CS & IT ENGINEERING

Operating Systems

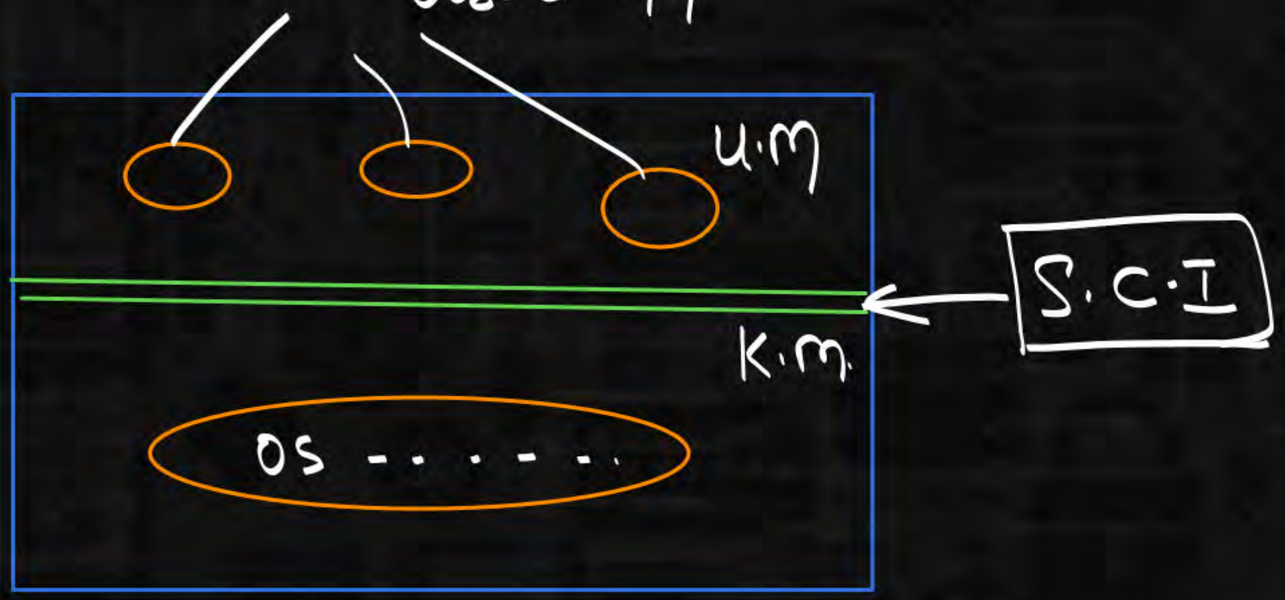
System Calls & threads

Lecture No. 1

By- Dr. Khaleel Khan Sir

- System Call Implementation
- Fork System Call

System Calls : is a means of availing O.S services;




```
main()
{
    f();      User defined
    Scanf();  Predefined
    fork();   OS defined
}
```

UM

KM

Privileged Instn | S/w Int. Inst.

System Calls

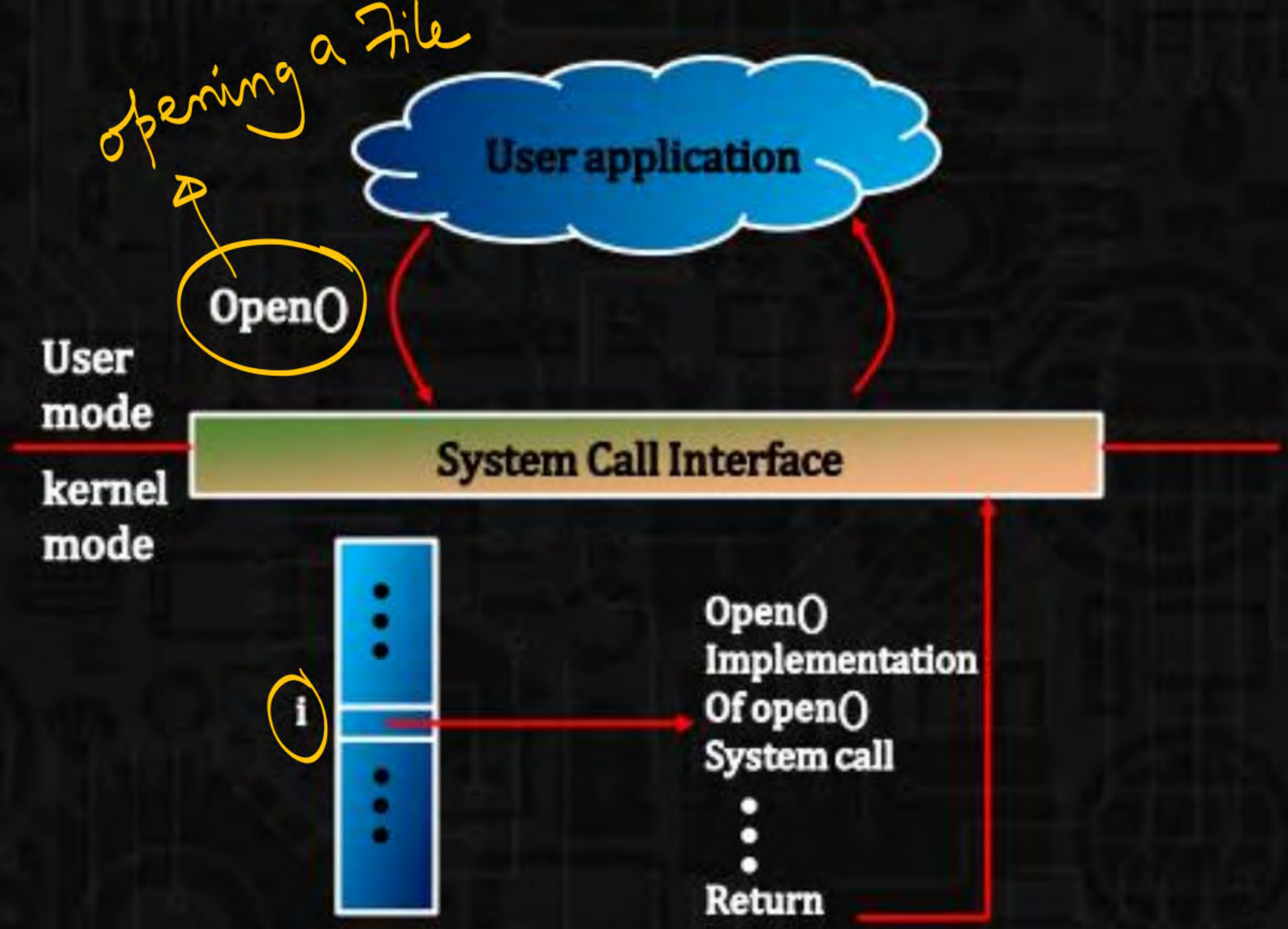
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

  (Note that the system-call names used throughout this text are generic)

