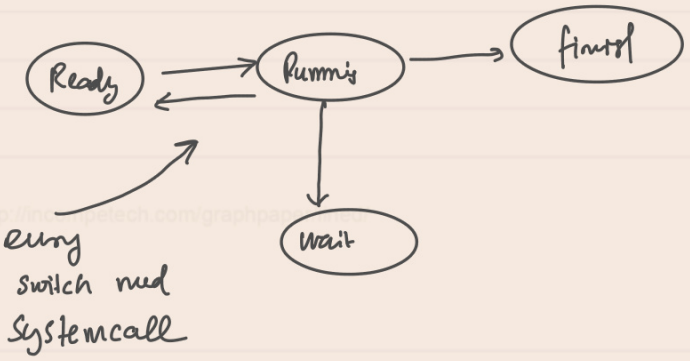
Q. P_1 P_2 P_3 P_4



> P_1 is running
> we want to schedule other procen now
> we will first run "scheduler" how to schedule, scheduler itself

There will some interept
Sched ← instruction





Q. What is one major advantages and disadvantage of User level thread?

① User flexibility

- fast to create (No systemcall needed)
- Kernal doesn't have to maintain diffrent threads. Hence context switching not

② efficiency could take hit

* if one of the threads is block then all the threads gets block, whole procen gets blocked

needed

T/F

Q1) The OS provide the illusion to each thread that it has its own address space?

Ans: false, not completely.

Q2) With Kernal threads, multiple threads from the same process can be scheduled on multiple CPUs simultaneously?

Ans: True

Q3) If a user level thread is blocked for I/O operation, then Kernal of OS will perform context switching to run another user-level-thread which is not blocked??

Ans: False

Q. 2. **Processes and threads.** Suppose that three processes exist in a system, as described table below. Suppose that the system uses preemptive, round-robin scheduling, and that T_{11} is running when the quantum expires.

Process	Threads within the process
P_1	T_{11}, T_{12}, T_{13}
P_2	T_{21}, T_{22}
P_3	T_{31}

(a) (4) If the threads are implemented entirely at the user level (with no support from the operating system), which threads might possibly execute at the beginning of the next quantum?
T_{21}, T_{22} or T_{31} The OS is unaware of the threads within process P_1

(b) (4) If threads are supported by the operating system (i.e., lightweight processes), which threads might possibly execute at the beginning of the next quantum?
any except T_{11}

(c) (2) How does your answer change if the system has two processors?
It actually doesn't change. In the first case, the kernel will not know about the user's threads, so no other choices can be made. In the second case, the kernel will have full knowledge of the user's threads and will make the same decision (of course, realizing that the OS should not run the same thread on simultaneously on multiple processors).

10:25 am ✓

(a) T_{21}, T_{22}, T_{31}

(b) any except T_{11}

(c) if all user level
Then
T_{31} will run
and
either T_{21} or T_{22}

if kernal level
except T_{11}
any other two threads will run

Q. A blocking kernal-scheduled thread blocks all the thread in the process – False
Q. Threads are cheaper to context switch than process – True
Q. A blocking user-level thread blocks the process – True

Indicate which statements are **True**, and which are **False**, regarding a single process and its threads:

38. In an operating system environment that operates with User-Level threads only (no Kernel-Level threads), the various threads associated with a single process all share a common context, including a common PC. — False
39. In an operating system environment that operates with Kernel-Level threads only (no User-Level threads), the various threads associated with a single process all share a common context, including a common PC. — False
40. The different threads associated with a single process all share a single processor context., *i.e.*, the value of the Processor Status Word is the same for all such threads. — ??
41. Thread creation takes more time than process creation. — False
42. Each thread associated with a single process has exclusive ownership of any files that it has opened; they are not available to other threads associated with the same process. — False
43. Each thread associated with a single process has a separate text (program code) area. — False

10:34 am ✓

HOMEWORK (GATE Questions)

read question correctly!

Q. **(4 marks)**
Suppose that two long running processes, P_1 and P_2, are running in a system. Neither program performs any system calls that might cause it to block, and there are no other processes in the system. P_1 has 2 threads and P_2 has 1 thread.
(i) What percentage of CPU time will P_1 get if the threads are *kernel threads*? Explain your answer.
(ii) What percentage of CPU time will P_1 get if the threads are *user threads*? Explain your answer.
Sample Answer:

(i) If the threads are kernel threads, they are independently scheduled and each of the three threads will get a share of the CPU. Thus, the 2 threads of P_1 will get 2/3 of the CPU time. That is, P_1 will get 66% of the CPU.
(ii) If the threads are user threads, the threads of each process map to one kernel thread, so each *process* will get a share of the CPU. The kernel is unaware that P_1 has two threads. Thus, P_1 will get 50% of the CPU.

