CO - 2

COURSE NAME : SYSTEM DESIGN AND INTRODUCTION TO CLOUD

COURSE CODE : 23AD2103A

TOPICS : CONCURRENCY: CONCURRENCY AND THREADS CODE, THREAD API, COMMON CONCURRENCY PROBLEMS

SESSION DESCRIPTION

- Concurrency
- Concurrency and Threads code
- Thread API
- Common concurrency problems

CONCURRENCY IN OS

- Concurrency in operating systems refers to the ability of an operating system to handle multiple tasks or processes at the same time.
- With the increasing demand for high performance computing, concurrency has become a critical aspect of modern computing systems.
- Concurrency is essential in modern operating systems due to the increasing demand for multitasking, real-time processing, and parallel computing.
- Operating systems that support concurrency can execute multiple tasks simultaneously, leading to better resource utilization, improved responsiveness, and enhanced user experience.

PRINCIPLES OF CONCURRENCY

- The principles of concurrency in operating systems are designed to ensure that multiple processes or threads can execute efficiently and effectively

INTERLEAVING

- Interleaving refers to the interleaved execution of multiple processes or threads.

SYNCHRONIZATION

- Synchronization refers to the coordination of multiple processes or threads to ensure that they do not interfere with each other.

PRINCIPLES OF CONCURRENCY

MUTUAL EXCLUSION

- Mutual exclusion refers to the principle of ensuring that only one process or thread can access a shared resource at a time.

DEADLOCK AVOIDANCE

- Deadlock is a situation in which two or more processes or threads are waiting for each other to release a resource, resulting in a deadlock.

PROCESS OR THREAD COORDINATION

- Processes or threads may need to coordinate their activities to achieve a common goal.

PRINCIPLES OF CONCURRENCY

RESOURCE ALLOCATION

- Operating systems must allocate resources such as memory, CPU time, and I/O devices to multiple processes or threads in a fair and efficient manner.

CONCURRENCY MECHANISMS

- These concurrency mechanisms are essential for managing concurrency in operating systems and are used to ensure safe and efficient access to system resources.

PROCESSES VS. THREADS

- An operating system can support concurrency using processes or threads.

SYNCHRONIZATION PRIMITIVES

- Operating systems provide synchronization primitives to coordinate access to shared resources between multiple processes or threads.

CONCURRENCY MECHANISMS

SCHEDULING ALGORITHMS

- Operating systems use scheduling algorithms to determine which process or thread should execute next.

MESSAGE PASSING

- Message passing is a mechanism used to communicate between processes or threads.

MEMORY MANAGEMENT

- Operating systems provide memory management mechanisms to allocate and manage memory resources.

INTERRUPT HANDLING

- Interrupts are signals sent by hardware devices to the operating system, indicating that they require attention.

ADVANTAGES OF CONCURRENCY

- Concurrency provides several advantages in operating systems, including

IMPROVED PERFORMANCE

- Concurrency allows multiple tasks to be executed simultaneously, improving the overall performance of the system.

RESOURCE UTILIZATION

- Concurrency allows better utilization of system resources, such as CPU, memory, and I/O devices.

ADVANTAGES OF CONCURRENCY

RESPONSIVENESS

- Concurrency can improve system responsiveness by allowing multiple tasks to be executed concurrently.

SCALABILITY

- Concurrency can improve the scalability of the system by allowing it to handle an increasing number of tasks and users without degrading performance.

FAULT TOLERANCE

- Concurrency can improve the fault tolerance of the system by allowing tasks to be executed independently. If one task fails, it does not affect the execution of other tasks.

PROBLEMS IN CONCURRENCY

- These problems can be difficult to debug and diagnose and often require careful design and implementation of concurrency mechanisms to avoid.
- **Sharing global resources**
  Sharing of global resources safely is difficult. If two processes both make use of a global variable and both perform read and write on that variable.
- **Optimal allocation of resources**
  It is difficult for the operating system to manage the allocation of resources optimally.
- **Locating programming errors**
  It is very difficult to locate a programming error because reports are usually not reproducible.
- **Locking the channel**
  It may be inefficient for the operating system to simply lock the channel and prevents its use by other processes.

THREADS

- Within a program, a **Thread** is a separate execution path.
- It is a lightweight process that the operating system can schedule and run concurrently with other threads.
- The operating system creates and manages threads, and they share the same memory and resources as the program that created them.
- This enables multiple threads to collaborate and work efficiently within a single program.