System Call Implementation

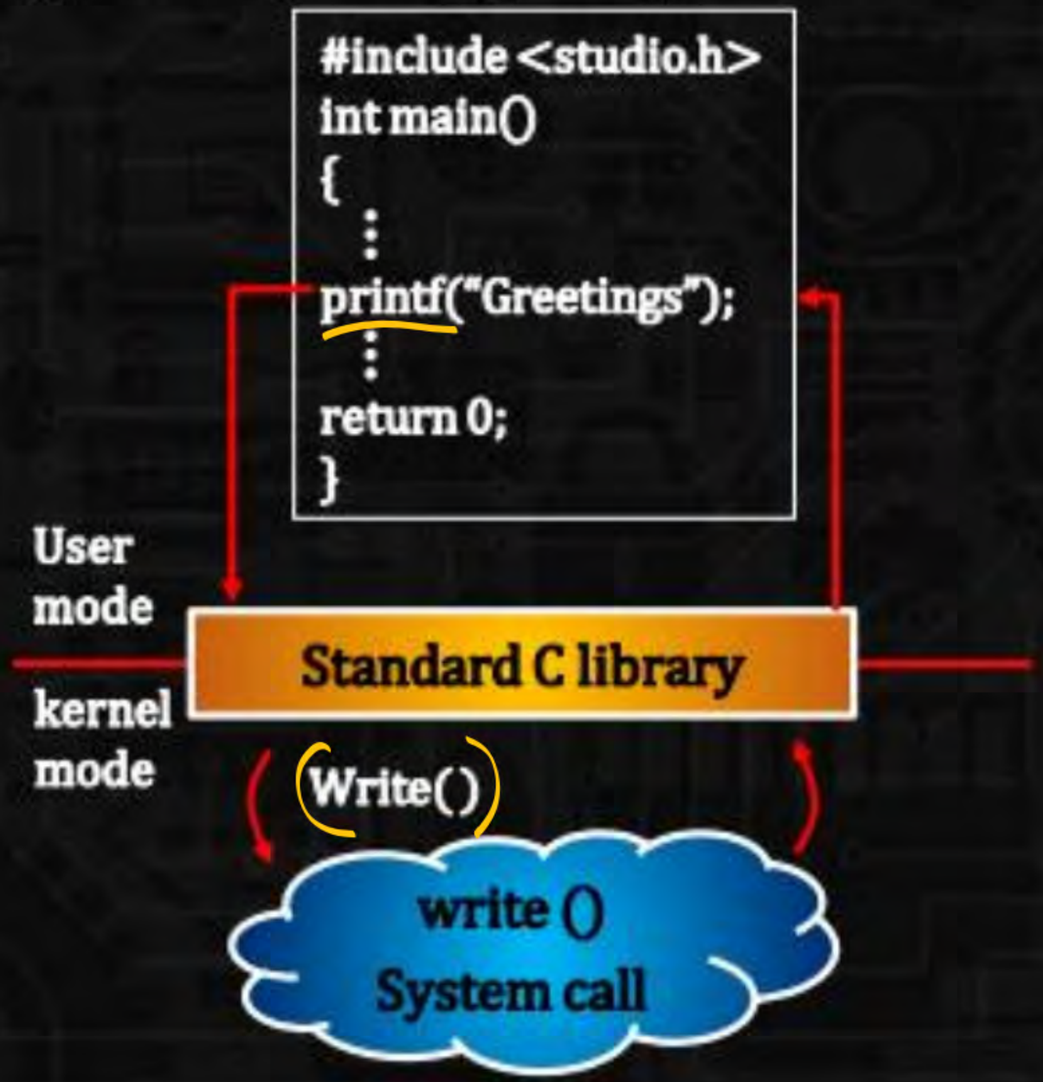
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
- Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship

