CS & IT ENGINEERING

Operating System

System Calls and Threads Part-1

Lecture no:01

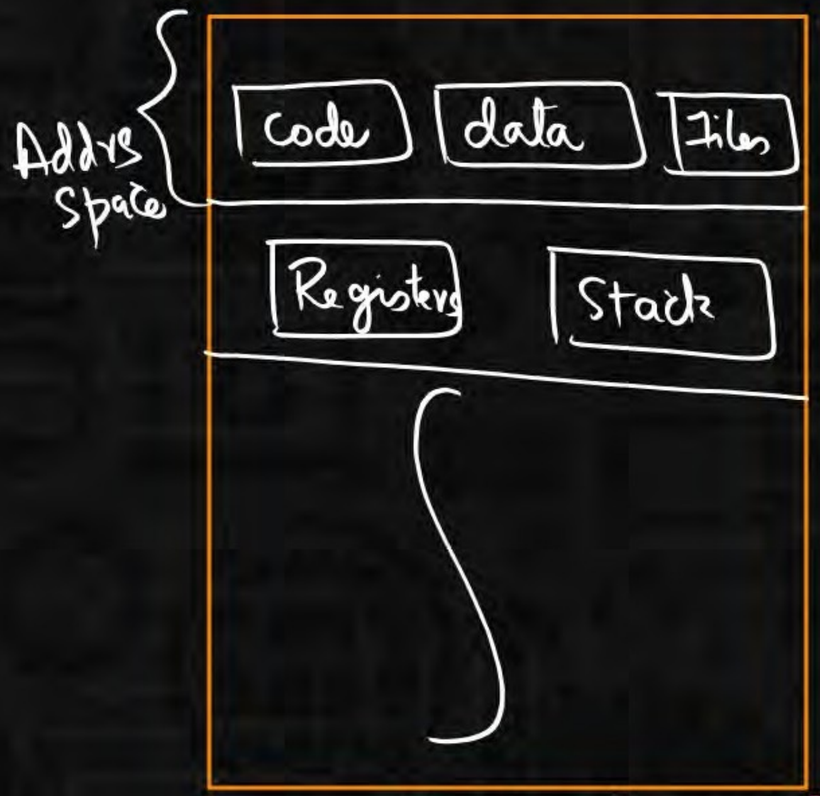
By- Dr. Khaleel Khan sir

System Calls and Threads
Part-1

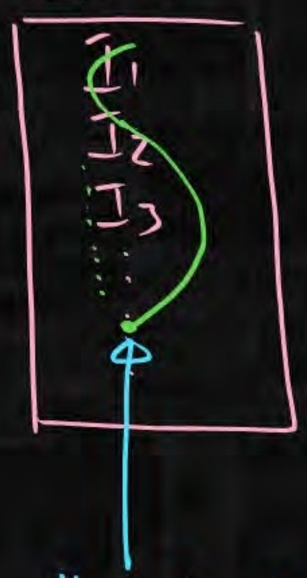
Threads and Multithreading

Thread : Light weight (Process) → unit of Execution (cpu)
Program under execution

Why & what is the need for M·T



Single Threaded Process



the order of execution of Instrns

Client - Server: n-req's

Concurrency — M·T

Multiprocess (fork)

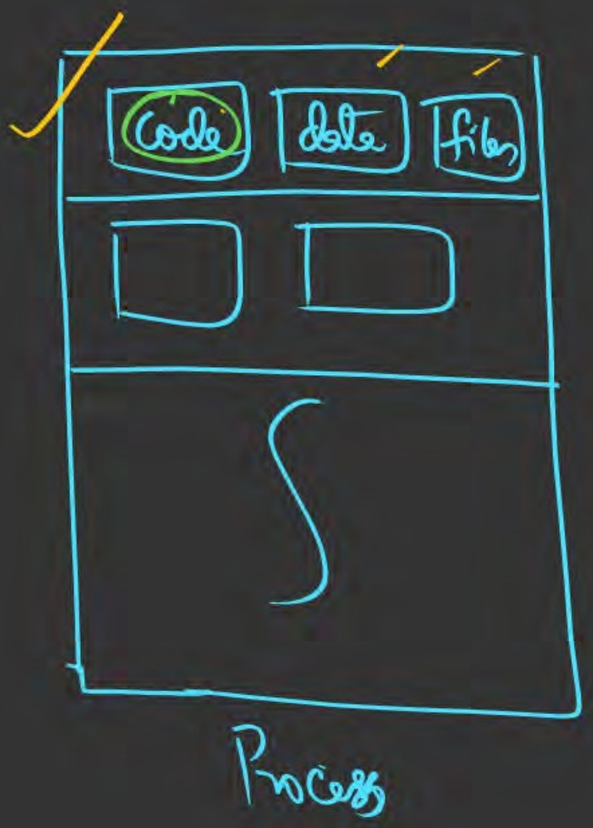
Iterative Mode

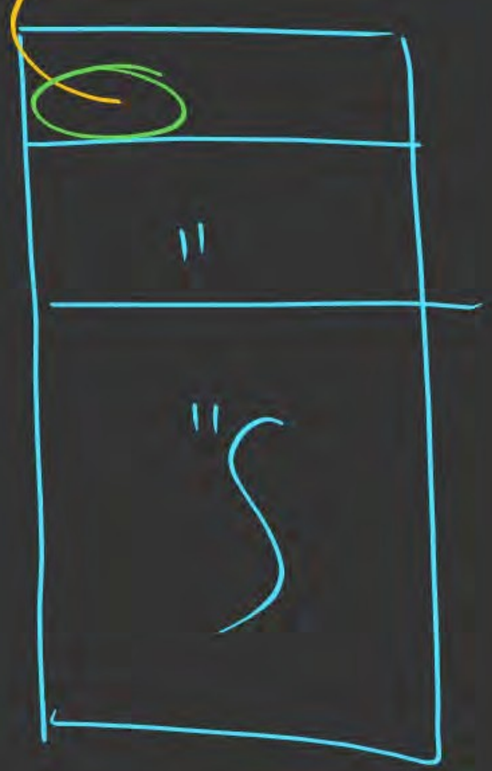
Concurrent Server

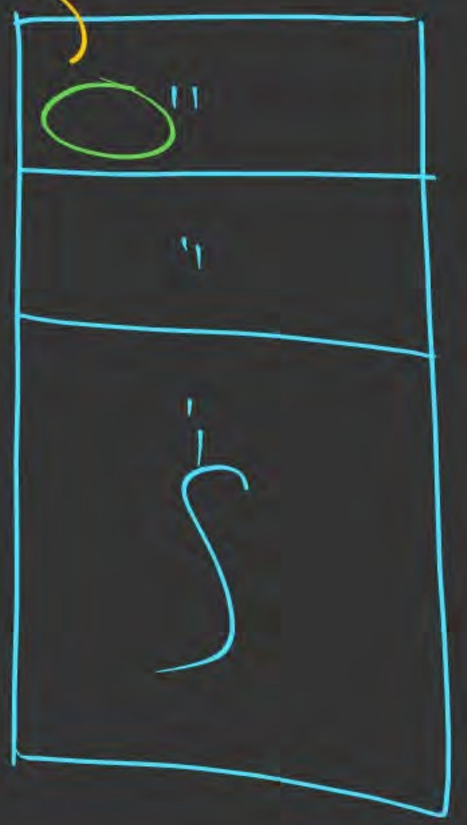
Drawback / Limitation of achieving Concurrency in Server design thru Multiprocess approach,

