Operating Systems

Group 17:
106119002: Abhijeet Mishra
106119004: Abhinav Gupta
106119056: Dhruv Kachhadia
106119096: Priyanshu Dangi

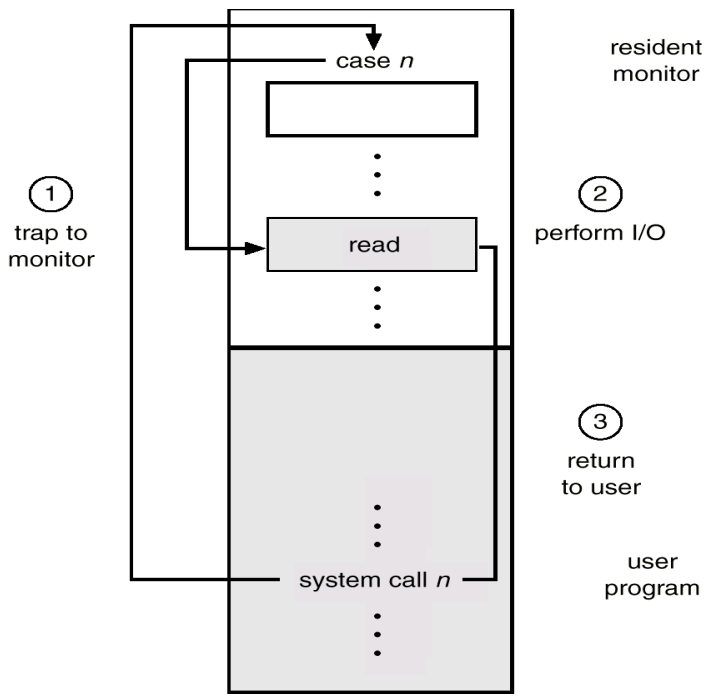
System Calls

System Calls

- System calls provide the interface between a running program and the operating system.
  - Generally available as assembly-language instructions.
  - Languages have been defined to replace assembly language for systems programming; allow system calls to be made directly (e.g., C, C++)
- Three general methods are used to pass parameters between a running program and the operating system.
  - Pass parameters in *registers*.
  - Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
  - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.

System Calls

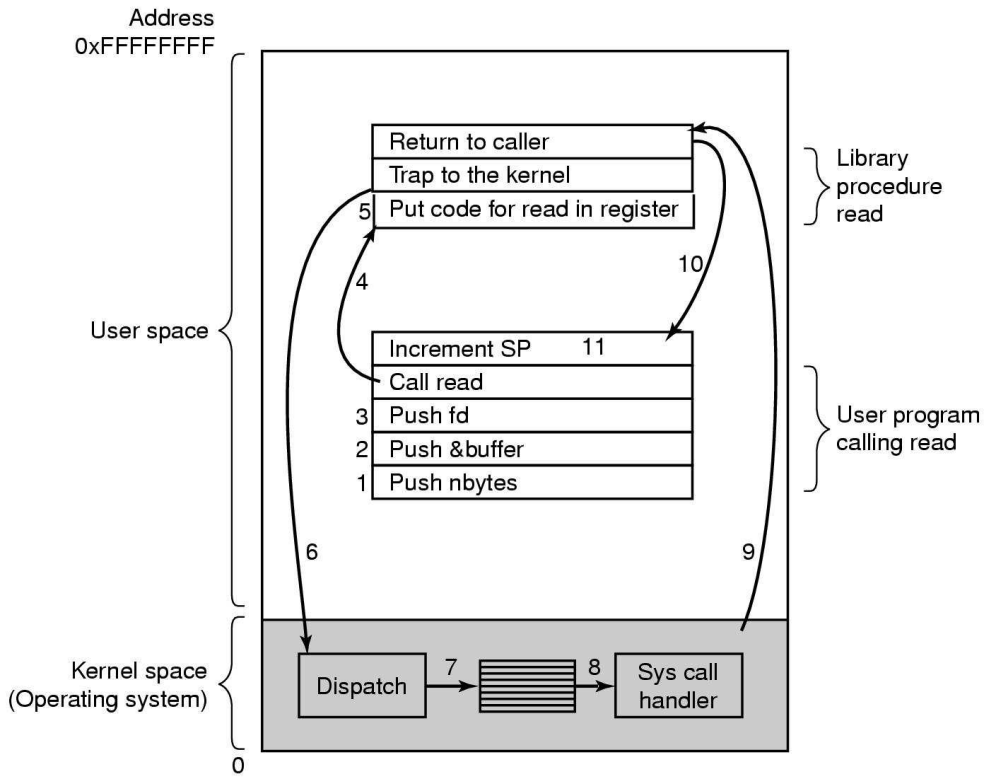
- A System Call is the main way a user program interacts with the Operating System.



System Calls

HOW A SYSTEM CALL WORKS

- Obtain access to system space
- Do parameter validation
- System resource collection (locks on structures)
- Ask device/system for requested item
- Suspend waiting for device
- Interrupt makes this thread ready to run
- Wrap-up
- Return to user



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

Linux API

System Call Implementation

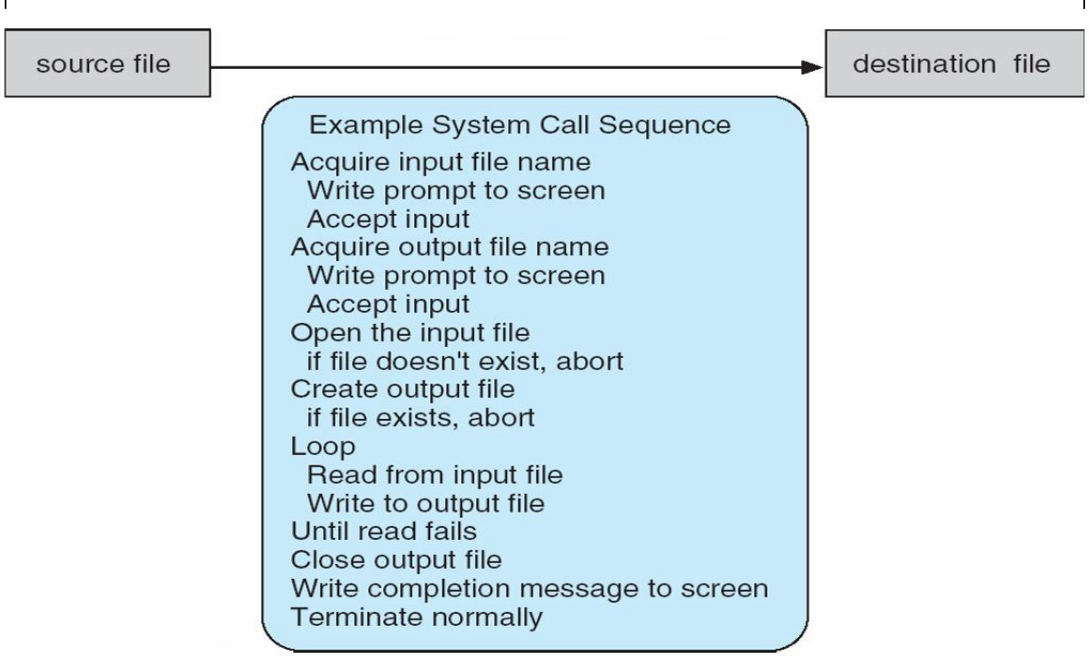
- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing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

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

Example of System Calls

- System call sequence to copy the contents of one file to another file



Types of System Calls

- Process control
- File manipulation
- Device manipulation
- Information maintenance
- Communications
- Protection

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - 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

- 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

- 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
  - 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

Gate Question

Which of the following standard C library functions will always invoke a system call when executed from a single-threaded process in a UNIX/Linux operating system?
(A) exit
(B) malloc
(C) sleep
(D) strlen

Ans - A, C

Process Management

Process Concepts

- The notion of a process is fundamental to OS and it defines the fundamental unit of computation for the computer and used by OS for concurrent program execution.

What is a process?