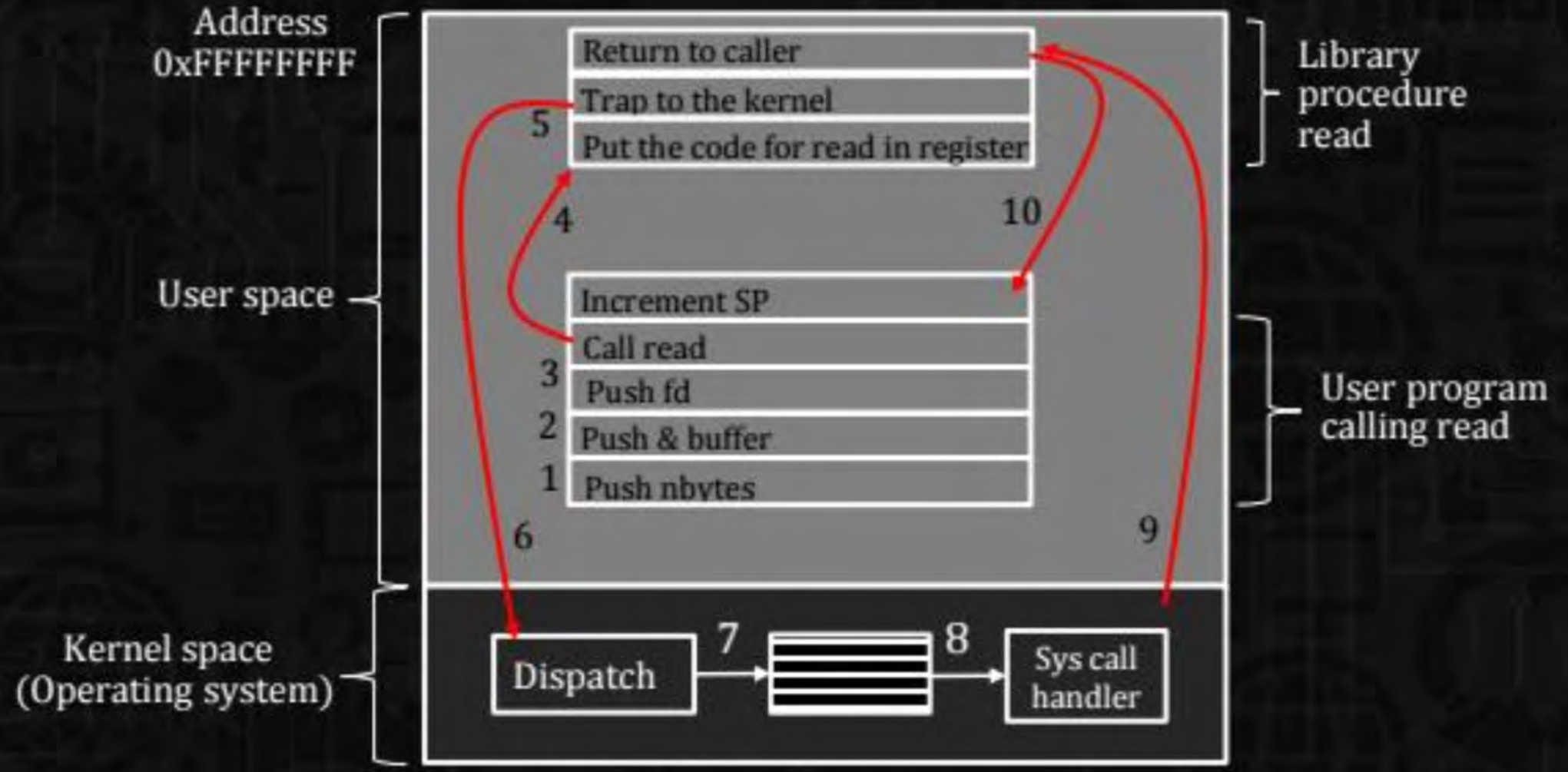


Standard C Library Example

- C program invoking printf() library call, which calls write() system call



Steps in Making a System Call



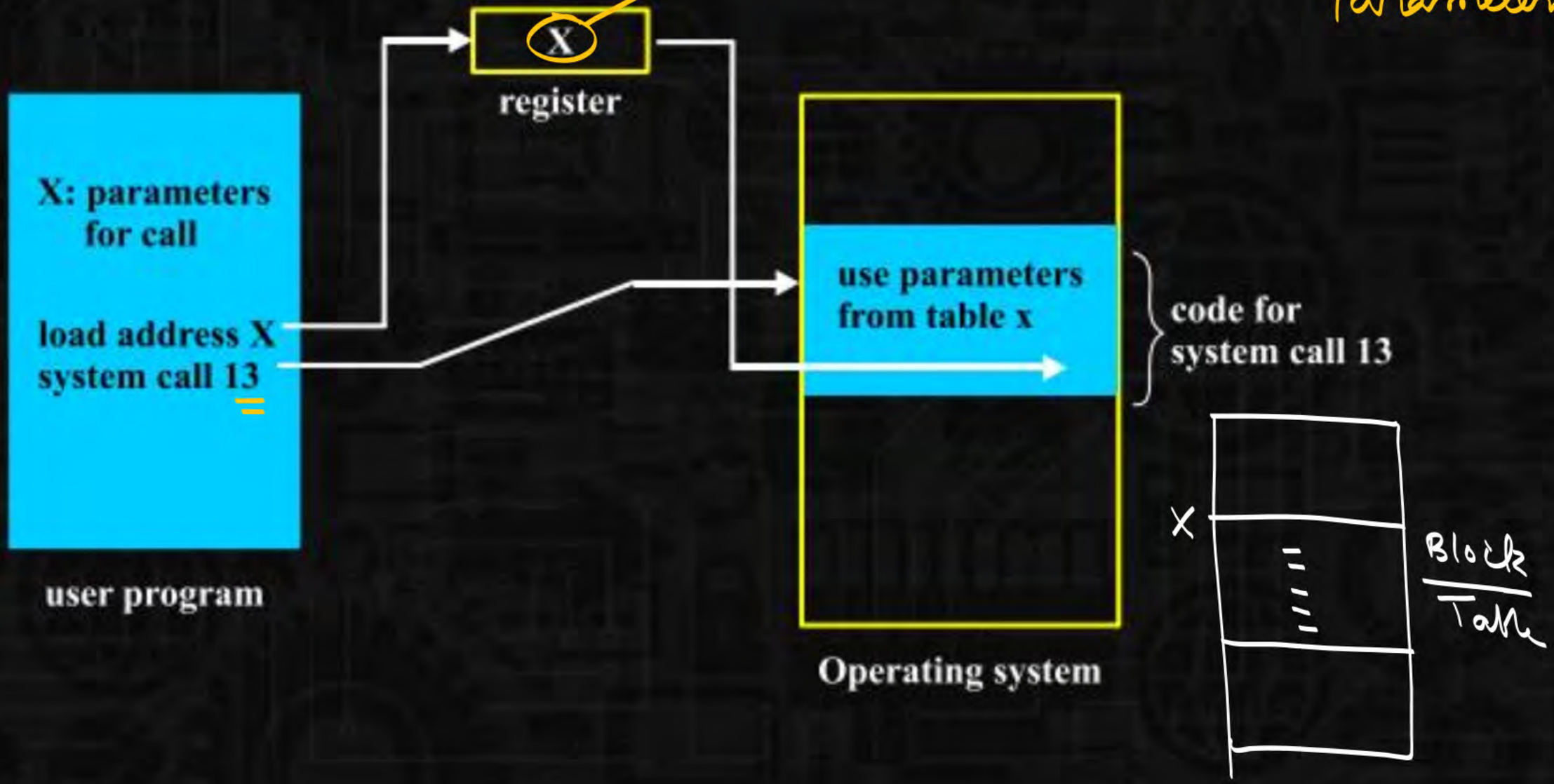
There are 11 steps in making the system call read (fd, buffer, nbytes)

System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table

Address of Block of Mem. containing Parameters;



Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - Debugger for determining bugs, single step execution
  - Locks for managing access to shared data between processes

Types of System Calls

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications <IPC Mechanisms>
  - create, delete communication connection
  - send, receive messages if message passing model to host name or process name
    - From client to server
  - Shared-memory model create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Some System Calls For Process Management

Process Management	
Call	**Description**
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Some System Calls For File Management

File Management	
Call	**Description**
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Some System Calls For Directory Management	
Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Some System Calls For Miscellaneous Tasks

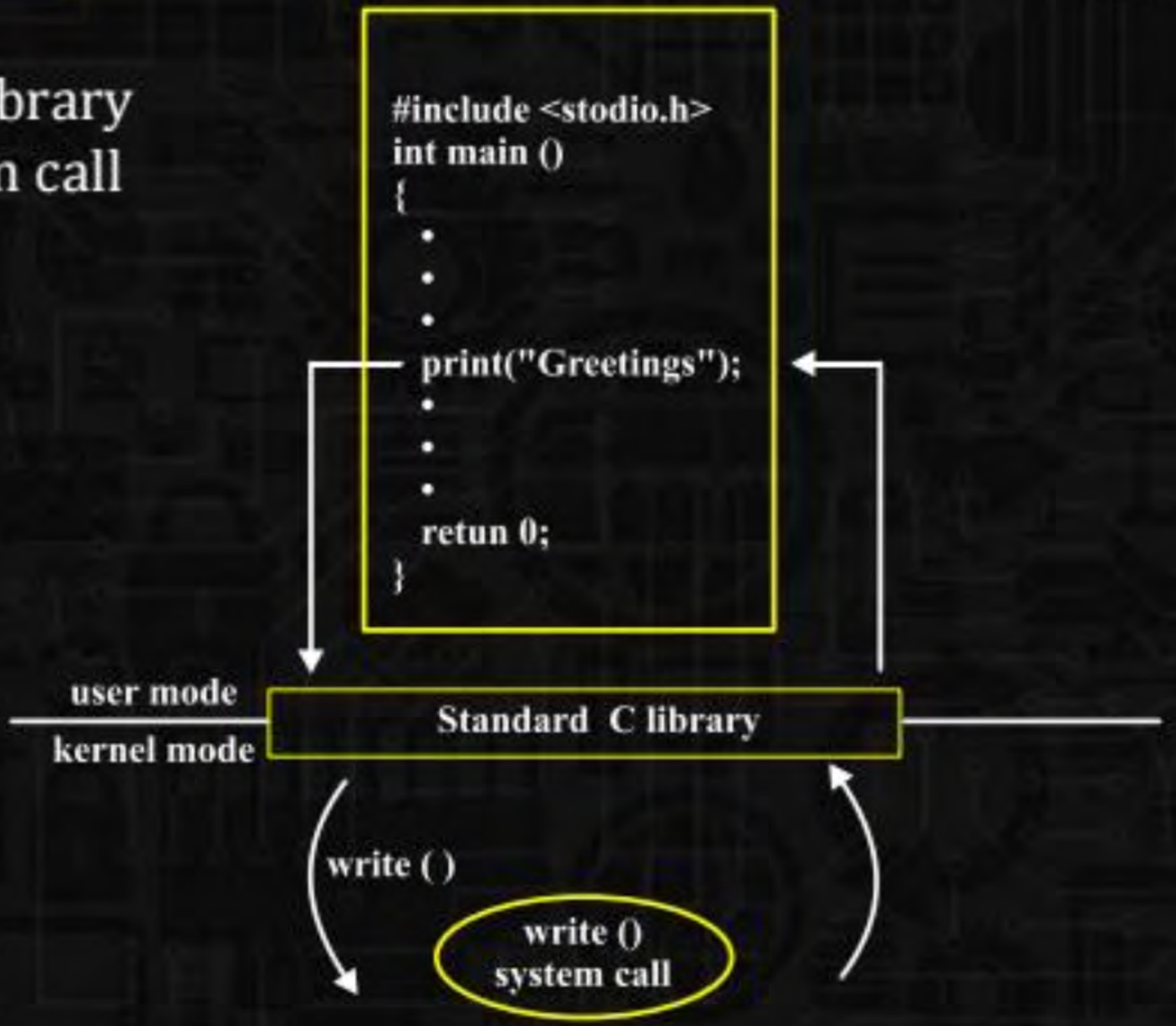
Miscellaneous	
Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan 1,1970

Examples of Unix and Windows System Calls

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device manipulation	SetConsolMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetcurrentProcessID() SetTimer() Sleep()	getpid() slarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSEcurityDiscriptorGroup()	chmod() umask() chown()

Standard C Library Example

- C program invoking printf() library call, which calls write() system call



Common and well known system calls are:

- access: checks if calling process has file access
- alarm: sets a process's alarm clock
- chdir: changes the working directory
- chmod: changes the mode of a file
- chown: changes the ownership of a file
- chroot: changes the root directory
- close: closes a file descriptor
- dup, dup2: duplicates an open file descriptor

Common and well known system calls are:

- execl, execv, execle, execve, execlp, execvp: executes a file
- exit: exits a process
- fcntl: controls open files
- fork: creates a new process
- getpid, getpgrp, getppid: gets group and process IDs
- getuid, geteuid, getgid, getegid: gets user and group IDs
- ioctl: controls character devices
- kill: sends a signal to one or more processes
- link: links a new file name to an existing file
- lseek: moves read/write file pointer

Common and well known system calls are:

- mknod: makes a directory, special or ordinary file
- mount: mounts a filesystem
- msgctl, msgget, msgsnd, msgrcv: message passing support
- nice: changes priority of a process
- open: opens a file for reading or writing
- pipe: creates an interprocess pipe
- plock: locks a process in memory
- ptrace: allows a process to trace the execution of another