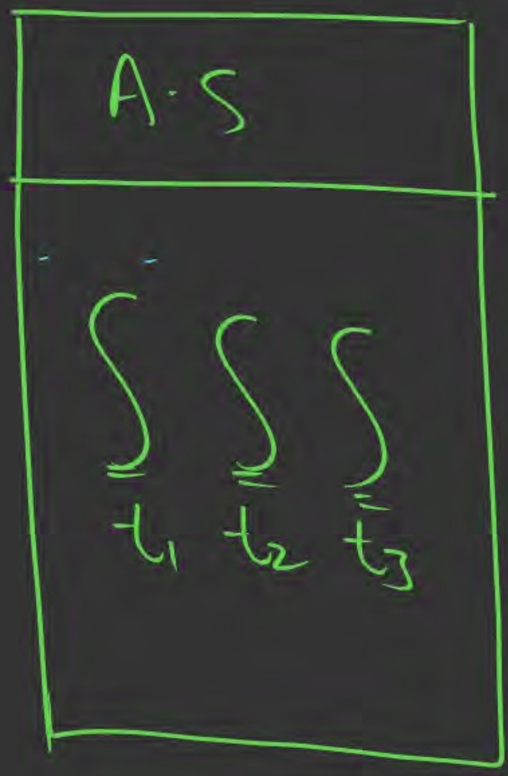
n - copies

Address Space | Resources









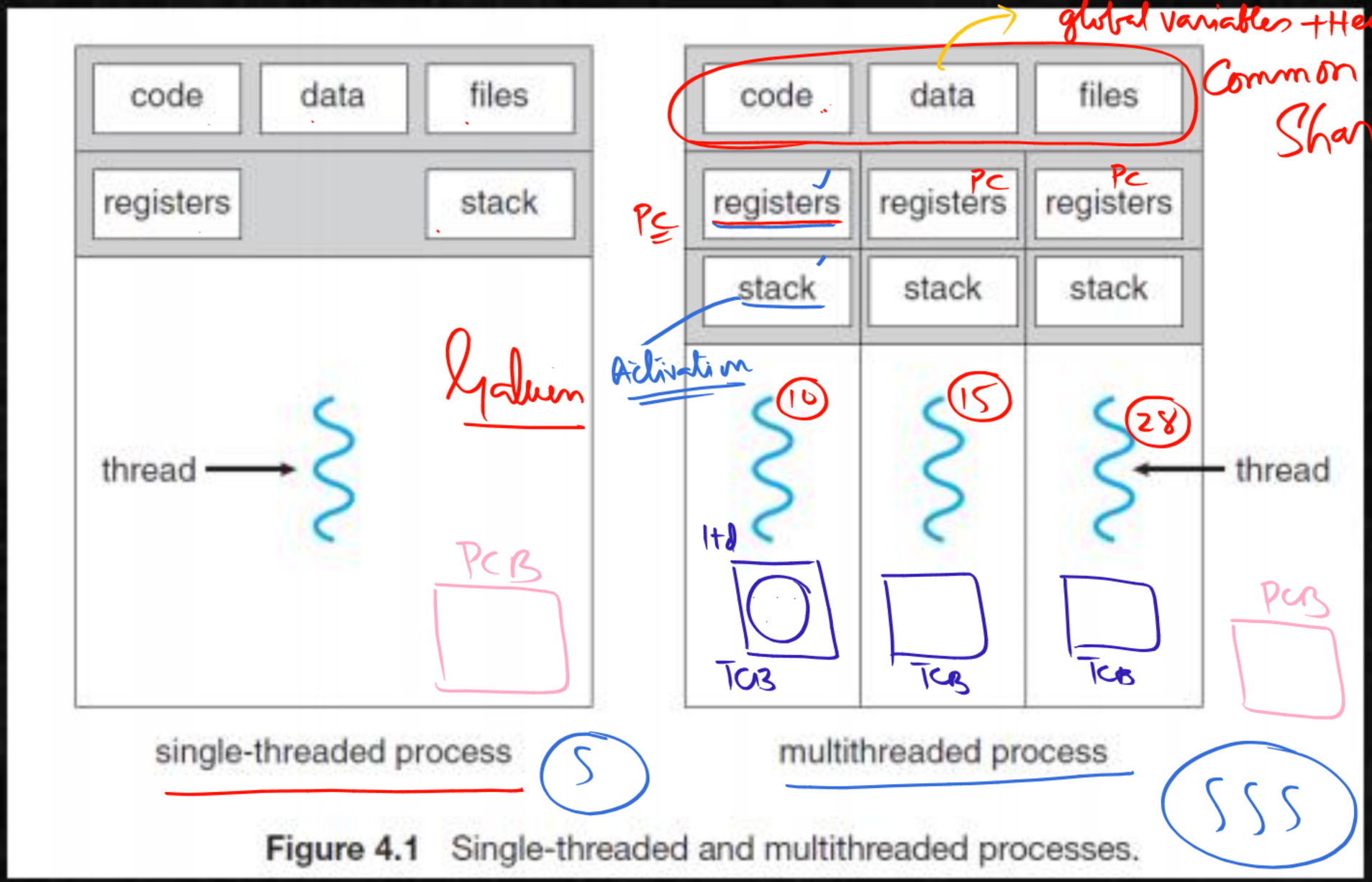


Figure 4.1 Single-threaded and multithreaded processes.

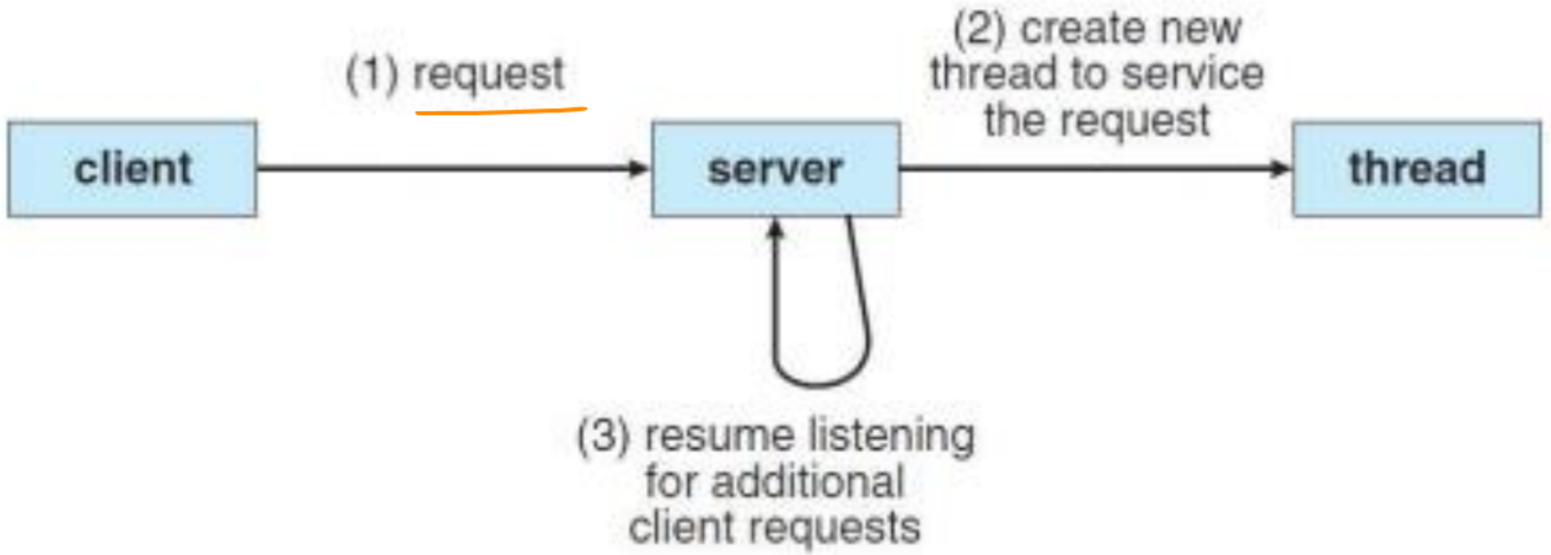
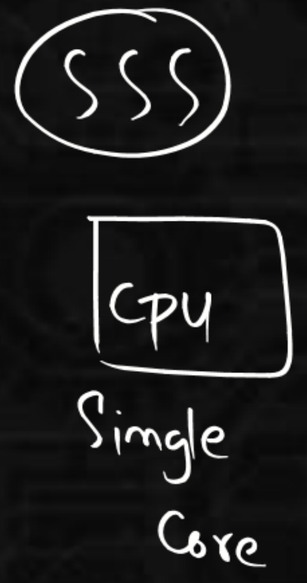