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

The components of process are:

- Object Program: Code to be executed
- Data: Data used for executing the program
- Resources: Resources needed for execution of the program
- Status of the process execution: Used for verifying the status of the process execution.
- A process is allocated with resources such as memory and is available for scheduling. It can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

A process includes: program counter; stack; data section

Process image

– Collection of programs, data, stack, and attributes that form the process

– User data
  - Modifiable part of the user space
  - Program data, user stack area, and modifiable code

– User program
  - Executable code

– System stack
  - Used to store parameters and calling addresses for procedure and system calls

– Process control block
  - Data needed by the OS to control the process

– Location and attributes of the process
  - Memory management aspects: contiguous or fragmented allocation

Keeping track of a process

- A process has code.
  - OS must track program counter (code location).
- A process has a stack.
  - OS must track stack pointer.
- OS stores state of processes' computation in a Process Control Block (PCB).
  - E.g., each process has an identifier (process identifier, or PID)
- Data (program instructions, stack & heap) resides in memory, metadata is in PCB (which is a kernel data structure in memory)

Implementation of a Process

Process Control Block (PCB)

- Each process contains the process control block (PCB). PCB is the data structure used by the OS and all information about a particular process such as Process id, process state, priority, privileges, memory management information, accounting information etc. are grouped by OS.
- PCB also includes the information about CPU scheduling, I/O resource management, file management information, priority and so on.
- The PCB simply serves as the repository for any information that may vary from process to process.

Contents of PCB are

1.Pointer: Pointer points to another process control block. Pointer is used for maintaining the scheduling list.

2. Process id

3. Process State: Process state may be new, ready, running, waiting and so on.

4. Program Counter: It indicates the address of the next instruction to be executed for this process.

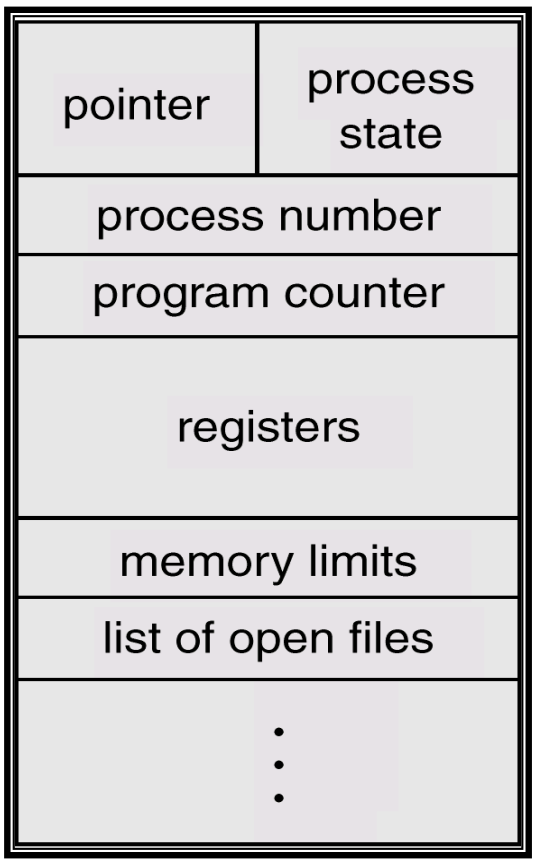
5. Event information: For a process in the blocked state this field contains information concerning the event for which the process is waiting.

6. CPU register: It indicates general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.

7. Memory Management Information: This information may include the value of base and limit register. This information is useful for deallocating the memory when the process terminates.

8. Accounting Information: This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.

Process Control Block (PCB)



- When a process is created, hardware registers and flags are set to the values provided by the loader or linker.
- Whenever that process is suspended, the contents of the processor register are usually saved on the stack and the pointer to the related stack frame is stored in the PCB.
- In this way, the hardware state can be restored when the process is scheduled to run again.

Operations on Processes

- The OS as well as other processes can perform operations on a process. Several operations are possible on the process such as create, kill, signal, suspend, schedule, change priority, resume etc.
- OS must provide the environment for the process operations.
- Two main operations that are to be performed are :
  - process creation
  - process deletion

Process Creation

OS creates the very first process when it initializes. That process than starts all other processes in the system using the create system calls.

OS creates a new process with the specified or default attributes and identifier.

A process may create several new sub-processes.

Syntax for creating new process is :

- CREATE (processid, attributes)

- When OS issues a CREATE system call, it obtains a new process control block from the pool of free memory, fills the fields with provided and default parameters, and insert the PCB into the ready list.
- Thus it makes the specified process eligible for running.
- When a process is created, it requires some parameters. These are priority, level of privilege, requirement of memory, access right, memory protection information etc.
- Process will need certain resources, such as CPU time, memory, files and I/O devices to complete the operation.

Process Hierarchy

- When one process creates another process, the creator is referred as parent and the created process is the child process. The Child process may create another process. So it forms a tree of processes which is referred as process hierarchy.
- When process creates a child process, that child process may obtain its resources directly from the OS, otherwise it uses the resources of parent process. Generally a parent is in control of a child process.

- In Operating System, the fork() system call is used by a process to create another process. The process that used the fork() system call is the parent process and process consequently created is known as the child process.

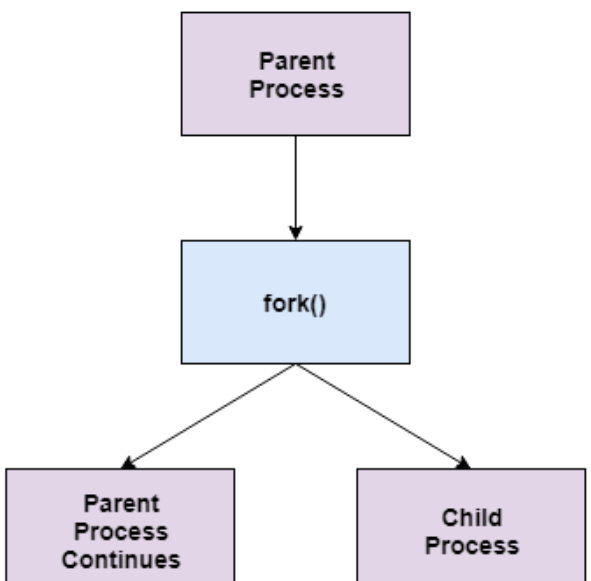
Parent Process

- All the processes in operating system are created when a process executes the fork() system call except the startup process. The process that used the fork() system call is the parent process. In other words, a parent process is one that creates a child process. A parent process may have multiple child processes but a child process only one parent process.

- On the success of a fork() system call, the PID of the child process is returned to the parent process and 0 is returned to the child process. On the failure of a fork() system call, -1 is returned to the parent process and a child process is not created.

Child Process

- A child process is a process created by a parent process in operating system using a fork() system call. A child process may also be called a subprocess or a subtask.
- A child process is created as its parent process's copy and inherits most of its attributes. If a child process has no parent process, it was created directly by the kernel.
- If a child process exits or is interrupted, then a SIGCHLD signal is send to the parent process.


Parent Process
fork()
Parent Process Continues
Child Process

When a process creates a new process, two possibilities exist in terms of execution.

1. Concurrent : The parent continues to execute concurrently with its children.

2. Sequential :The parent waits until some or all of its children have terminated.

- For address space, two possibilities:
    1. The child process is a duplicate of the parent process.
    2. The child process has a program loaded into it.
- Resource sharing possibilities
    1. Parent and child share all resources
    2. Children share subset of parent’s resources
    3. Parent and child share no resources

Process Termination

- DELETE system call is used for terminating a process.
- A process may delete itself or by another process. A process can cause the termination of another process via an appropriate system call.
- The OS reacts by reclaiming all resources allocated to the specified process, closing files opened by or for the process. PCB is also removed from its place of residence in the list and is returned to the free pool.
- The DELETE service is normally invoked as a part of orderly program termination. Parent may terminate execution of children processes (abort) if Child has exceeded allocated resources or Task assigned to the child is no longer required or parent is exiting.
- OS may not allow child to continue if its parent terminates. This may result in cascading termination.

Zombie Process:

A child process which has finished the execution but still has entry in the process table goes to the zombie state.

A child process always first becomes a zombie before being removed from the process table.