Common and well known system calls are:

- read: reads from a file
- semctl, semget, semop: semaphore support
- setpgrp: sets process group ID
- setuid, setgid: sets user and group IDs
- shmctl, shmget, shmop: shared memory support
- signal: control of signal processing
- sleep: suspends execution for an interval
- stat, fstat: gets file status
- stime: sets the time
- sync: updates the super block

Common and well known system calls are:

- time: number of seconds since 1/1/1970
- times: gets process and child process times
- ulimit: gets and sets user limits
- umask: gets and sets file creation mask
- umount: unmounts a file system
- uname: gets system information
- unlink: removes directory entry
- ustat: gets file system statistics
- utime: sets file access and modification times
- wait: waits for a child process to stop or terminate
- write: writes to a file

System Programs

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- Status information
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a registry - used to store and retrieve configuration information

System Programs

- File modification
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided
- Program loading and execution - Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

Fork System Call

Case Study:

- create a child Process;
- The code of child will be an exact Replica/Copy of Parent Process;
- Execution in child will start from next Statement after fork, till end of Program;

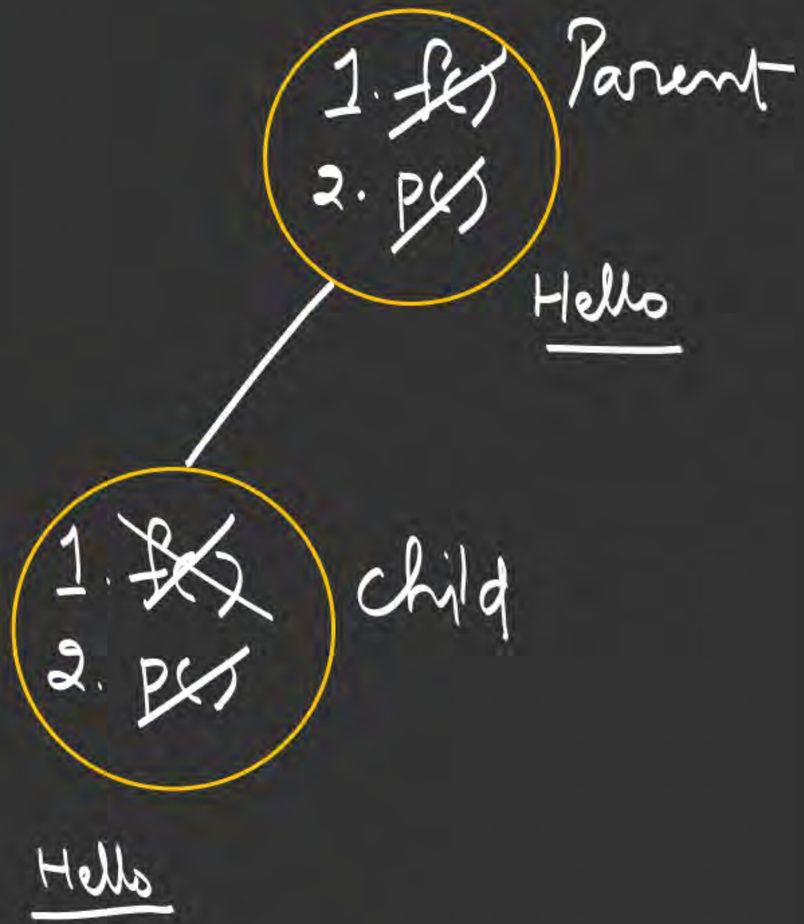
```
main()
{
    printf("Hello");
}
```

Process

printf("~~Hello~~");

Hello

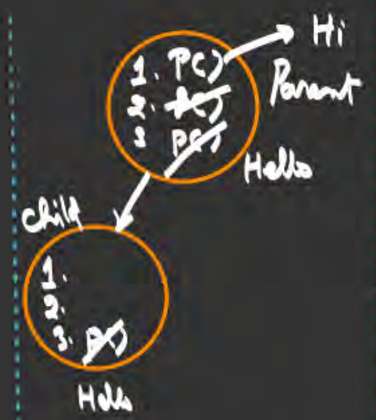
```
main()
{
    1. fork();
    2. printf("Hello");
}
```



```
main()
{
1. printf("Hi");
2. fork();
3. printf("Hello");
}
```

Total : 2

child : 1

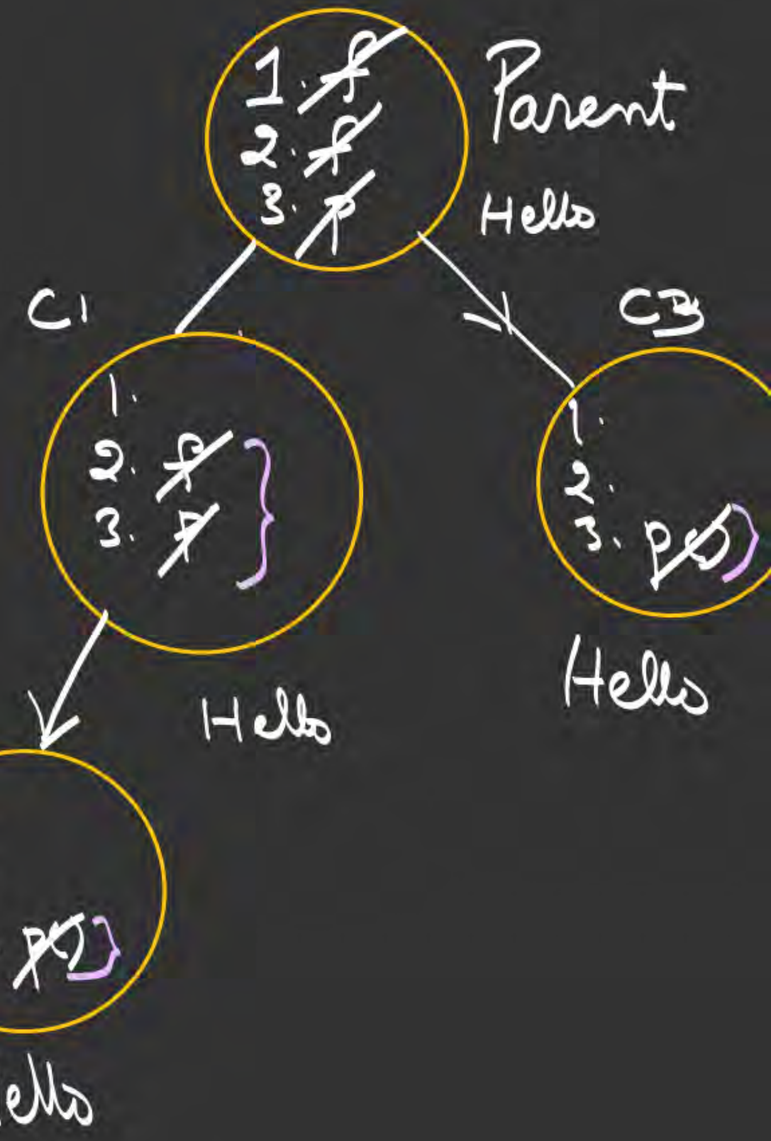



```
main()
{
1. fork();
2. fork();
3. printf("Hello");
}
```