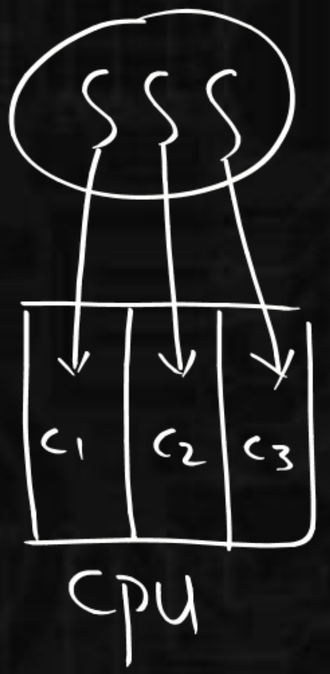
Figure 4.2 Multithreaded server architecture.

4.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

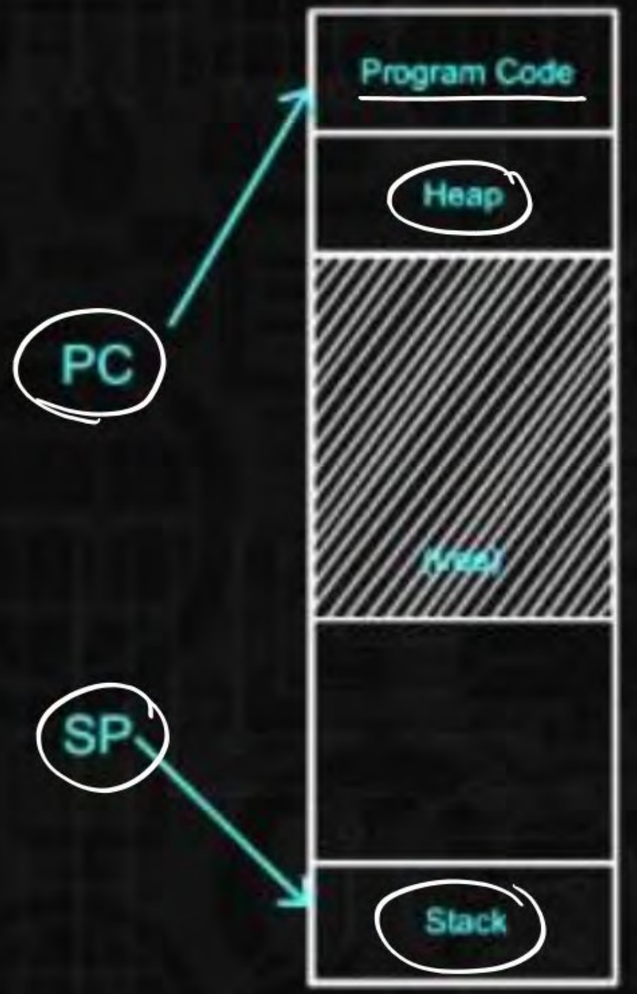
1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.
2. **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is significantly more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times
4. **Scalability**. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.





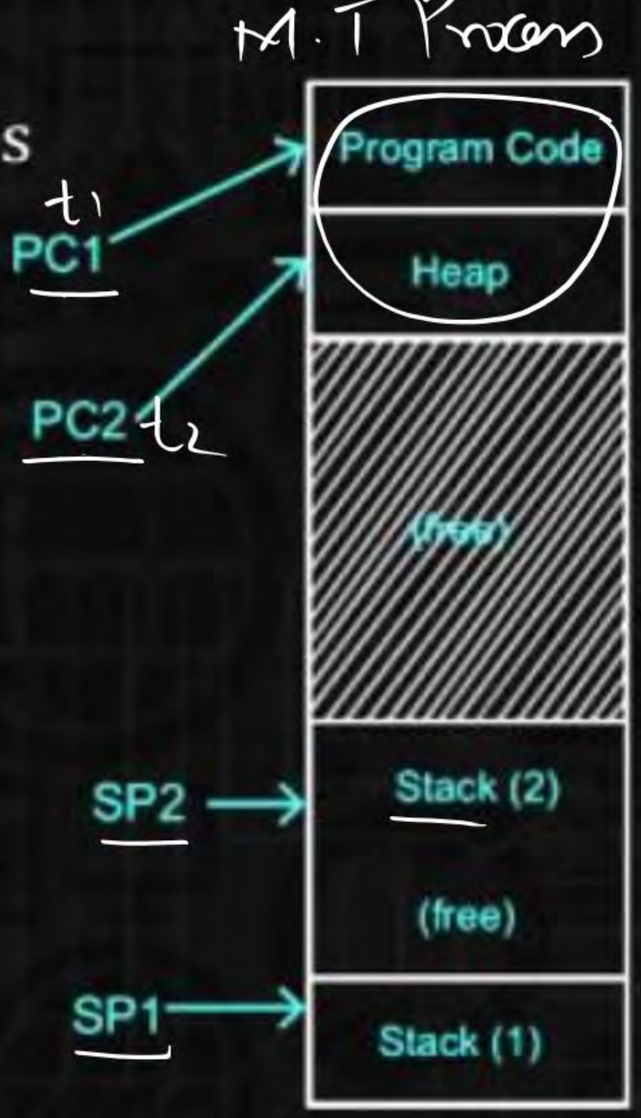
Single Threaded Process

- So,far we have studied single threaded programs
- Recap: process execution
  - Pc points to current instruction being run
  - SP points to stack frame of current function call
- A program can also have multiple threads of execution
- What is a thread



Multi Threaded Process

- A thread is like another copy of a process that executes Independently.
- Threads share the same address space (code heap)
- Each thread has a separate PC
  - Each thread may run over different part of a program
- Each thread has a separate stack for independent function calls