The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

Orphan Process:

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

Static and Dynamic Process:

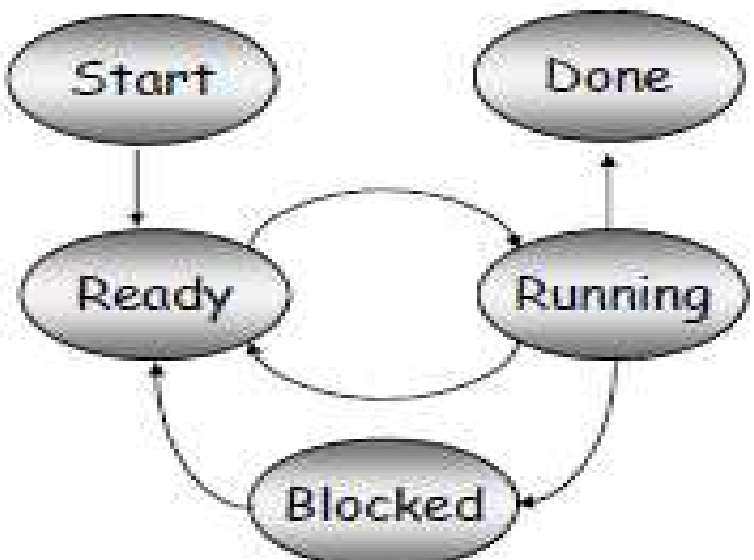
- A process that does not terminate while the OS is functioning is called static.
- A process that may terminate is called dynamic.

Life Cycle of a Process

When process executes, it changes state. Process state is defined as the current activity of the process.

Once created a process can be in any of the following three basic states:

- Ready: The process is ready to be executed, but CPU is not available for the execution of this process.
- Running: The CPU is executing the instructions of the corresponding process. A running process possesses all the resources needed for its execution, including the processor.
- Blocked/ Waiting: The process is waiting for an event to occur.
  - The event can be I/O operation to be completed
  - Memory to be made available
  - A message to be received etc.


Start
Done
Ready
Running
Blocked

- A running process gets blocked because of a requested resource is not available or can become ready because of the CPU decides to execute another process.
- A blocked process becomes ready when the needed resource becomes available to it. A ready process starts running when the CPU becomes available to it.
- Whenever processes changes state, the OS reacts by placing the process PCB in the list that corresponds to its new state. Only one process can be running on any processor at any instant and many processes may be in ready and waiting state.

Gate Question

Consider the following statements about process state transitions for a system using preemptive scheduling.

I. A running process can move to ready state.
II. A ready process can move to ready state.
III. A blocked process can move to running state.
IV. A blocked process can move to ready state.

Which of the above statements are TRUE?
A I, II and III only
B II and III only
C I, II and IV only
D I, II, III and IV

Ans C

Gate Question

Consider four processes P, Q, R and S scheduled on a CPU as per round robin algorithm with a time quantum of 4 units. The processes arrive in the order P, Q, R, S, all at time t = 0. There is exactly one context switch from S to Q, exactly one context switch from R to Q, and exactly two context switches from Q to R. There is no context switch from S to P. Switching to a ready process after the termination of another process is also considered a context switch. Which one of the following is NOT possible as CPU burst time (in time units) of these processes?

A) P = 4, Q = 10, R = 6, S = 2
B) P = 2, Q = 9, R = 5, S = 1
C) P = 4, Q = 12, R = 5, S = 4
D) P = 3, Q = 7, R = 7, S = 3

Ans D

Threads

Threads

- Thread: single sequential flow of control within a program
- Single-threaded program can handle one task at any time.
- Multitasking allows single processor to run several concurrent threads.
- Most modern operating systems support multitasking.

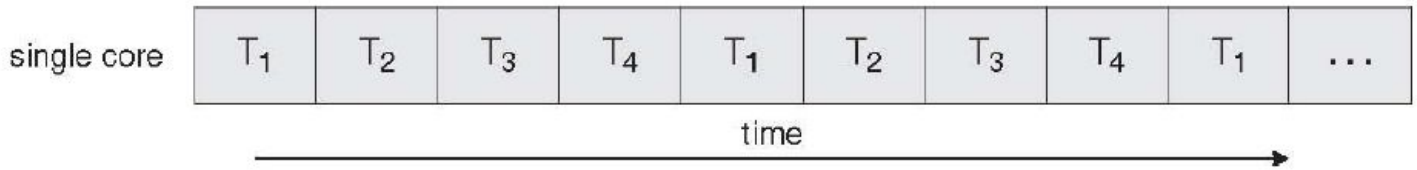
Thread Concepts

- Traditionally a process has a single address space and a single thread of control to execute a program within that address space.
- To execute a program, a process has to initialize and maintain state information.
- The state information is comprised of page tables, swap image, file descriptors, outstanding I/O requests, saved register values etc. This information is maintained on a per program basis and thus a per process basis.
- The volume of this information makes it expensive to create and maintain processes as well as to switch between them.
- Threads or light weight processes have been proposed to handle situations where creating, maintaining and switching between processes occur frequently.

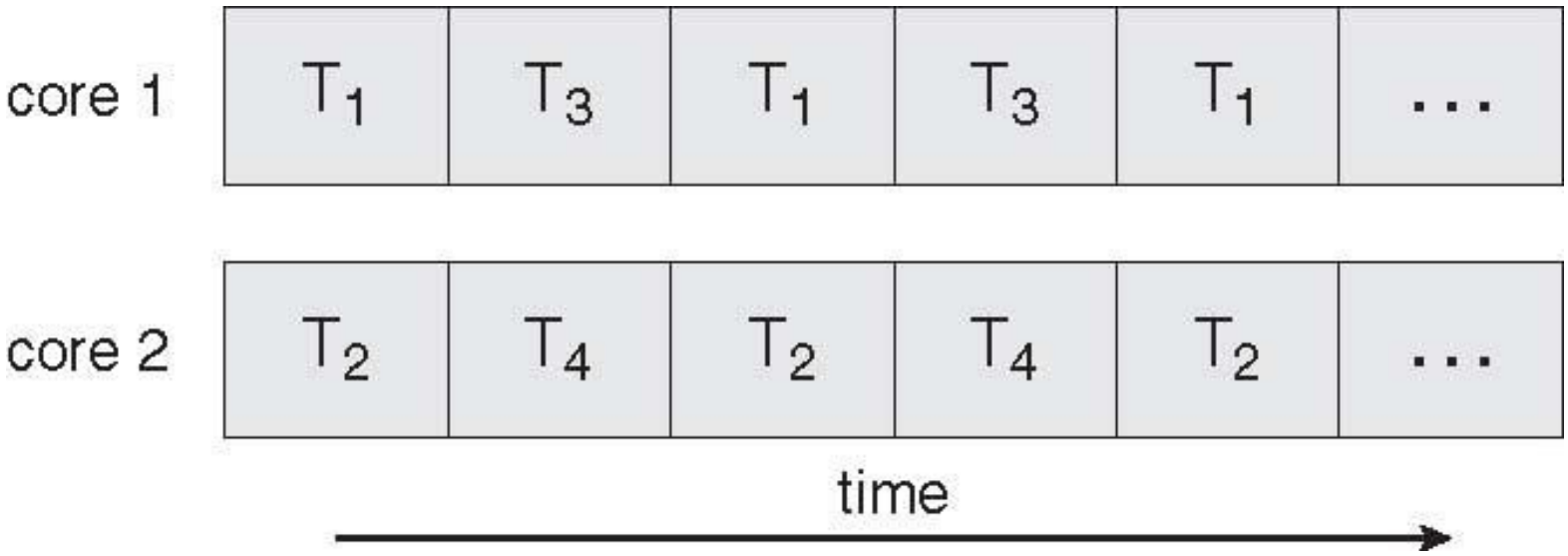
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