Total Processes : 4

child Processes : 3

New

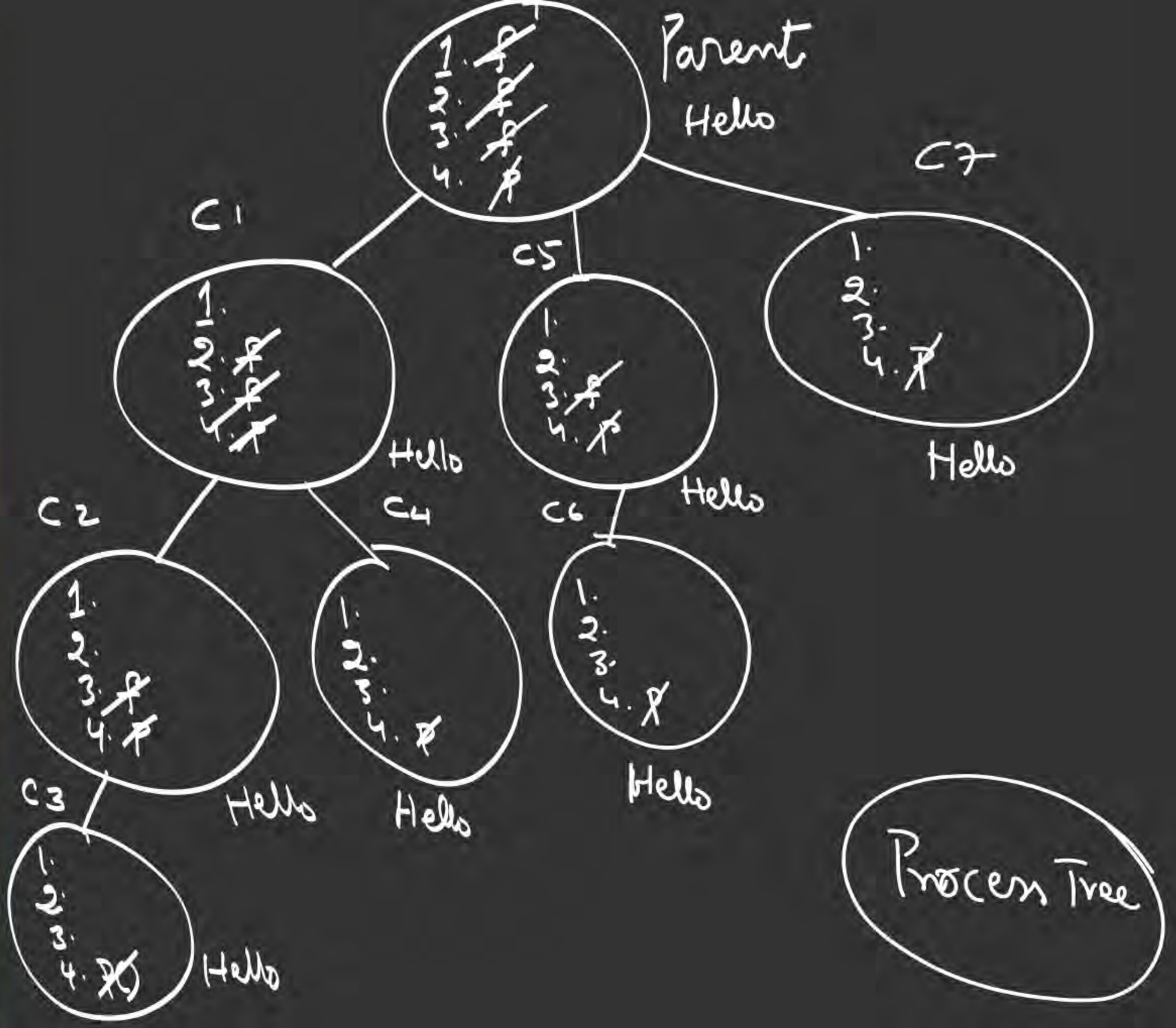


```
main()
{
 1. fork();
 2. fork();
 3. fork();
 4. printf("Hello");
}
```

3 forks:

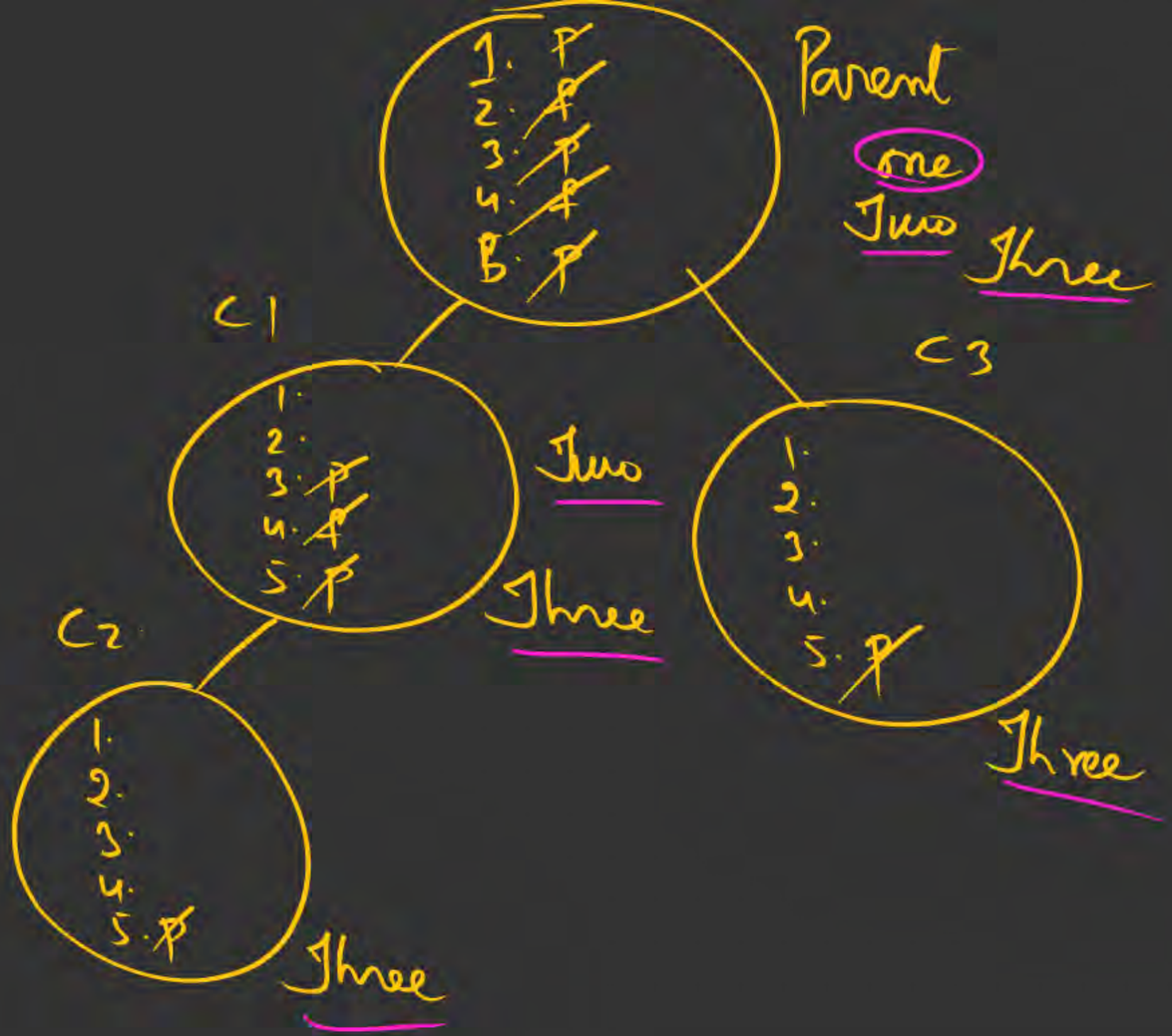
Total : 8 ✓

child : 7 ✓



No. of forks	Total Processes	Child Processes
1	2	1
2	4	3
3	8	7
4	16	15
'n'	2^n	$(2^n - 1)$

```
main()
{
1. printf("one");
2. fork();
3. printf("Two");
4. fork();
5. printf("Three");
}
```



```
main()
{
    int i, n;
    for(i=1; i<=n; ++i)
        fork();
}
```