Process Vs. Thread

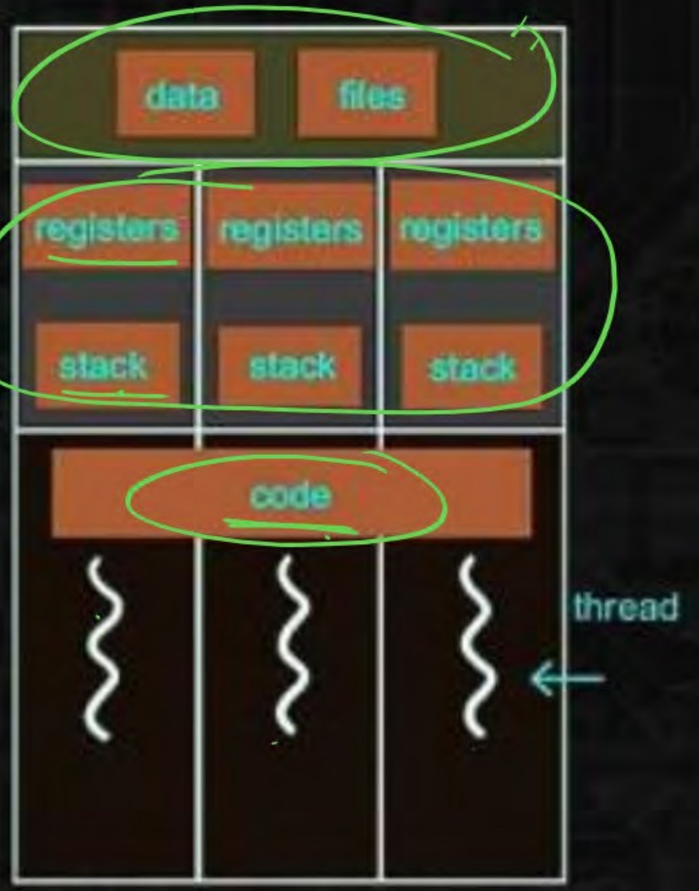
- Parent P forks a child C
  - P and C does not share any memory
  - Need complicated IPC mechanism to communicate
  - Extra copies of code, data in memory
- Parent p execute two threads T1 and T2
  - T1 and T2 share parts of the address space
  - Global Variables can be used for communication
  - Smaller memory footprint
- Threads are like separate processes, except they share the same address space

Why Threads?

- Parallelism: single process can effectively utilise multiple CPU cores
  - Understand the difference between concurrency and the parallelism
  - Concurrency: running multiple threads/ processes at the same time, even on single CPU core, by interleaving their execution (Preemptive)
  - Parallelism: running multiple threads/process in parallel over different CPU cores
- Even if no parallelism can concurrency of threads ensure effective use of CPU when one of the threads blocks(e.g.,for I/0)

Threads

- Separate stream of execution within a single process
- Threads in a process not isolated from each other
- Each thread States (thread control block) contains
  - Registers including (EIP, ESP)
  - Stack



Threads Vs Processes

- A thread has no data segment or heap.
- A thread cannot live on its own it need to be attached to a process.
- There can be more than one thread in a process. Each thread has its own stack.
- If a thread dies, its stack is reclaimed.
- A process has code, heap, stack, and other segments
- A process has at-least one thread.
- Heads within a process share the same code, files.
- If a process dies, all thread die.

Pthread library

Pthread:

POSIX

- Create a thread in a process

```
Int
pthread_create(pthread_t*thread,
Const pthread_attr_t*attr,
void*(*start_routine) (void*),
void*arg);
```

Thread identifier (TID) much like

Pointer to a function which starts execution in a different thread

Arguments to the function

- Destroying a thread

```
void
pthread_exit(void*retval);
```

Exit vale of the thread

Pthread library contd.

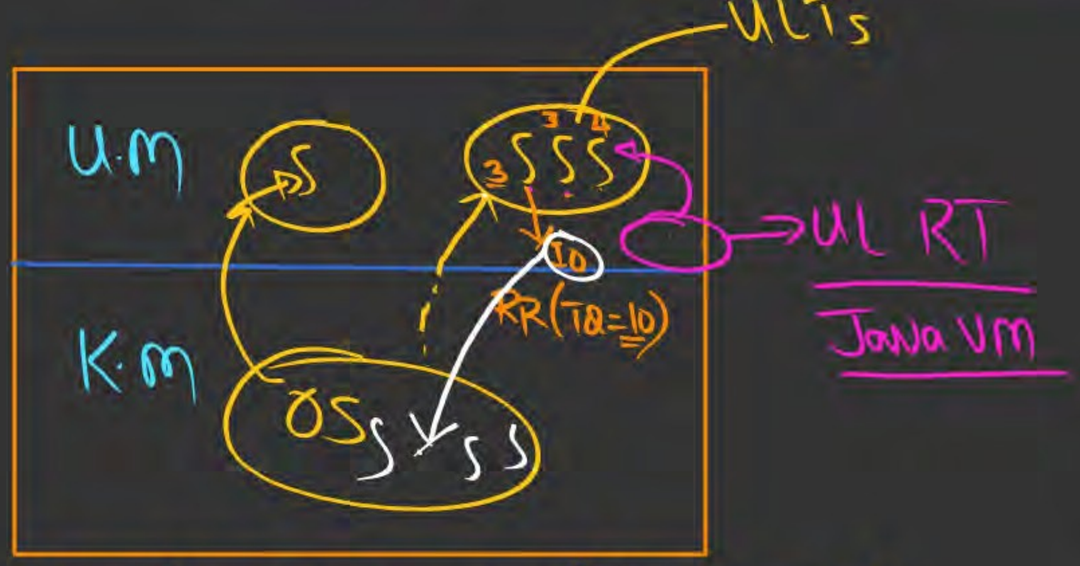
- Join : wait for a specific thread to complete

Int pthread_join(pthread_t thread, void**retval);

TID of the thread to wait for exit status of the thread

⊛ Types of Threads

- (ULTs) User-level Threads
- (KLTs) Kernel Level threads

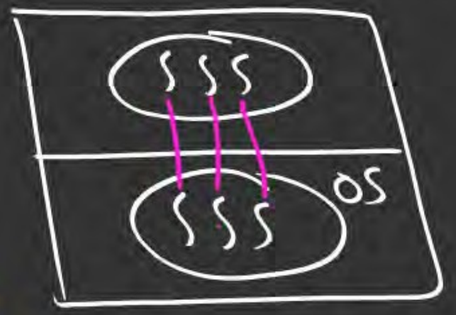


(Threads that are created & Managed @ the user level without any support of OS)

Benefits of ULTs

(i) Flexibility
(ii) Transparency
(iii) Fastest Content switching
↳ does not require Mode Shifting

(iv) If a ULT initiates IO or execute any S.C
↓
Blocking of whole Process



Who manages threads?

Two strategies

- User threads
  - Thread management done by user level threads library. Kernel knows nothing about the threads.

Transparency

- Kernel threads
  - threads directly supported by Kernel.
  - Known as light weight processes.

Use Level threads

Advantages:

- Fast (really lightweight) (no system calls to manage threads. The thread library does everything). JVM
- Can be implemented on an OS that does not support threading.
- Switching is fast. No switch from user to protected mode.