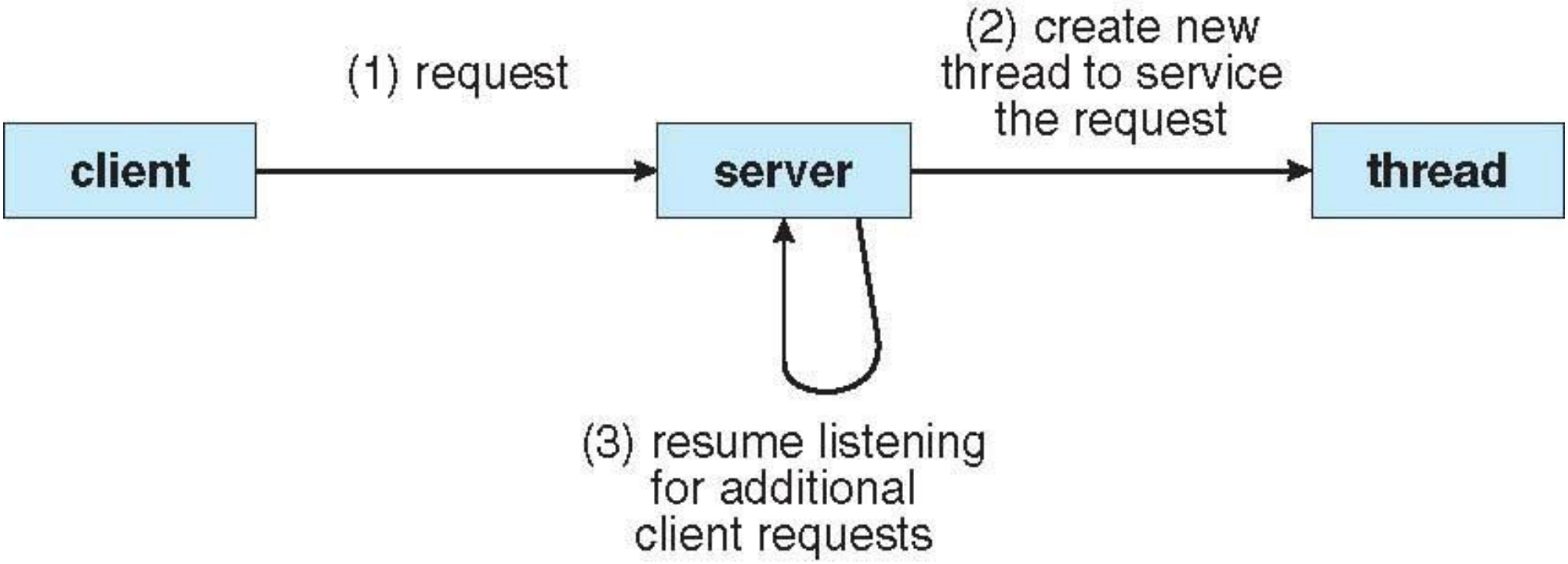
Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



Multithreaded Server Architecture



Threads Vs Process

• A process is an abstraction for representing resource allocation.

• A thread is an abstraction for execution: a thread represents the execution of a particular sequence of instructions in a program's code, or equivalently a particular path through the program's flow of control.

• A process may have multiple threads, each sharing the resources allocated to the process.

Threads vs Processes

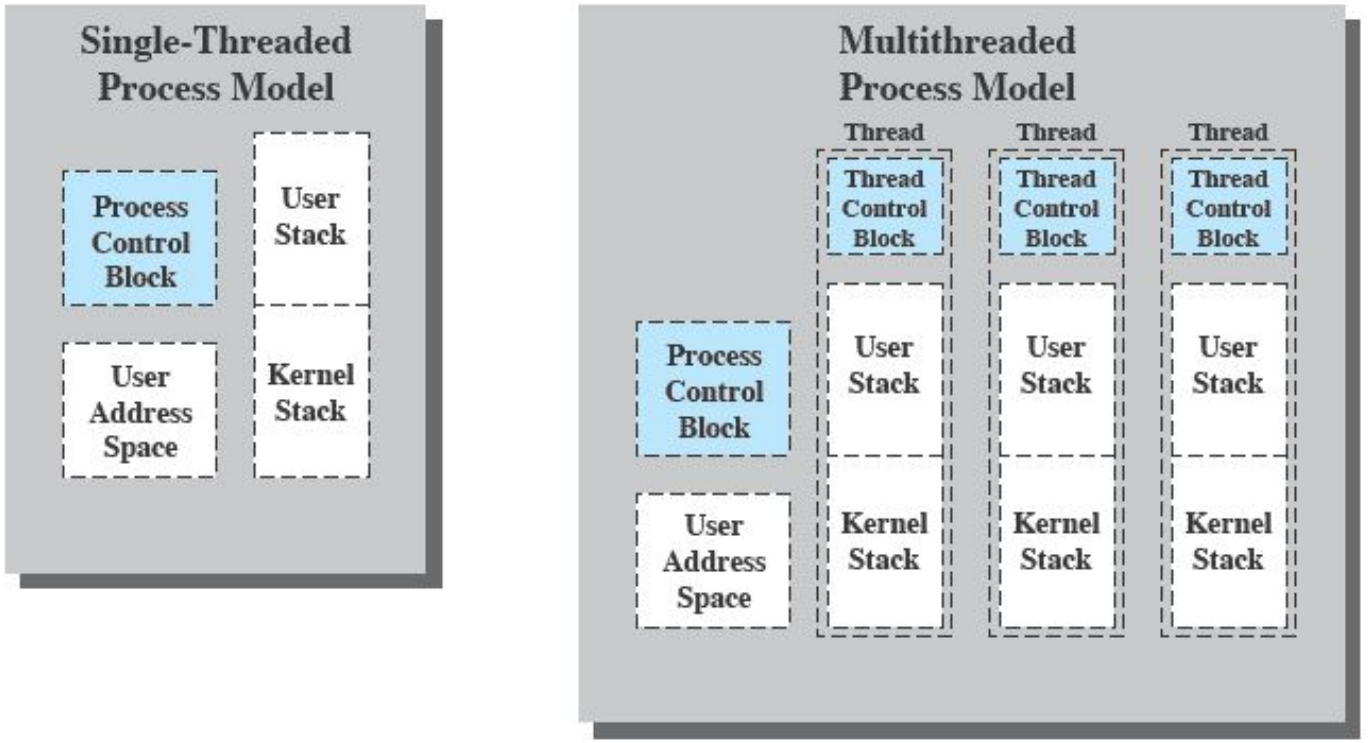
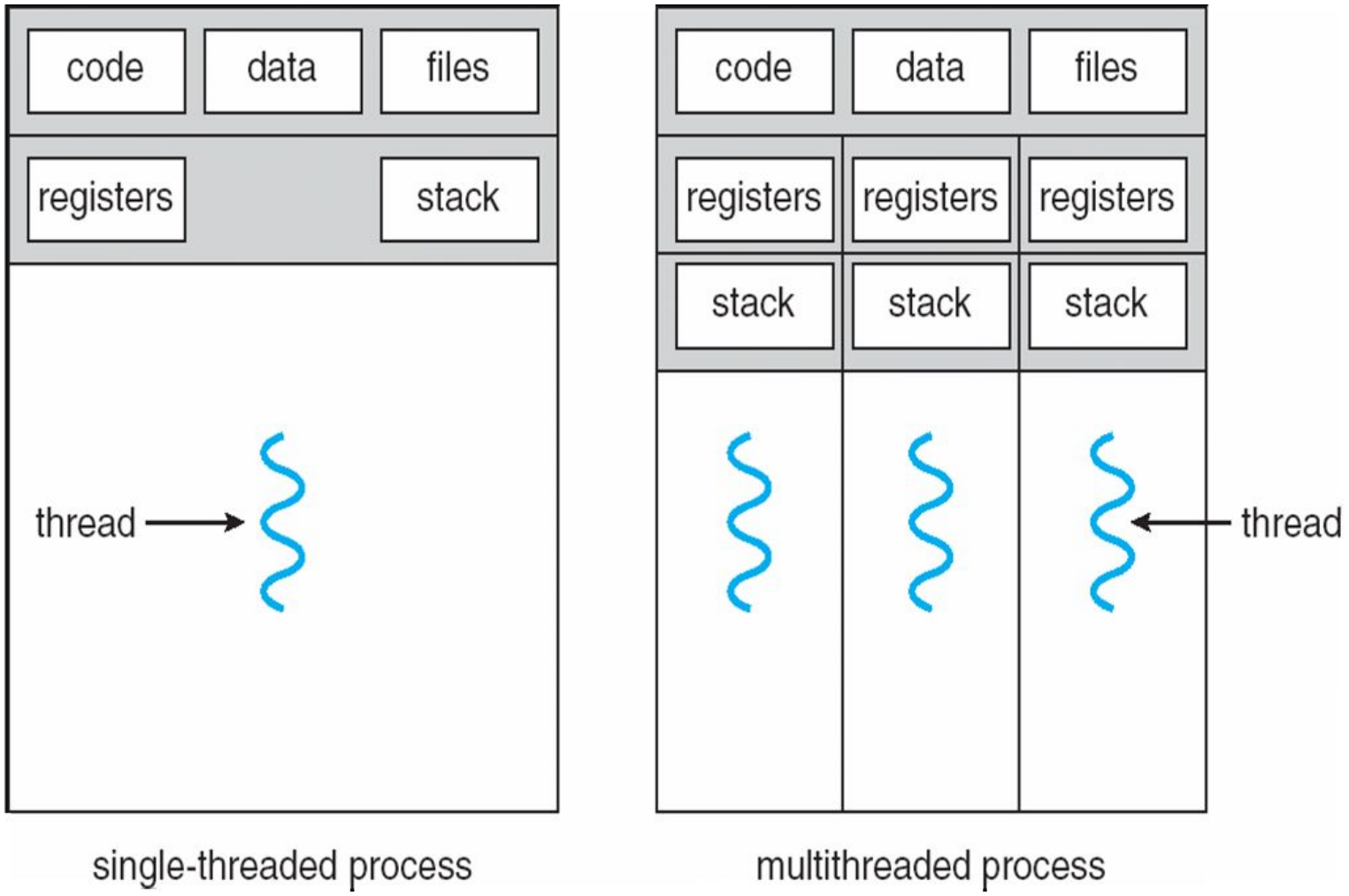