How many child Processes are created =

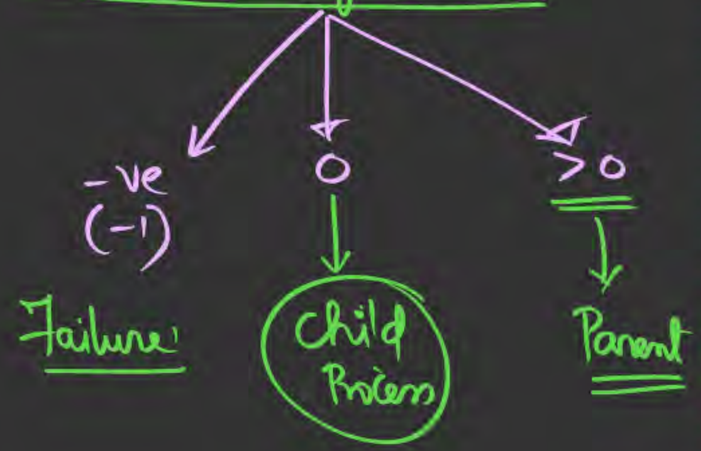
```
fork();
fork();
fork();
  :
  :
fork();
```

'n' Times

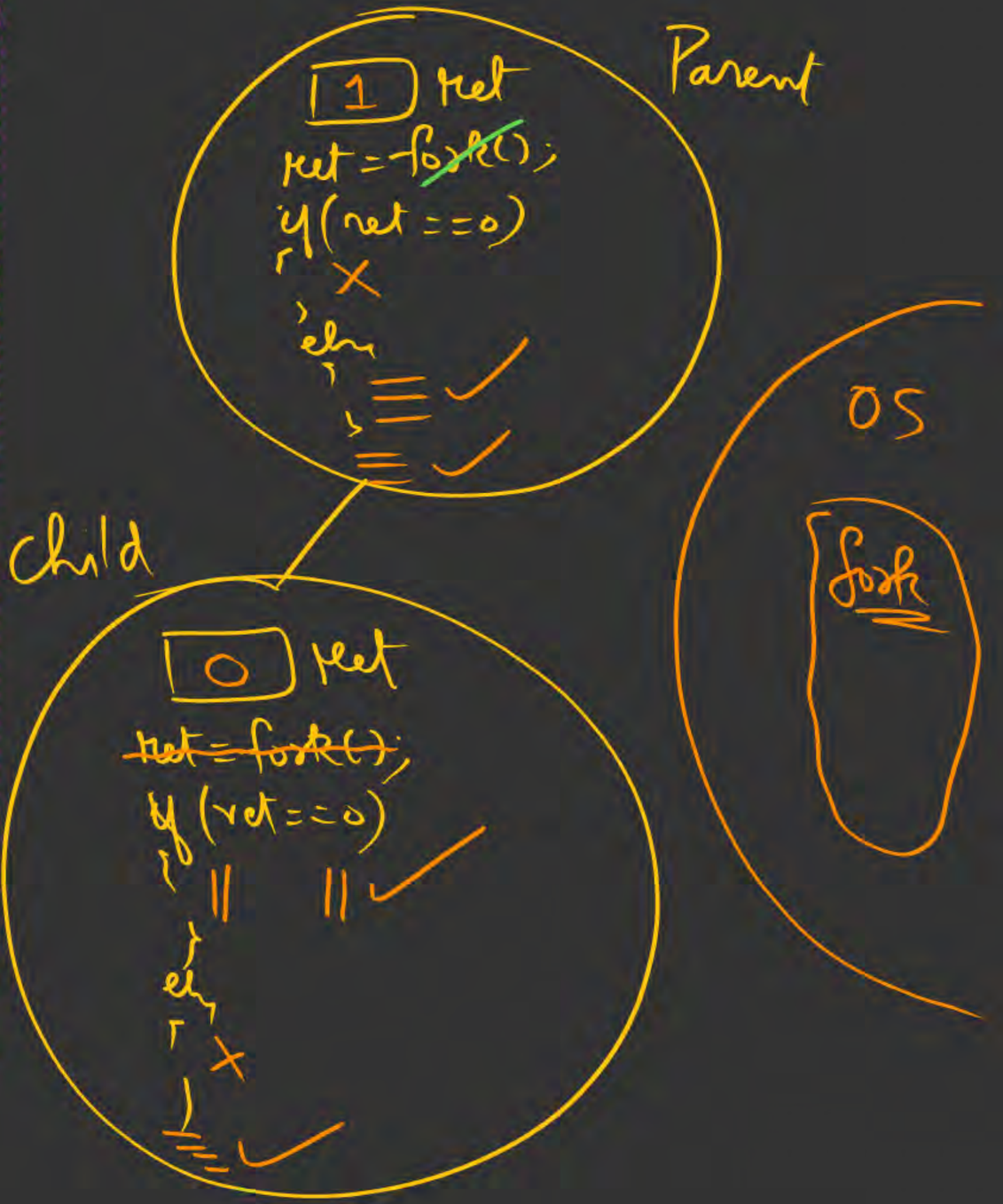
$= 2^n$

Child : $(2^n - 1)$

Return value of fork()




```
{Assume: fork() always succeed}
main()
{
    int ret;          <Parent>
    ret = fork();
    if(ret==0)
    {
        || child ||
    }
    else
    {
        || Parent
    }
    == <child + Parent>
}
```



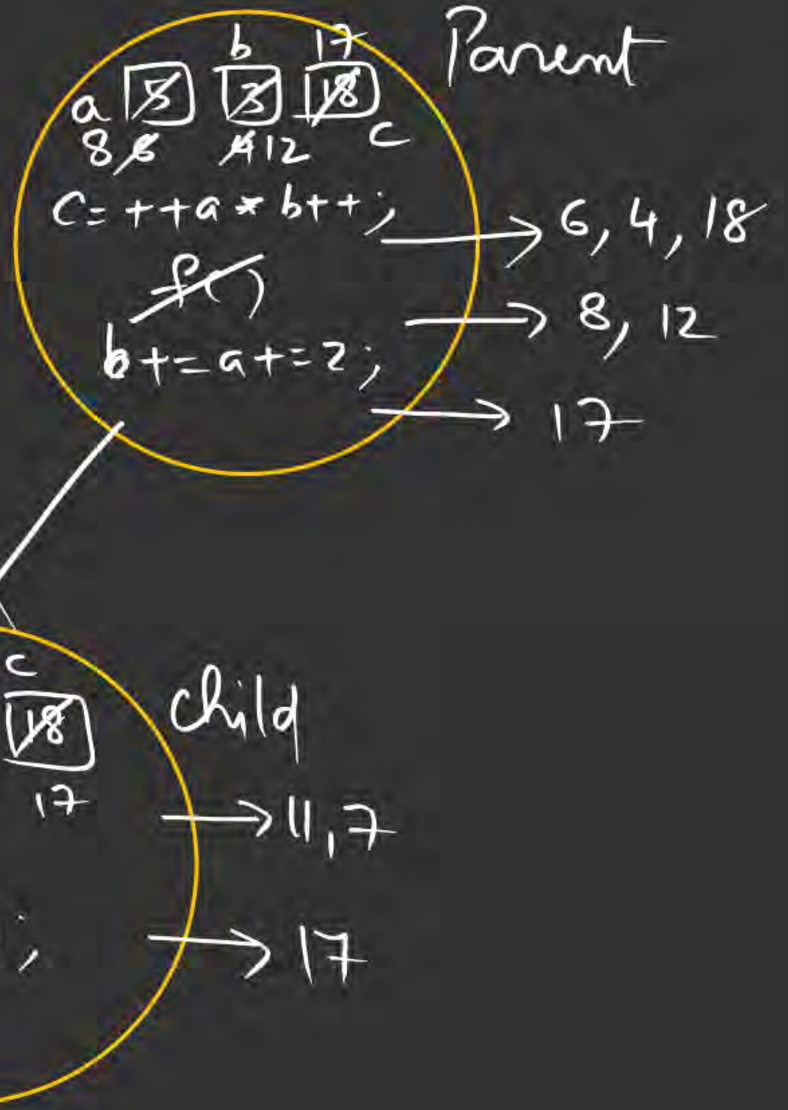
```
main()
{
    int a=5, b=3; c;
    c = ++a * b++;
    Pf(a,b,c);
    if(fork()==0)
    {
        a+=5;
        b+=3;
        printf(a,b);
        c--;
    }
    else
    {
        b+=a+=2;
        Pf(a,b);
        c--;
    }
```

child

parent

P+C

```
Pf(c);
```



Q. 1

```
main ()
{
    int i, n;
    for (i = 1; i <= n; ++i)
    if (fork ( ) == = 0);
        print ("*");
}
```

PYQ

No. of times '*' get printed = 2^n

→ every Process will execute

```
int i, n;
for(i=1; i<=n; ++i)
{
    if((i % 2) == = 0)
        (fork();)^{n/2}
}
```