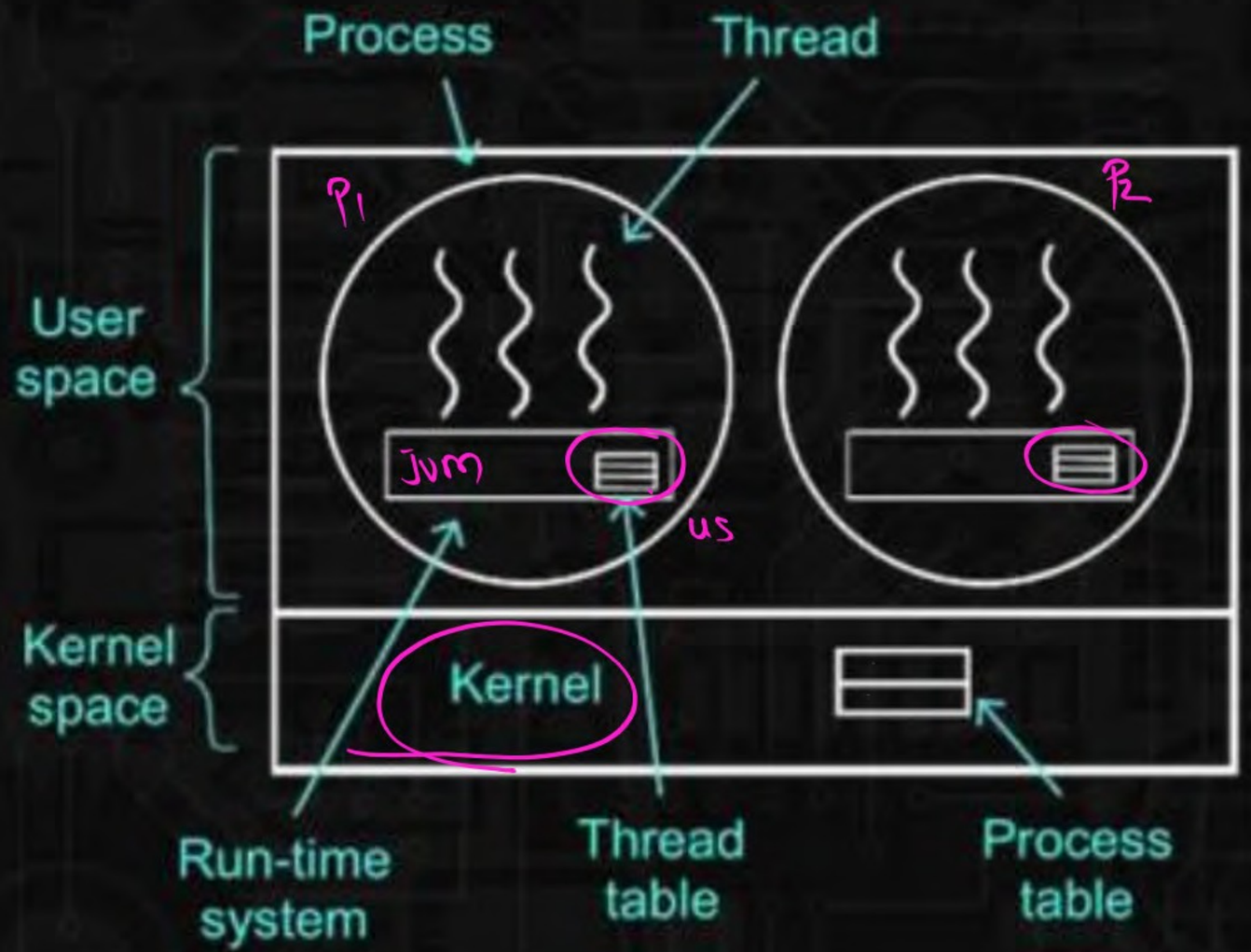
Disadvantages:

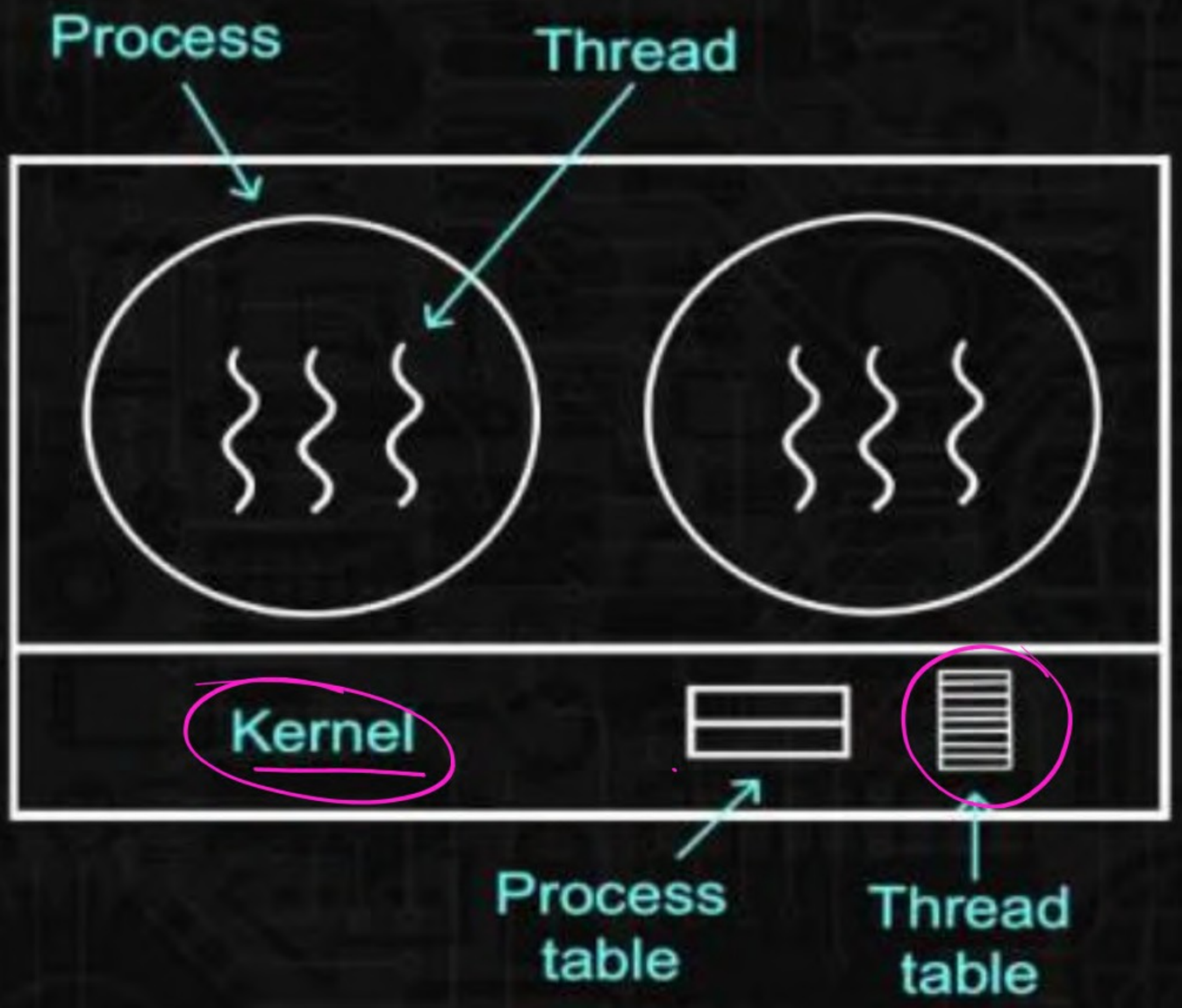
Scheduling can be an issue. (Consider, one third that is blocked on an IO and another runnable.)

Lack of coordination between kernel and threads. (A process with 100 threads complete for a time slice with the process having just 1 thread.)

Requires non-blocking system calls. (If one thread invokes a system call, all threads need to wait

Use Level threads




Process
Thread
Kernel
Process table
Thread table

Q.1

Which of the following is/are shared by all the threads in a process?

I. Program counter

II. Stack

III. Address space

IV. Registers

A. I and II only

B. III only

C. IV only

D. III and IV only

Q.2

Threads of a process share

A. Global variables but not heap

B. Heap but not global variables

C. Neither global variables nor heap

D. Both heap and global variables

Q.3

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

A. User level threads are not scheduled by the kernel. T

B. When a user level thread is blocked, all other threads of its process are blocked. T

C. Context switching between user level threads is faster than context switching between kernel level threads. T

D. Kernel level threads cannot share the code segment. F

Q.4

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.

Q.5

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

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

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

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

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

Q.6

Which one of the following is NOT shared by the threads of the same process?