Figure 4.2 Single Threaded and Multithreaded Process Models

Single and Multithreaded Processes



- A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space. Each thread makes use of a separate program counter and a stack of activation records and a thread control block.
- The control block contains the state information necessary for thread management, such as putting a thread into a ready list and for synchronizing with other threads.
- Threads share all resources (memory, open files, etc.) of their parent process *except* the CPU.

Each thread requires its own context:

- program counter
- stack
- Registers

Each Thread has

- an execution state (Running, Ready, etc.)
- saved thread context when not running (TCB)
- an execution stack
- some per-thread static storage for local variables
- access to the shared memory and resources of its process (all threads of a process share this)

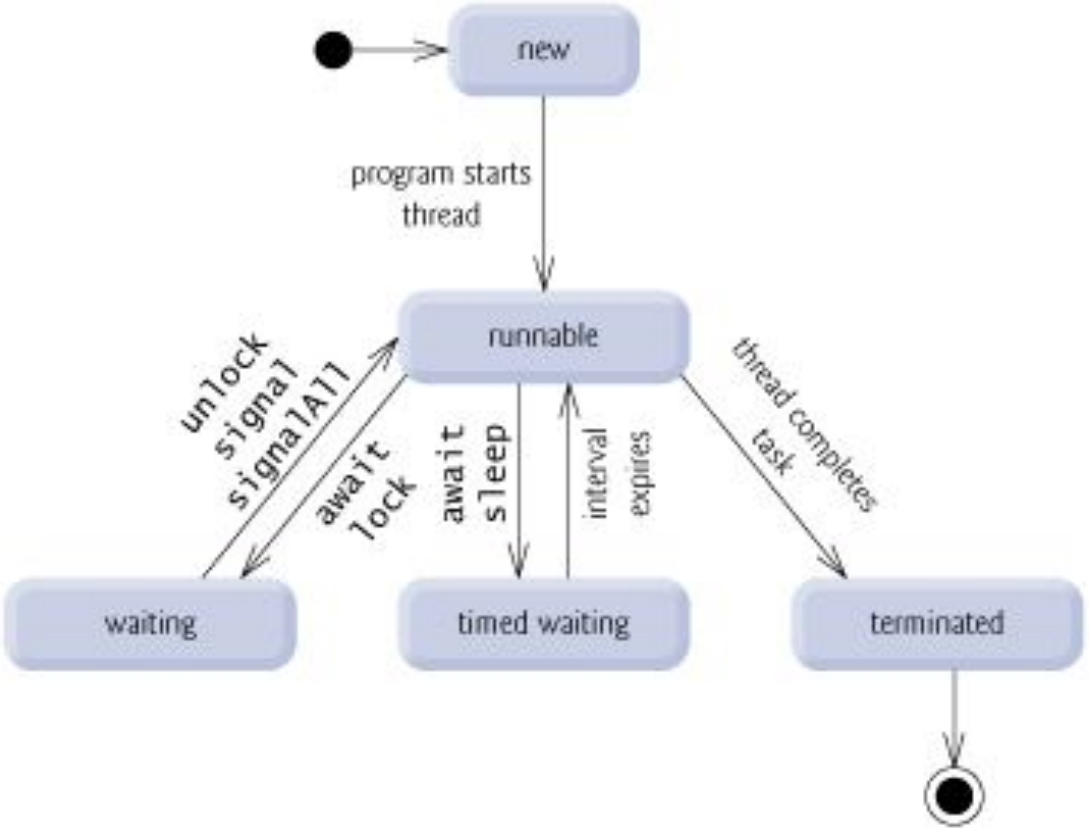
Thread Execution States

Similar to a process a thread can be in any of the primary states: Running, Ready and Blocked.

The operations needed to change state are:

- Spawn: new thread provided register context and stack pointer.
- Block: event wait, save user registers, PC and stack pointer
- Unblock: moved to ready state
- Finish: deallocate register context and stacks.

Thread States



Thread Scheduling

- An operating system's thread scheduler determines which thread runs next.
- Most operating systems use *timeslicing* for threads of equal priority.
- *Preemptive scheduling*: when a thread of higher priority enters the running state, it preempts the current thread.
- *Starvation*: Higher-priority threads can postpone (possible forever) the execution of lower-priority threads.

Common Thread Models

- User level threads

  These threads implemented as user libraries. Thread library provides programmer with API for creating and managing threads. Benefits of this are no kernel modifications, flexible and low cost. The drawbacks are thread may block entire process and no parallelism.

- Kernel level threads

  Kernel directly supports multiple threads of control in a process. Benefits are scheduling/synchronization coordination, less overhead than process, suitable for parallel application. The drawbacks are more expensive than user-level threads and more overhead.

- Light-Weight Processes (LWP)

  It is a kernel supported user thread. LWP bound to kernel thread but a kernel thread may not be bound to an LWP. It is scheduled by kernel and user threads scheduled by library onto LWPs, so multiple LWPs per process.

Multithreading

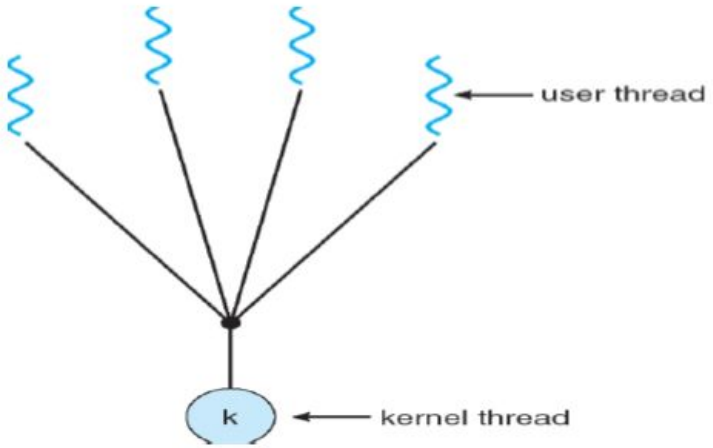
Kernels are generally multi-threaded.

Multi-threading models include

- Many-to-One: Many user-level threads mapped to single kernel thread
- One-to-One: Each user-level thread maps to kernel thread
- Many-to-Many: Many user-level threads mapped to many kernel threads.