No. of children: $(2^{n/2} - 1)$

Assume fork always succeeds

```
main()
{
  int a;
  if(fork()==0)
  {
    a=a+5;
    print(a); ⟶ u
  }
  else
  {
    a=a-5;
    print(a); ⟶ x
  }
}
```

⇒ What is the Relation b/w 'u' & 'x'

a) u = x

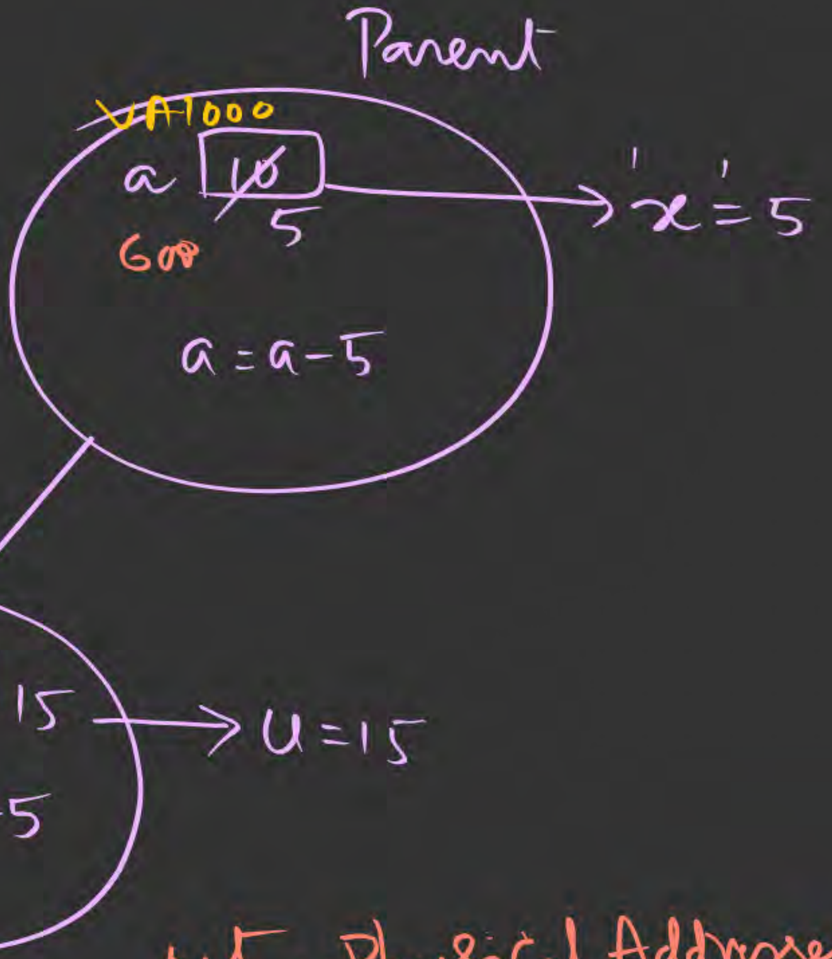
b) u = x+5

c) u = x+10 ✓

d) x = u-5

&a ⟶ Same (VA)

&a ⟶ Same (VA)



Note: 1) Physical Addresses will be different

2) Virtual Addresses of 'a' will be Same

Q. 2

H/W

```
main ()
 {
      int i, n;
      for (i = 1; i< = n; ++i)
      {
      1. fork ();
      2. print ("*");
      }
 }
```

Q. 3

```
main ( )
{
    int i, n;
    for (i = 1; i < = n; ++ i)
    {
     1. print ("*");
     2. fork ( );
    }
}
```

H/W

Q3

" * "

formula

Q2

```
for i ← 1 to n
{
    fork();
    print(*);
}
```

" * "

*

formula

Q. 3

```
main ( )
 {
      int i, n;
      for (i = 1; i < = n; ++ i)
      {
          print (“*”);
          fork ( );
      }
 }
```

Q. 4

```
main ()
 {
      int a = 1, b = 2, c = 3;
      a + = b+ = + + c;
      print (a, b, c);
      if (fork () = = 0)
       {
        int d;
       + + a; + + b; – – c;
       print (a, b, c);
       if (fork () = = 0)
        {
            d = a + b + c;
            print (a, b, c, d);
       }
      else
       {
            – – a; – – b;
            c = a + b; d = a + b + c;
            print (a, b, c, d);
       }
       }
      else
       {
       c + = b + = + + a;
       print (a, b, c) ;
       }
       print (d);
 }
```

Q. 5

PYQ

The following C program is executed on a Unix/Linux system:

```
#include <unistd. h>
int main( )
{     int i;
      for (i = 0; i < 10; i + +)
          if (i % 2 = = 0) fork( );
}
```

The total number of child processes created is 31

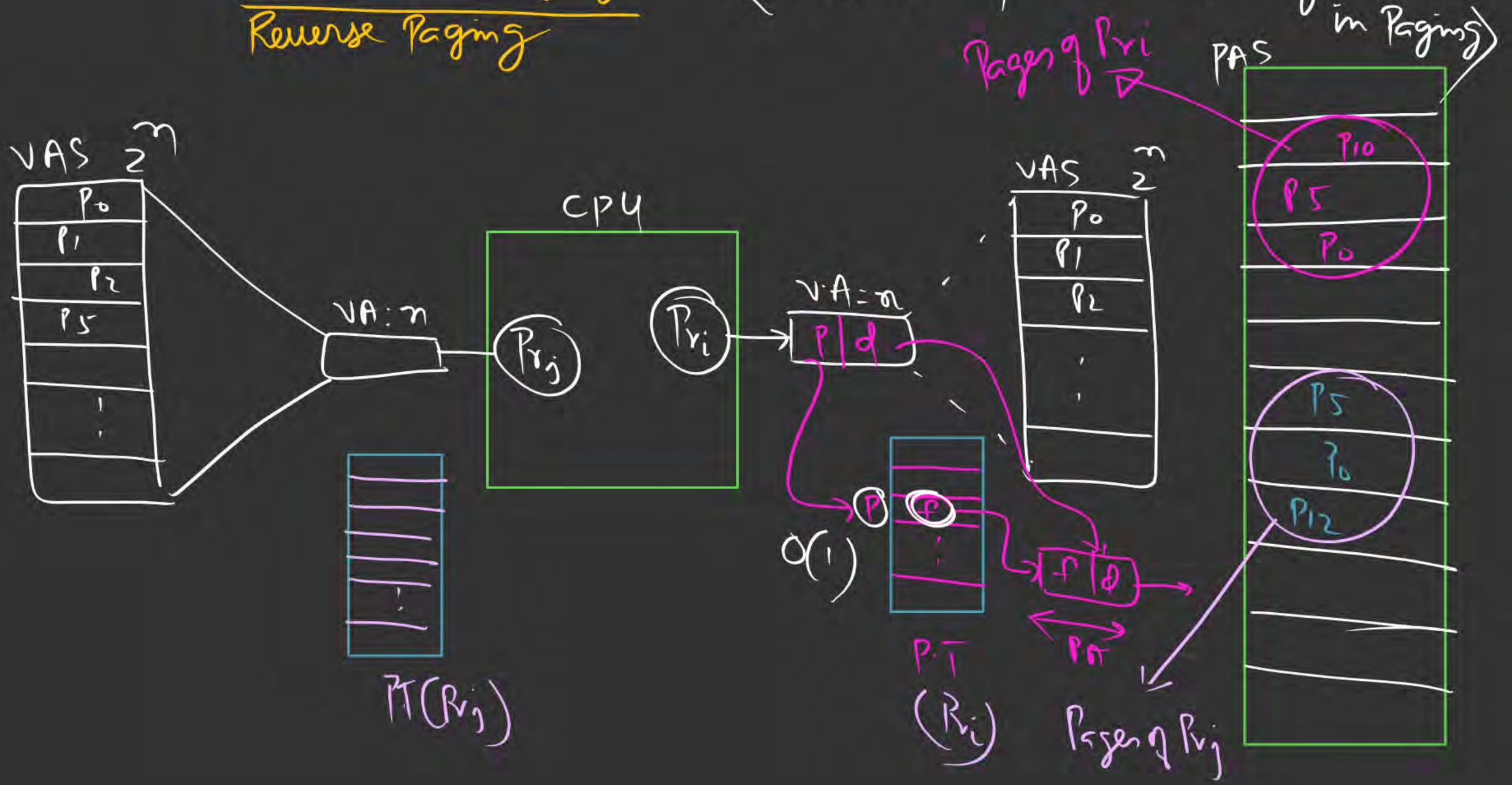
```c
#include <sys/types.h>
#include < stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* fork a child process */
    pid = fork () ;
    if (pid < 0) { /* error occurred */
        fprintf ( stderr, "Fork Failed");
    return 1;
    } else if (pid == 0){/* child process */
        execlp ("/ bin/ls", "ls", NULL);
    }
    else {/* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
    } return 0;
}
```