5:17 am ✓

Trying

P_1 P_2
{{ {

(No syscalls that would block)
(No other process)

(i) 1 CPU, assume round robin
then $\frac{1}{3}$ cause CPU knows that

proc 1 has 2 threads and it can schedule them

(II) $1/2 \cdot (1/2)$ ← in 1 proc P_1 has that chance

↑ 1 proc

$\frac{2}{3}$

$1/4$

$1/2$

GATE CSE 2004

Consider the following statements with respect to user-level threads and kernel-supported threads

44

I. context switch is faster with kernel-supported threads

II. for user-level threads, a system call can block the entire process

III. Kernel supported threads can be scheduled independently

IV. User level threads are transparent to the kernel

Which of the above statements are true?

A. (II), (III) and (IV) only
B. (II) and (III) only
C. (I) and (III) only
D. (I) and (II) only

5:22 am ✓

(I) F
(II) T
(III) T
(IV) T

Ans: (A)

transparent: invisible

GATE 2007

Consider the following statements about user level threads and kernel level threads. Which one of the following statements is FALSE?

32

A. Context switch time is longer for kernel level threads than for user level threads.

B. User level threads do not need any hardware support.

C. Related kernel level threads can be scheduled on different processors in a multi-processor system.

D. Blocking one kernel level thread blocks all related threads.

A. T
B. F, User level thread can call System call as Needed
C. T
D. F, in the case with user level threads

Ans: (B), (D)

B - True, User level threads don't need any hardware support.

Kernal level thread - GOD - implemented and recognised by OS
Require Hardware support
Independent
More context switching time (b/c more attributes
hard implementation

Usual level thread - HUMAN

USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
Example : Java thread, POSIX threads.	Example : Window Solaris.

5:37 am

GATE CSE 2011

A thread is usually defined as a light weight process because an Operating System (OS) maintains smaller data structure for a thread than for a process. In relation to this, which of the following statement is correct?

A. OS maintains only scheduling and accounting information for each thread
B. OS maintains only CPU registers for each thread
C. OS does not maintain virtual memory state for each thread
D. OS does not maintain a separate stack for each thread

A. Thread (stack, reg) (heap, data, code)

B. False,

C. TRUE,  inside a process there is thread so maybe

D. False

OS, on per thread basis, maintain only two things → CPU registers, Stack Space

Even TLB and paging tables are also shared since they belong to same process
↓ Scheduling help ↓ Accounting (unable to understand)

GATE 2014

Which one of the following is **FALSE**?

A. User level threads are not scheduled by the kernel.
B. When a user level thread is blocked, all other threads of its process are blocked.
C. Context switching between user level threads is faster than context switching between kernel level threads.
D. Kernel level threads cannot share the code segment.