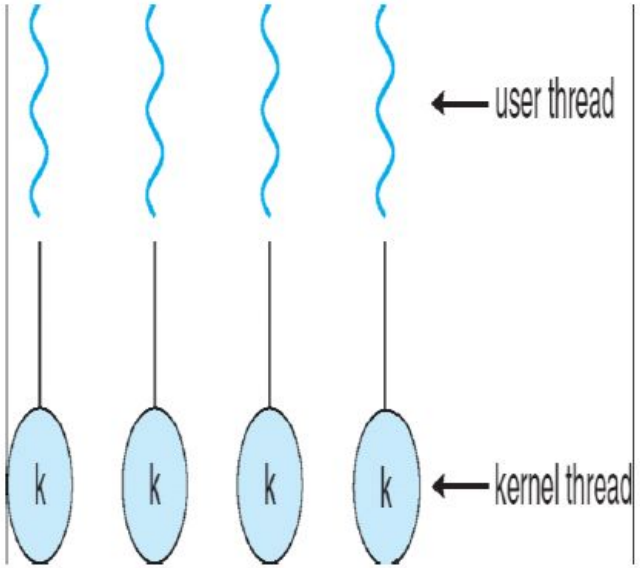
Many-to-One

- Thread management is done by the thread library in user space
- The entire process will block if a thread makes a blocking system call
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Mainly used in language systems, portable libraries
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



- Advantages:
  - totally portable
  - easy to do with few systems dependencies
- Disadvantages:
  - cannot take advantage of parallelism
  - may have to block for synchronous I/O

One-to-One

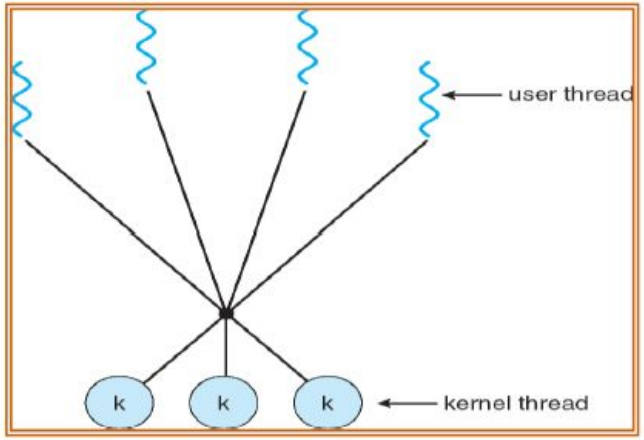


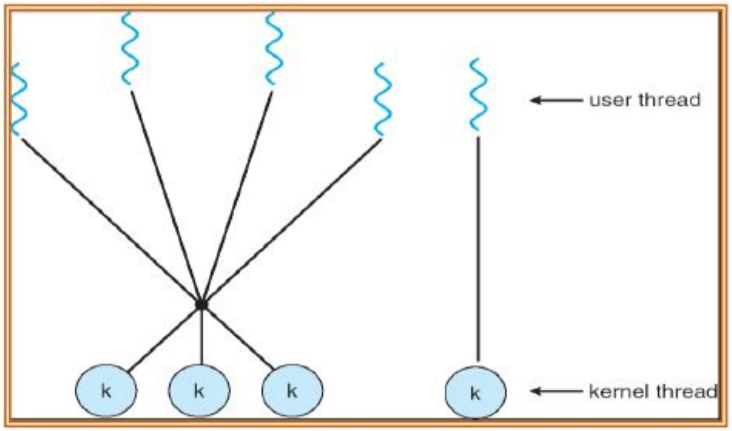
- Each **user-level thread** maps to **kernel thread**
  - Creating a **user-level** thread **creates** a **kernel thread**
  - *This is a drawback to this model; creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may **burden** the performance of a system.*
- More concurrency than many-to-one
  - Allows another thread to run when a thread makes a blocking system call
  - It also allows multiple threads to run in parallel on multiprocessors.
- Number of threads per process sometimes restricted due to overhead
- Used in LinuxThreads and other systems where LWP creation is not too expensive

- Advantages:
  - can exploit parallelism, blocking system calls
- Disadvantages:
  - thread creation involves LWP creation
  - each thread takes up kernel resources
  - limiting the number of total threads

Many-to-many

- In this model, the library has two kinds of threads: *bound* and *unbound*
  - bound threads are mapped each to a single lightweight process
  - unbound threads *may* be mapped to the same LWP





- Issues in multithreading include
  - Thread Creation
  - Thread Cancellation
  - Signal Handling (synchronous / asynchronous),
  - Handling thread-specific data and scheduler activations.

Thread pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled

GATE | GATE CS 2011 |

A thread is usually defined as a “light weight process” because an operating system (OS) maintains smaller data structures for a thread than for a process. In relation to this, which of the following is TRUE?

(A) On per-thread basis, the OS maintains only CPU register state

(B) The OS does not maintain a separate stack for each thread

(C) On per-thread basis, the OS does not maintain virtual memory state

(D) On per-thread basis, the OS maintains only scheduling and accounting information

Answer: (C)

Explanation: Threads share address space of Process. Virtually memory is concerned with processes not with Threads.

Interprocess Communication

Message passing systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment where the communicating processes may reside on different computers connected by a network.
- A message passing facility provides at least two operations:

  send (message) and receive (message).

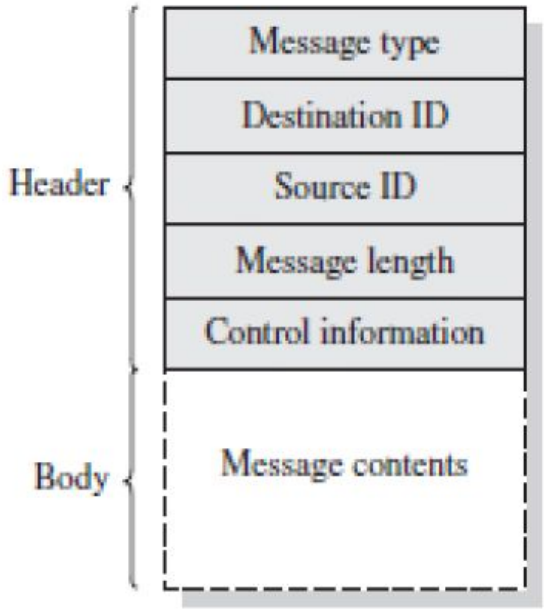
Issues related to message passing systems

- Message size:

Messages sent by a process can be of either fixed or variable size. If only fixed - sized messages can be sent, the system – level implementation is straight forward.

Variable sized messages require a more complex system level implementation.

Message Format



- Communication link:

If two processes want to communicate, they must send messages to and receive messages from each other, a communication link must exist between them.

A communication link has the following properties :

A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

A link is associated with exactly two processes.

Between each pair of processes, there exists exactly one link.

Direct or indirect communication

Naming: Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. The send () and receive() primitives are defined as:

Send (P, message) – send a message to process P

Receive (Q, message) – receive a message from process Q

This scheme exhibits symmetry in addressing; that is both the sender process and receiver process must name the other to communicate.

A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.

Here, the send () and receive () primitives are defined as:

Send (P, message) – send a message to process P

Receive (id, message) – receive a message from any process, the variable id is set to the name of the process with which communication has taken place.

With indirect communication, the messages are sent to and received from mail boxes or ports.

- A mail box can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mail box has a unique identification.
- A process can communicate with some other process via a number of different mail boxes.

Two processes can communicate only if the processes have a shared mail box.

The send () and receive() primitives are defined as:

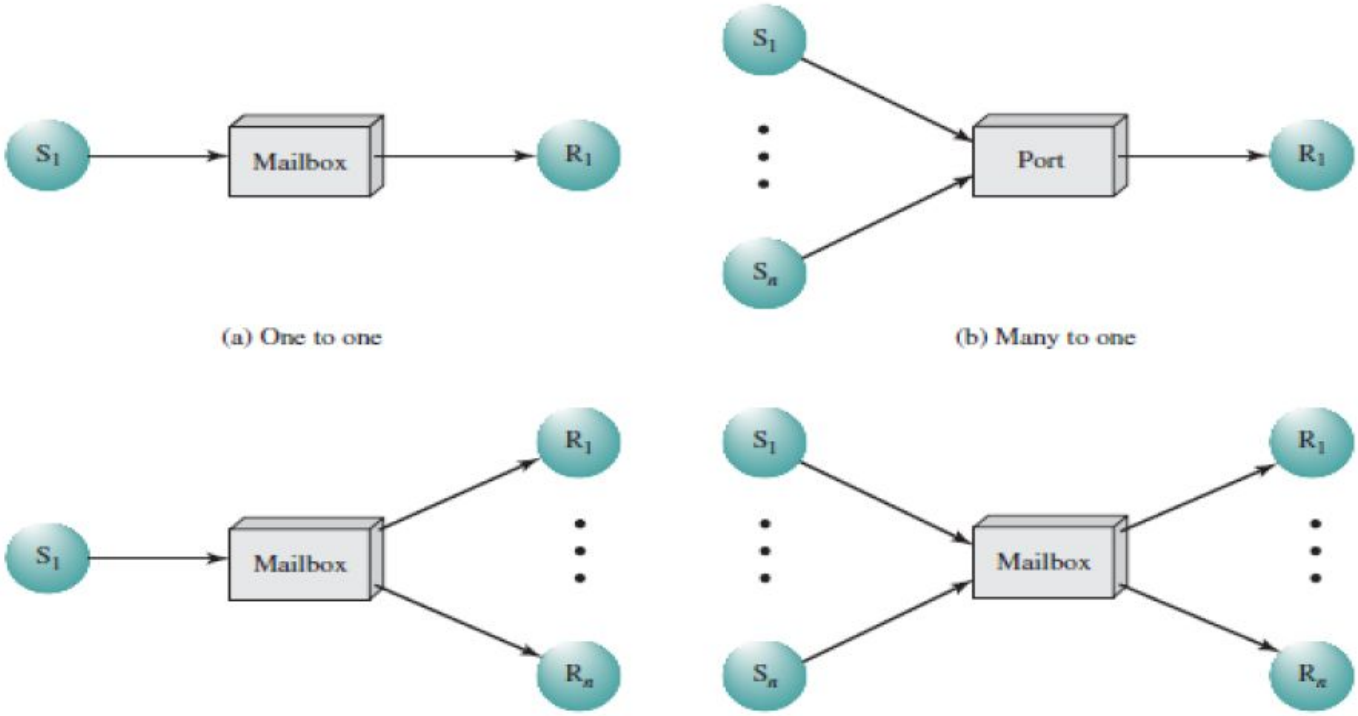
- Send (A, message) – send a message to mail box A
- Receive (A, message) – receive a message from mail box A.