- fork() ; exec ; wait

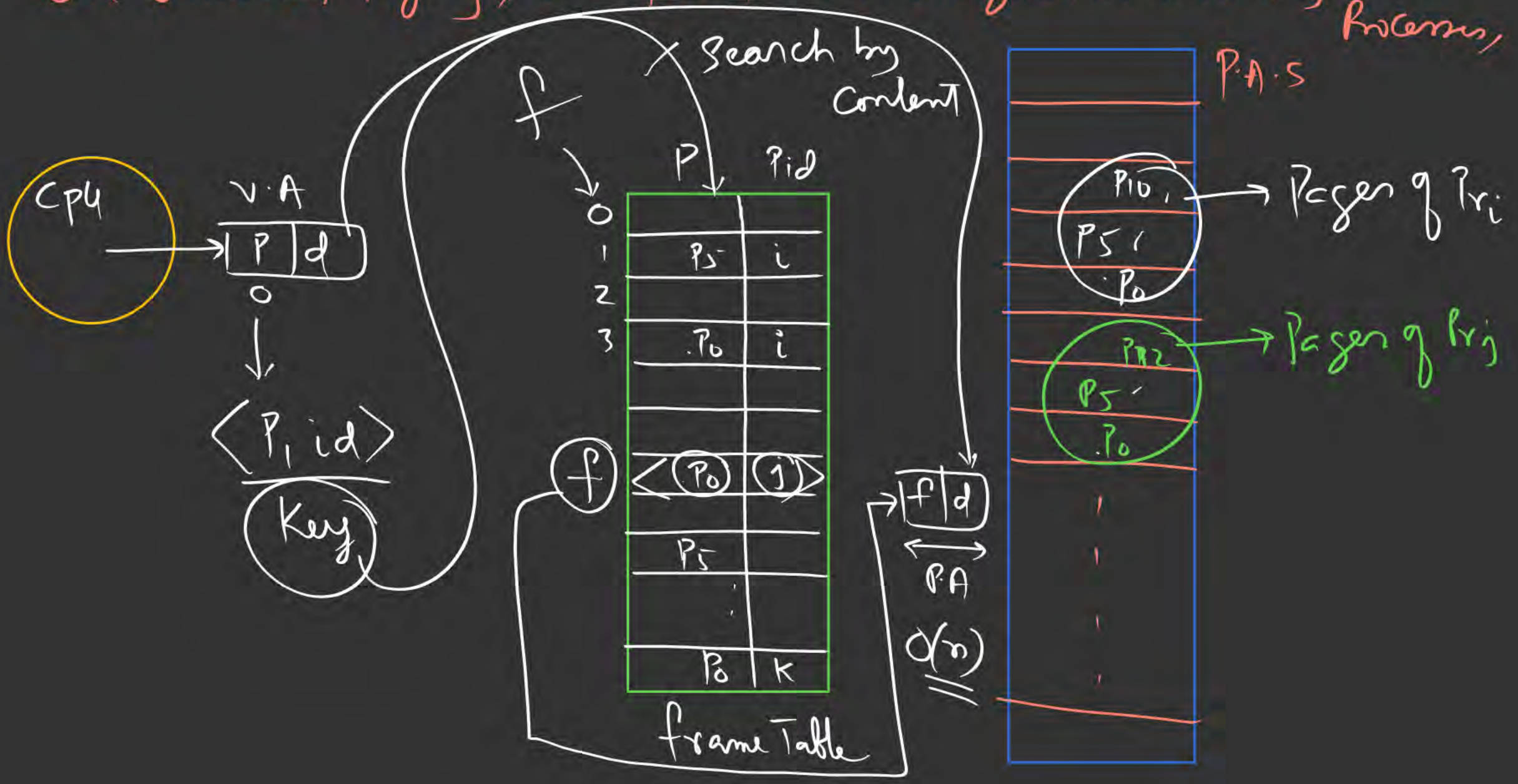
Inverted - Paging

Reverse Paging

< Reduce Space overhead of P.T in Paging



In Inverted Paging, one Global P.T is designed to be used by all Processes,



Consider a System with V.A = 34 bits & P.A = 29 bits;

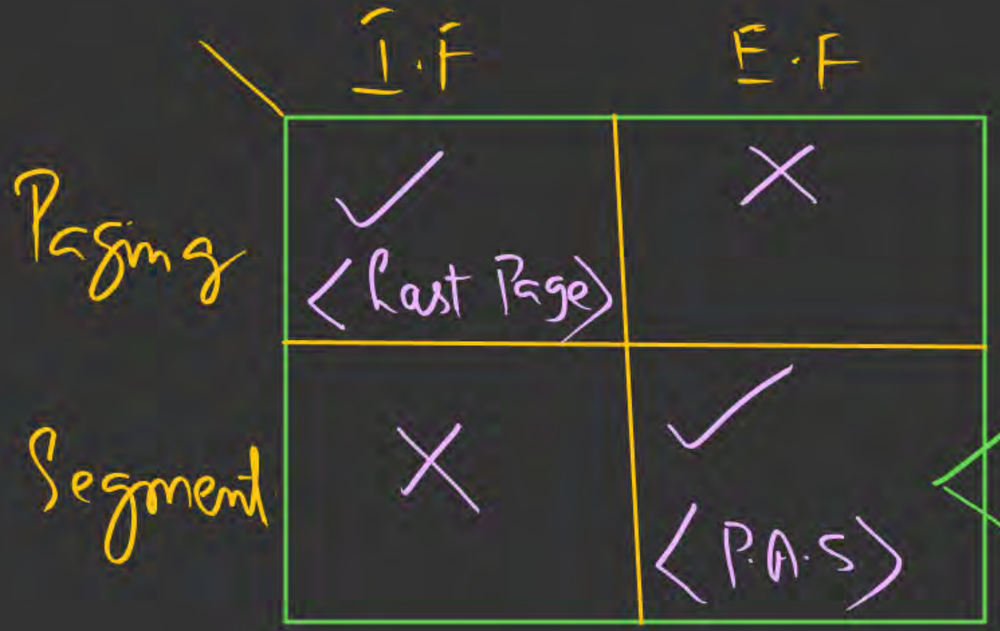
P.S = 8KB; P.T.E = 32 bits;

What is the Size of

(i) Traditional P.T: $\left(\frac{2^{34}}{2^{13}}\right) * 2^{2}$ B $= 2^{21+2} = 2^{23} = 8$ MB ✓

(ii) Inverted P.T: $\left(\frac{2^{29}}{2^{13}}\right) \times 2^{2} = 2^{16+2} = 2^{18} = 256$ KB

Paging vs Segmentation



2) To reduce Seg Table Size ovhd
we apply Paging on S.T

1) Compaction

2) Paging on Segment
① (Segmented - Paging)


THANK YOU GW
SOLDIERS !