A. Stack

B. Address Space

C. File Descriptor Table

D. Message Queue

Shared

Q.7

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

I. Context switch is faster with kernel-supported threads. X

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

III. Kernel supported threads can be scheduled independently. T

IV. User level threads are transparent to the kernel. T

Which of the above statements are true?

A. II, III and IV only ✓

B. II and III only

C. I and III only

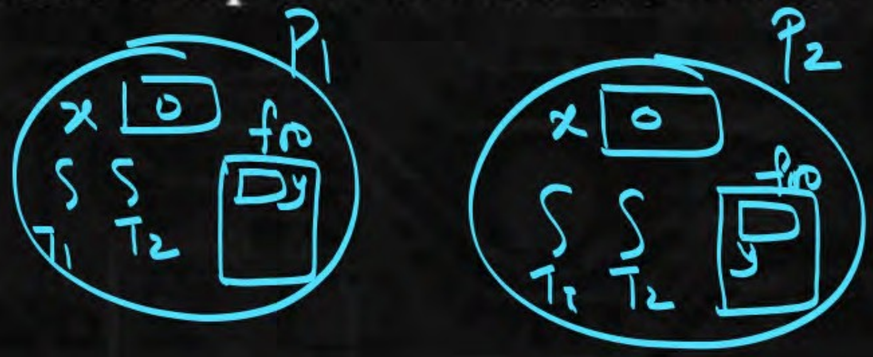
D. I and II only

Q.8

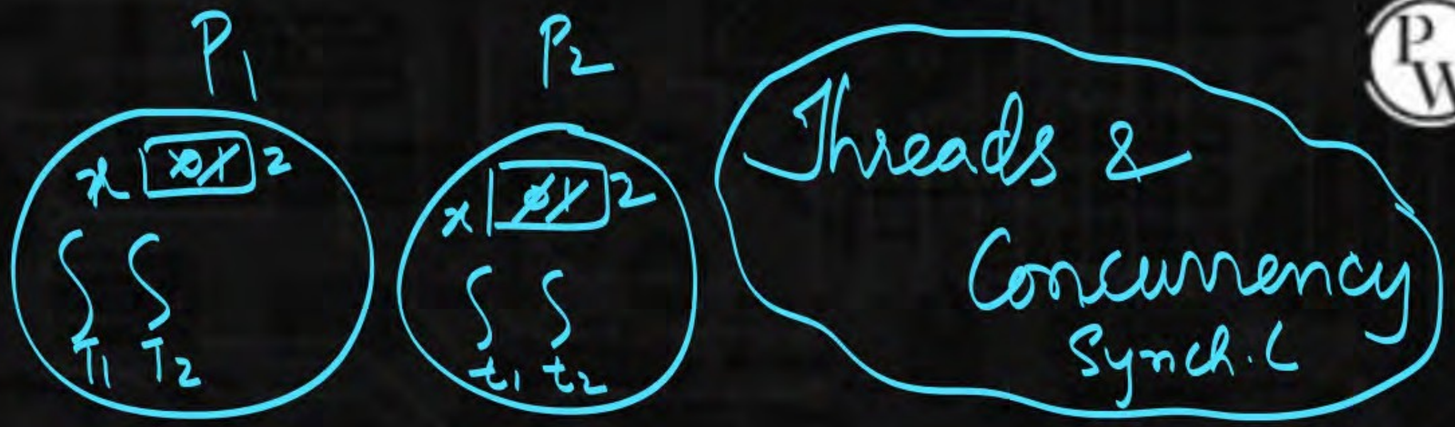
Consider the following multi-threaded code segment segment (in a mix of C and pseudo-code), invoked by two processes P1 and P2 each of the processes spawns two threads T1 and T2:

```
int x = 0; // global
Lock L1; // global

main( ) {
    create a thread to execute foo( ); // Thread T1
    create a thread to execute foo( ); // Thread T2
    wait for the two threads to finish execution;
    print(x);}
```



```
foo( ) {
    int y = 0;
    Acquire L1;
    x = x + 1;
    y = y + 1;
    Release L1;
    print (y);}
```



MSQ

(A, D)

Which of the following statement(s) is/are correct?

A. Both P1 and P2 will print the value of x as 2. ✓

B. At least of P1 and P2 will print the value of x as 4. ✗

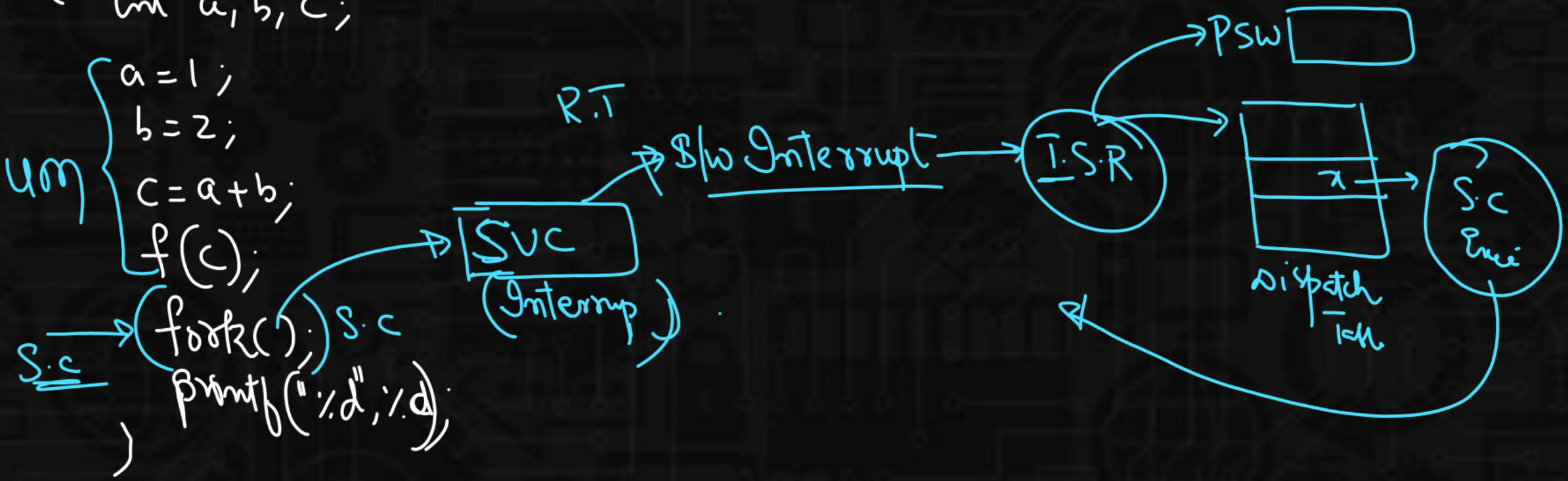
C. At least one of the threads will print the value of y as 2.