In this scheme, a communication link has the following properties –

A link is established between a pair of processes only if both members of the pair have a shared mail box.

A link may be associated with more than two processes.

Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mail box.


S1
Mailbox
R1
(a) One to one
S1
Sn
Port
R1
(b) Many to one
S1
Mailbox
R1
Rn
(c) One to many
S1
Sn
Mailbox
R1
Rn
(d) Many to many

A mail box may be owned either by a process or by the OS.

- If the mail box is owned by a process, then we distinguish between the owner and the user. When a process that owns the mail box terminates, the mail box disappears. Any process that subsequently sends a message to this mail box must be notified that the mail box no longer exists.
- A mail box owned by the OS is independent and is not attached to any particular process. The OS then must provide a mechanism that allows a process to do the following:
  - Create a new mail box
  - Send and receive messages through the mail box
  - Delete a mail box

Synchronous or asynchronous communication

Communication between processes takes place through calls to send () and receive () primitives. Message passing may be either blocking or non blocking – also known as synchronous and asynchronous.

- Blocking send: The sending process is blocked until the message is received by the receiving process or the mail box.
- Non blocking send: The sending process sends the message and resumes operation.
- Blocking receive: The receiver blocks until a message is available.
- Non blocking receive: The receiver retrieves either a valid message or a null.

- When both send () and receive () are blocking, we have a rendezvous between the sender and the receiver.

Automatic or explicit buffering

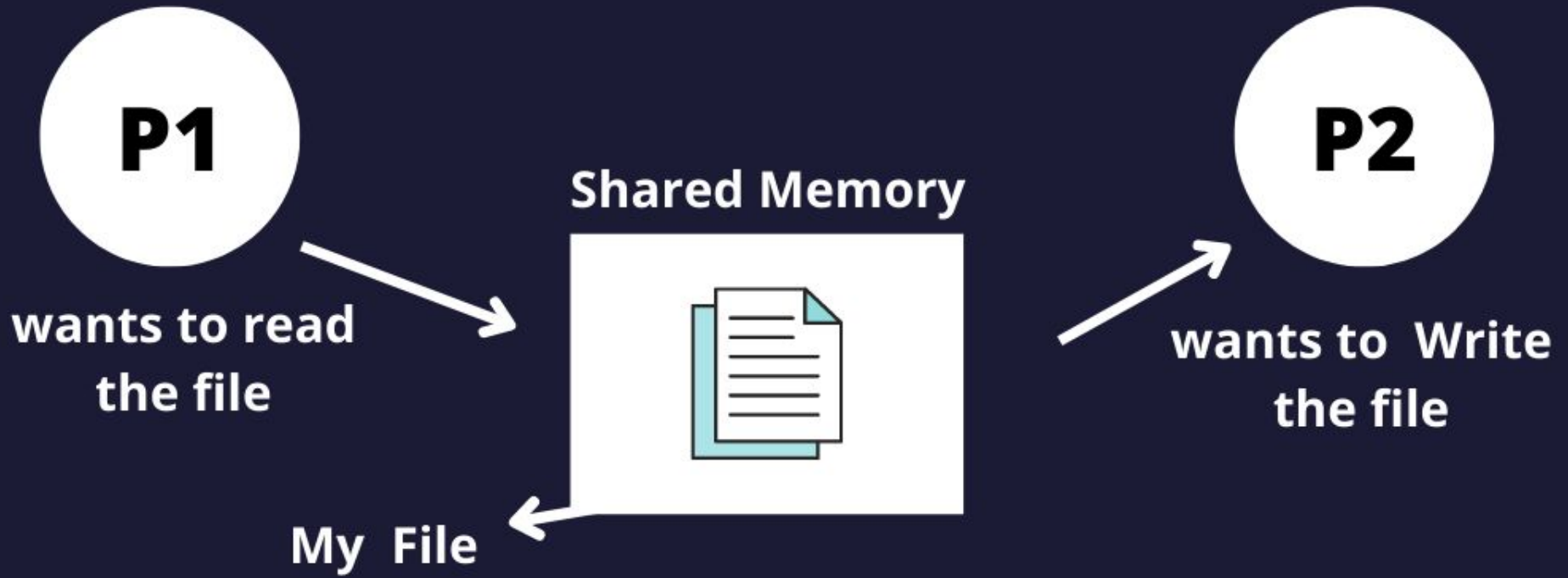
- Buffering: Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways –
  - Zero capacity: The queue has the maximum length of zero; thus the link cannot have any messages waiting in it.
  - Bounded capacity: The queue has finite length n; thus, at most n messages can reside in it.
  - Unbounded capacity: The queues length is infinite; thus any number of messages can wait in it. The sender never blocks.
- The zero capacity buffer is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

Concurrency and Synchronization

Synchronization

Synchronization is a process in which various jobs which share the common memory space are handled by the Operating System. Subsequently, it is used for maintaining the consistency of the system.