A. True, kernal level threads are scheduled by kernal

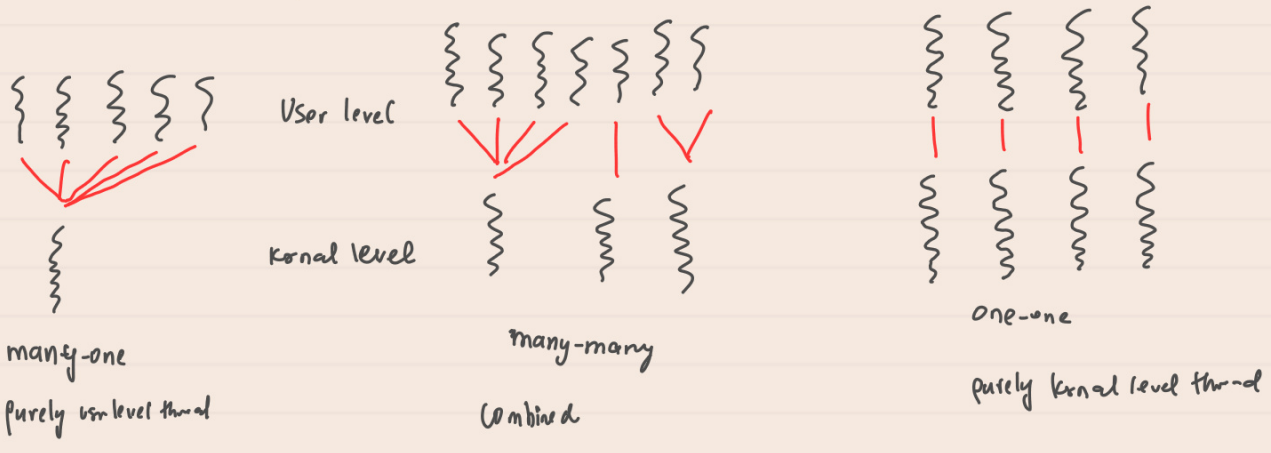
B. True

C. True

D. False, be it user level or kernal level, threads share code, heap, data with common thread of self process.

Multithreading Models

- many-one
- one to one
- many to many



Q.  suppose: we want to run U3 even if U1 is blocked then

U1 U2 U3 U4

④ We will map U1, U3 to one Kernal level thread

⑤ we will map U1 U3 to diffrent Kernal level thread

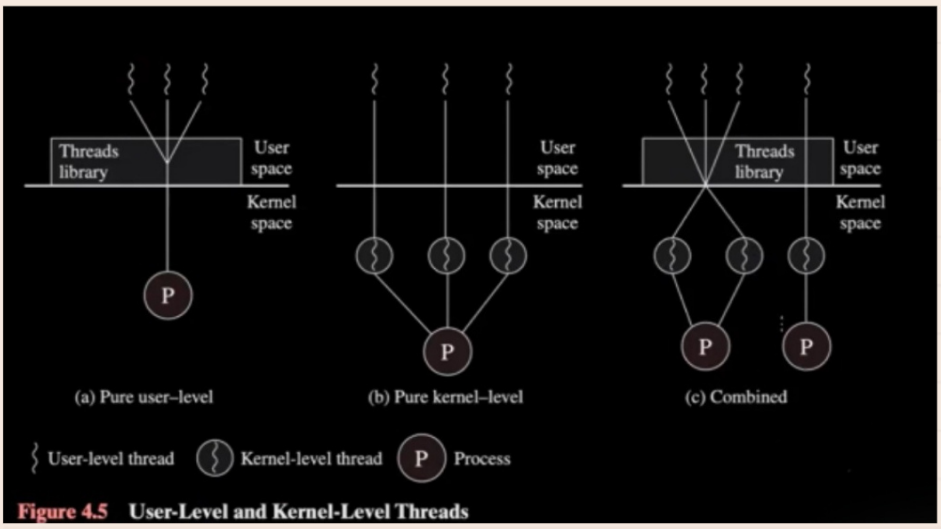
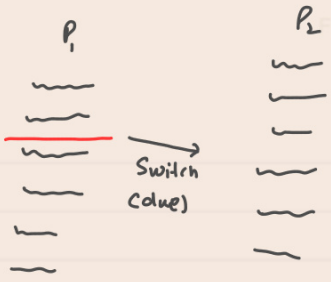


Figure 4.5 User-Level and Kernel-Level Threads

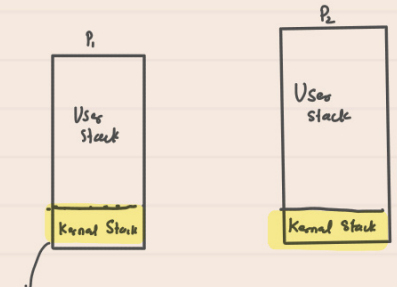
CONTENT SWITCHING

due to any reason!



Content Switching

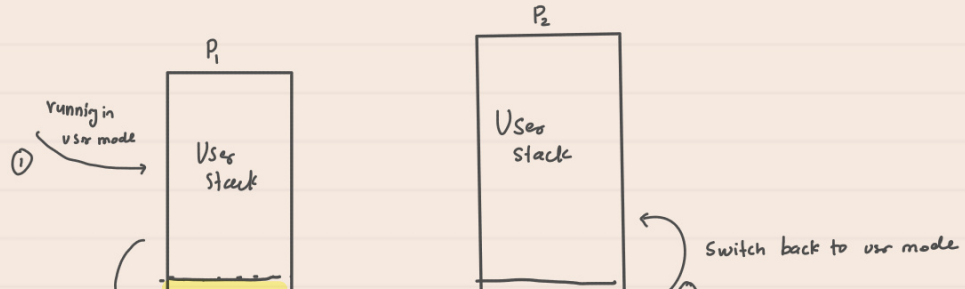
- Saves state of a process before a switching to another process
- Restore original process state when switching back

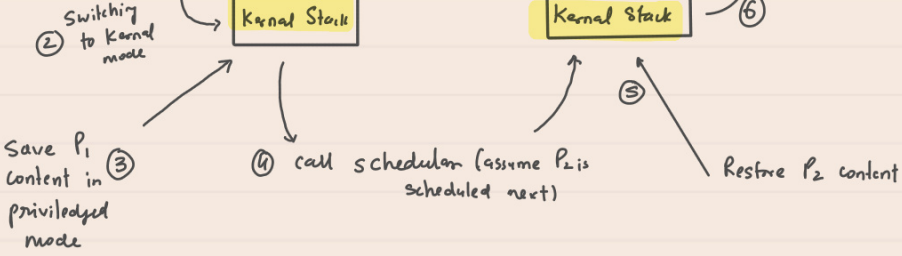


We can push the register value here at the time of content switching

Why are we using Stack data Structure?

Cause stack isn't fix, it can grow and shorten itself realtime depending upon the required unlike fixed array size allocated for storing reg. value.





P_1, usr mode

P_1, kernal mode (I want to save P_1 content in privileged area)

Scheduler

P_2, kernal mode (I want to restore P_2 context)

P_2, usr mode

Q. Context switching only happens when process is already in kernal mode

Ans: True, you can't directly switch from P_1(usr mode) to P_2

(Thread questions)