D. Both T1 and T2 , in both processes, will print the value of y as 1. ✓

2) System Calls

→ one can access OS services;

```
main()
{
    int a, b, c;
    a = 1;
    b = 2;
    c = a + b;
    f(c);
    fork();
    printf("%d", %d);
}
```



System Calls

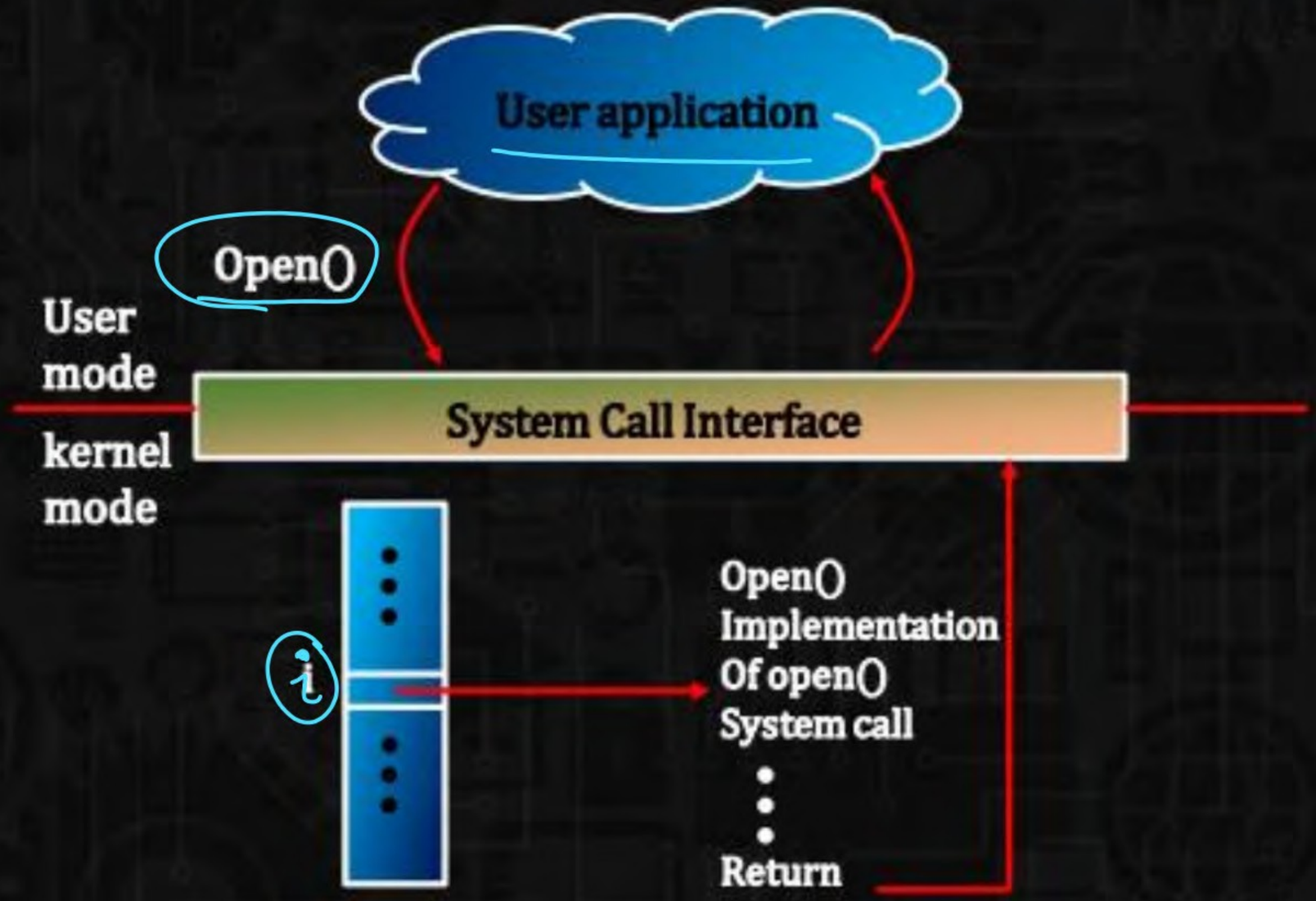
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based 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

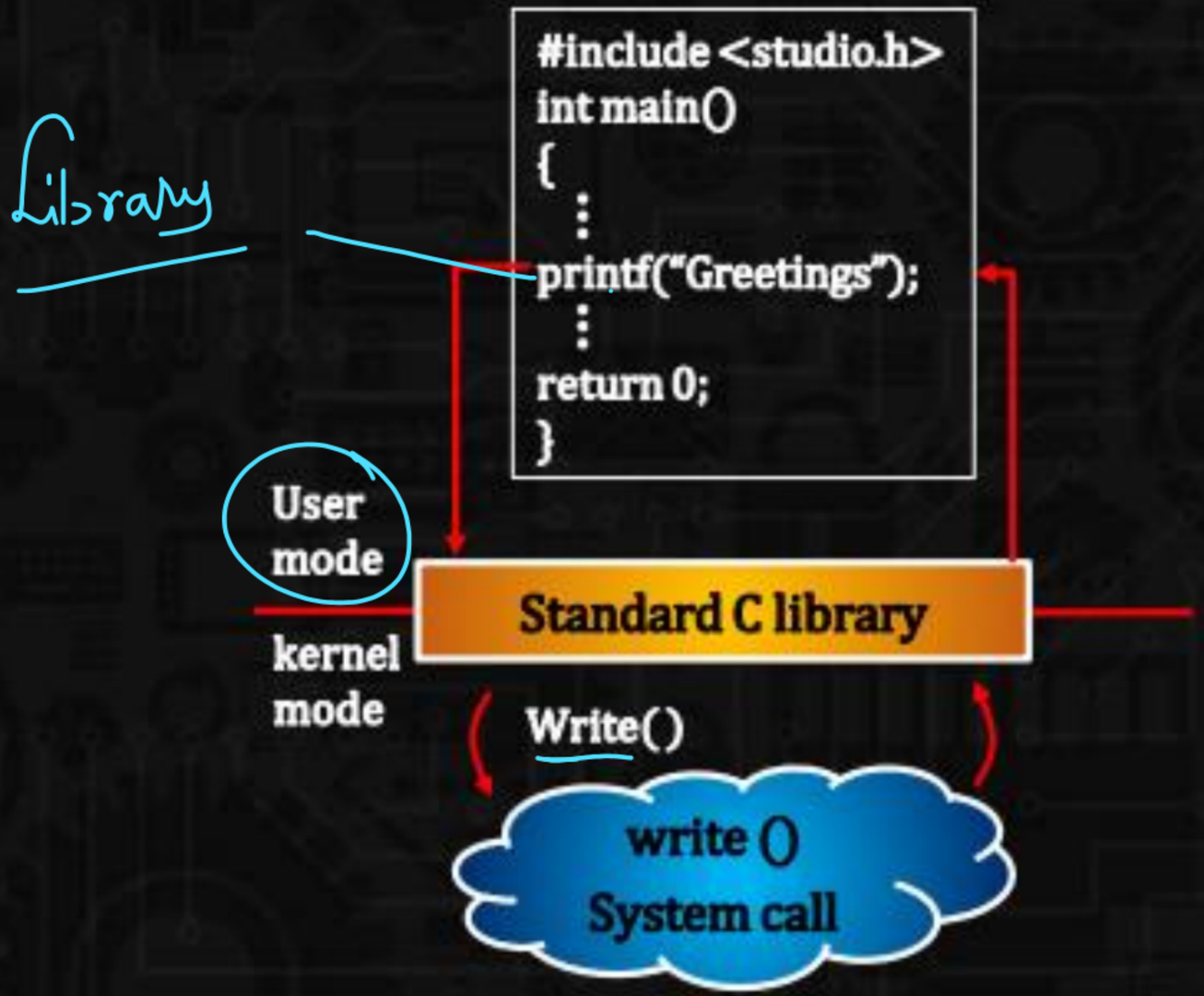
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about 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