Process Synchronization



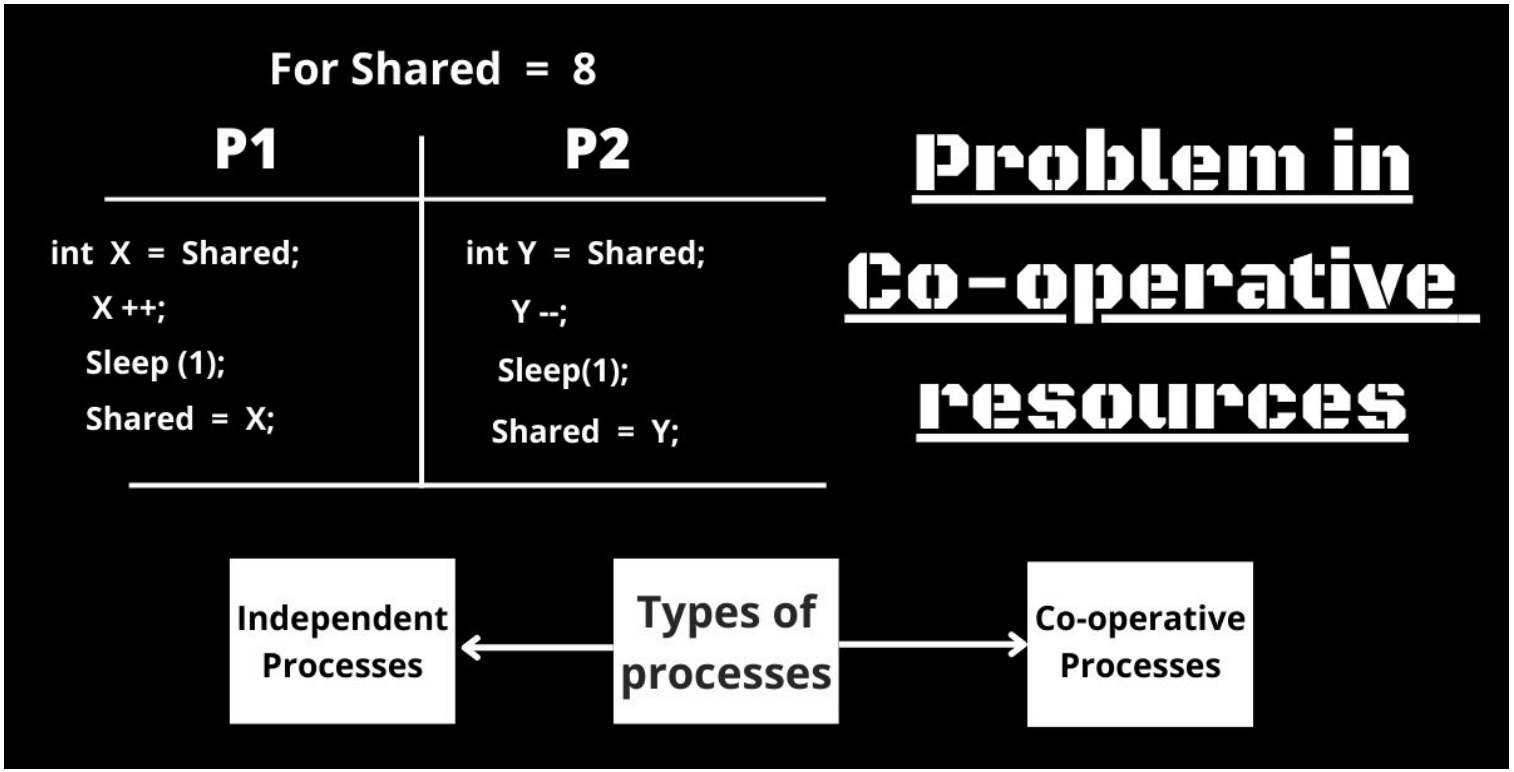
Visit - www.ahadsprogrammingclub.com

TYPES OF PROCESSES

There are two types of **processes** and both are described below –

- **Co-operative Process** – In the Co-operative process, one execution affects the other because they share something like memory, buffer, code, or resources.
- **Independent Process** – In Independent Process, the execution of the process will not affect another. They do not share anything and also don’t have anything in common.

The co-operative processes can occur problems if not executed properly. for instance, ref. image next slide –



However, the process runs parallel, and if the CPU Grants to P1 when P1 sleep, then P2.

It is wrong because the value of the shared changes to 4 but it must be 5. The processes are not synchronized and the condition is known as the **RACE Condition**.

Concurrency

The points which describe Concurrency are mentioned below –

Able to run multiple applications at the same time.

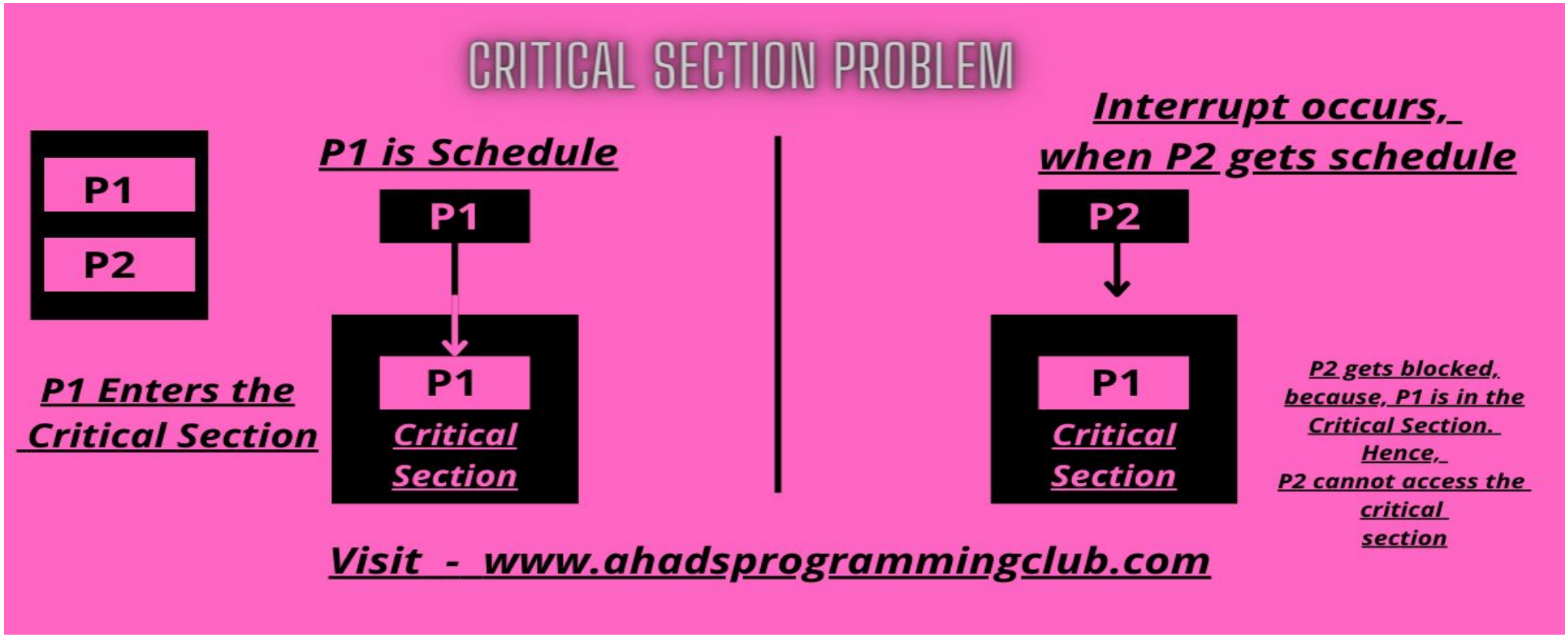
Allow better resource utilization.

Better average response time of applications.
Achieves better performance.

There are various problems based on Concurrency and Synchronization and all are practically implemented further.

CRITICAL SECTION PROBLEM

The **Critical Section** is the program where the shared resources are accessed by various processes or can be said it is the portion of a program text where the shared resources are accessed.



To achieve the **synchronization** the following four conditions must have to be met –

- **Mutual Exclusion** – One process at a time in the **critical section**.
- **Progress** – No Process can stop another process to access the **critical section**.
- **Bounded Wait** – No process must not continuously keep on accessing the **Critical Section** and other processes keep waiting.
- **No Assumption related to Hardware** – There must not be a limitation. Suppose, a program running on 64 bits Operating System must also run on 32 bits.

SEMAPHORE

Semaphore is a special variable whose value indicates the information about the locks. Further, there are two types of semaphores –

- **Counting Semaphore** (-infinity to +infinity)
- **Binary Semaphore** (0, 1)

The **Semaphore** is an integer that is used in a mutually exclusive manner by various concurrent co-operative processes to achieve **synchronization** and also it is used to remove the **Race Condition**.

DINING PHILOSOPHER PROBLEM

The **Dining philosopher Problem** is an important and yet another **synchronization** problem that states that there is a dining table and one rice bowl along with the five philosophers and Five forks.

Subsequently, achieving **synchronization** is the motive of the problem.

The code of the **Dining Philosopher Problem** is as follows –

```
void(philosopher)
{
   while(true)
   {
      Thinking();
      take_fork(i);
      take_fork((i+1)%N)
      EAT();
      put Fork(i);
      put Fork((i+1)%N);
   }
  }
```

If more than one philosopher arrives in real-time, then the **race condition** occurs. So, to remove it we use the concept of **Binary semaphore**.

Consider three concurrent processes P1, P2 and P3 as shown below, which access a shared variable D that has been initialized to 100.The process are executed on a uniprocessor system running a time-shared operating system. If the minimum and maximum possible values of D after the three processes have completed execution are X and Y respectively, then the value of Y−X is _____.

P1	P2	P3
: : D = D + 20 : :	: : D = D – 50 : :	: : D = D + 10 : :

A) 80
B) 20
C) 70
D) 50

Answer

Correct answer is 80 to 80

Explanation

Case 1 :

Let initially P2 reads D = 100 and then got preempted.

Then P1 executes D = D + 20, so now D = 120.

Then P3 executes D = D + 10, so now D = 130.

Initially when P2 got preempted the value of D was = 100.

Now P2 executes D = D - 50 = 100 - 50 = 50

P2 writes D = 50 final value. This is minimum

When the result of a computation depends on the speed of the processes involved there is said to be

A) Cycle Stealing
B) Race Condition
C) Time Clock
D) Deadlock

Answer : B

A critical section is a program segment

A) Which should run in specified
B) Which avoids deadlocks
C) Where shared resources are accessed
D) Which must be enclosed by semaphore

ANSWER: C