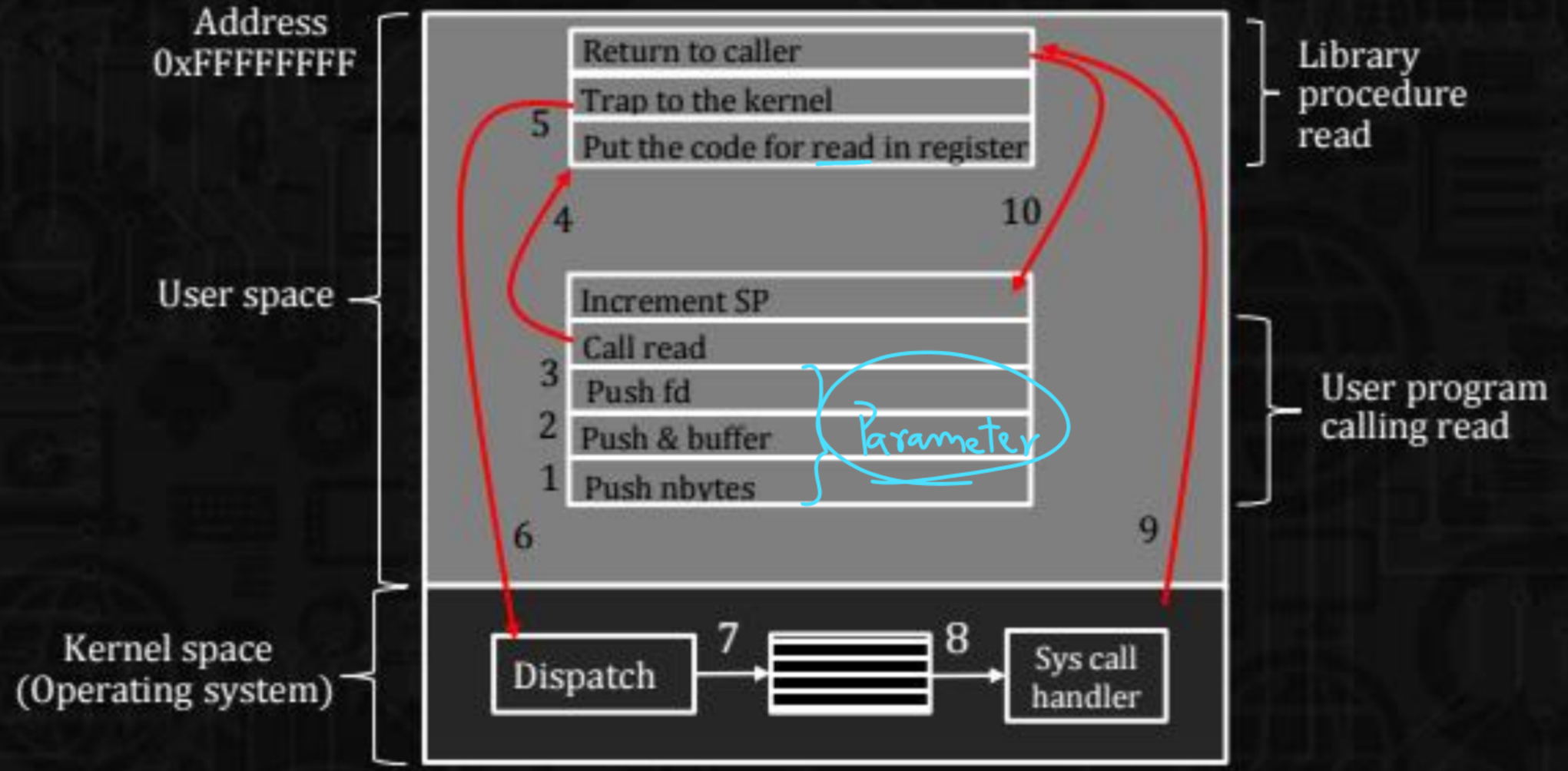


Standard C Library Example

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



Steps in Making a System Call

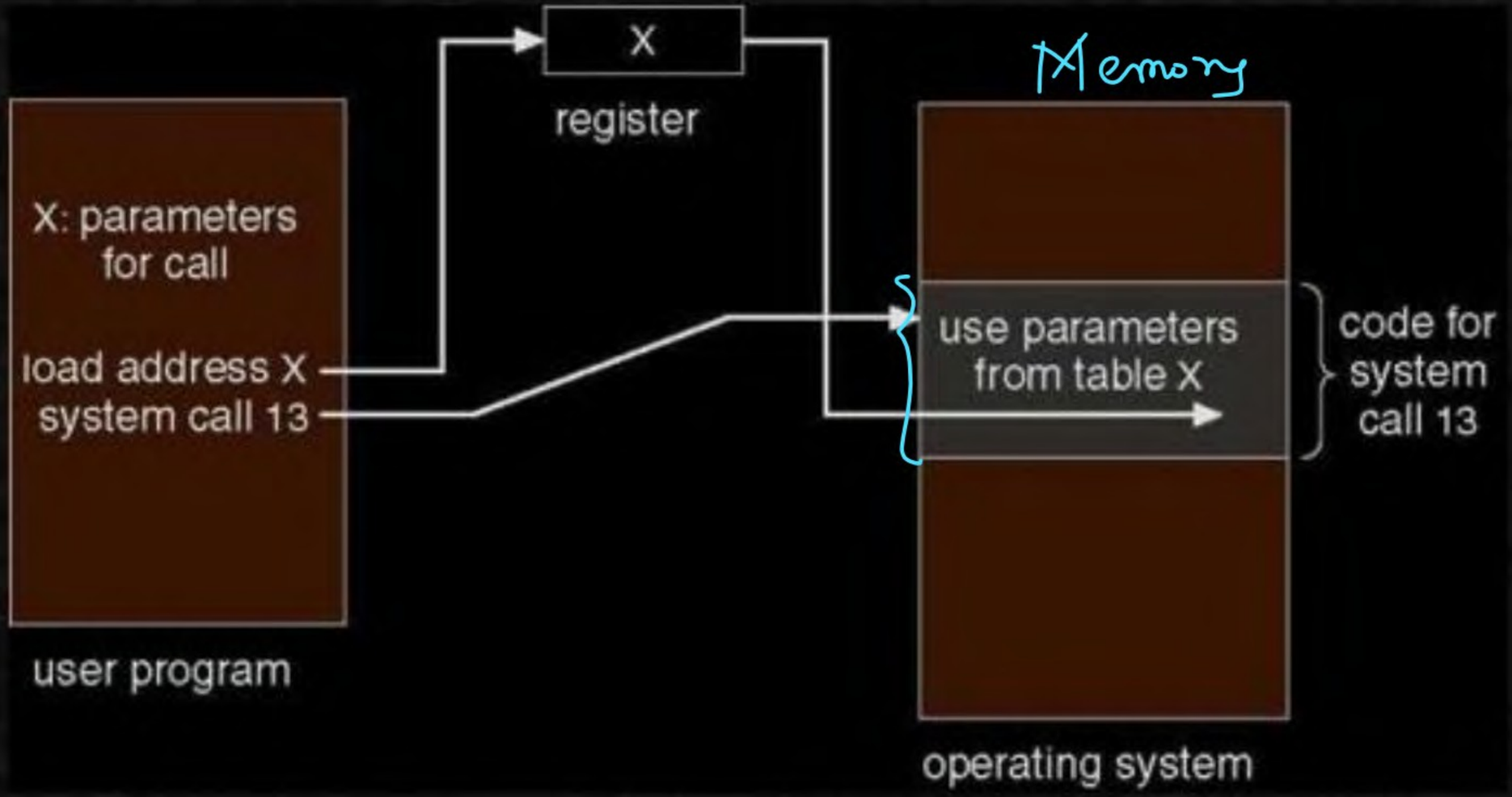


There are 11 steps in making the system call read (fd, buffer, nbytes)

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 (1)
    - 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 (2)
    - 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



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

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()

```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;
}
```

→ System Call failure

(fork() System Call Impl. after break (11:30 am)


THANK YOU
GW
SOLDIERS !