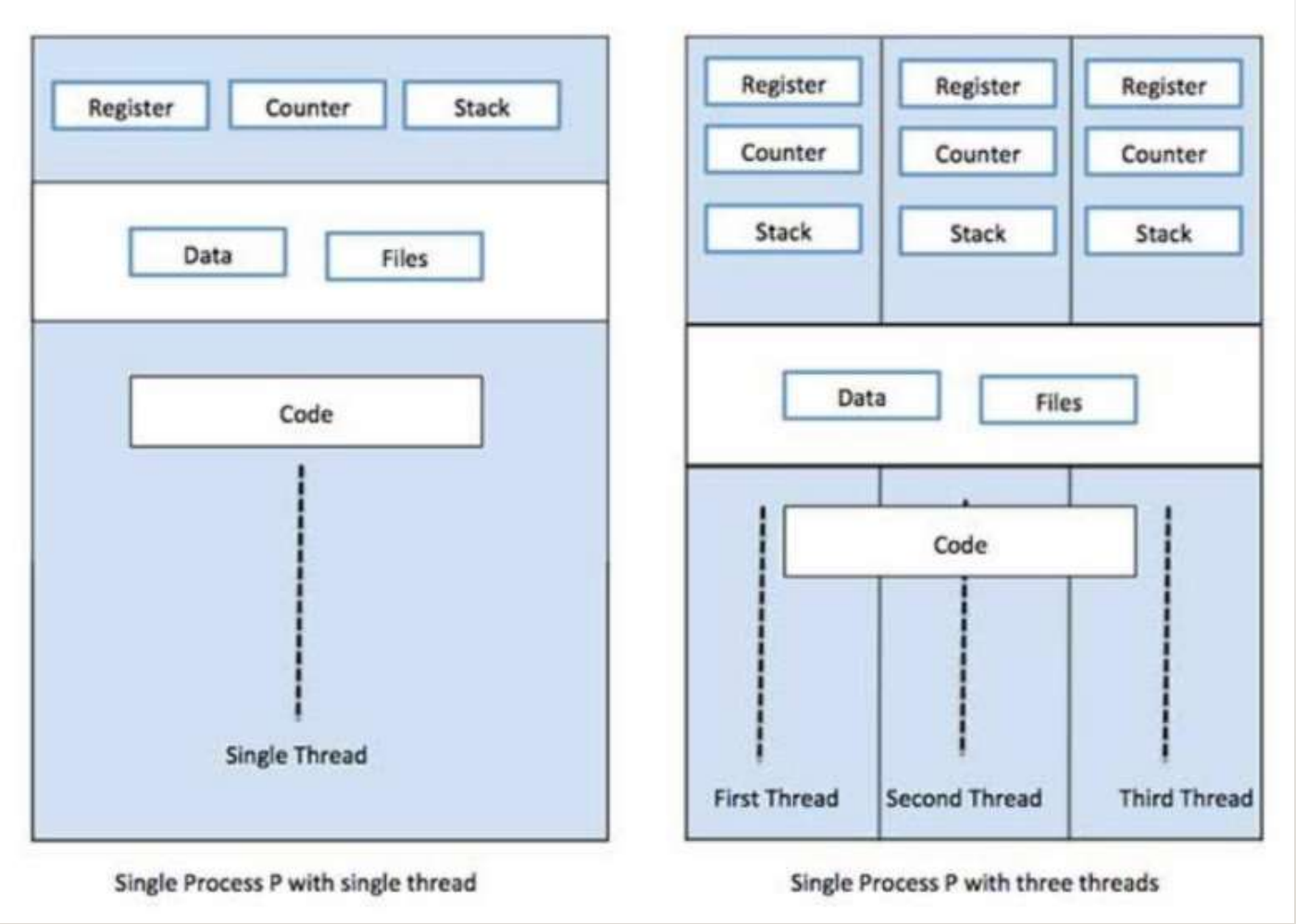
THREADS

- A new abstraction for a single running process
- Multi-threaded program
- A multi-threaded program has more than one point of execution.
- Multiple PCs (Program Counter), each of which is being fetched and executed from.
- thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.

THREADS

- A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.
- A thread shares with its peer threads few information like code segment, data segment and open files.
- A thread is a basic unit of CPU utilization. A thread, sometimes called as light weight process whereas a process is a heavyweight process.
- Thread comprises:
  - A thread ID
  - A program counter
  - A register set
  - A stack

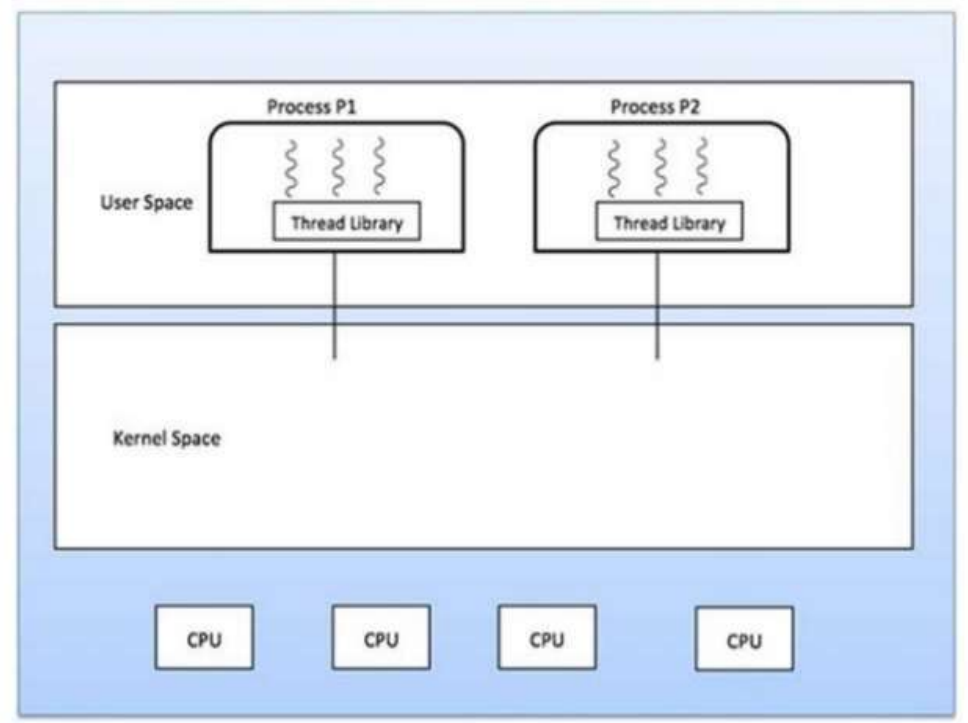
THREADS


Register
Counter
Stack
Data
Files
Code
Single Thread
Single Process P with single thread
Register
Counter
Stack
Register
Counter
Stack
Register
Counter
Stack
Data
Files
Code
First Thread
Second Thread
Third Thread
Single Process P with three threads

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

USER LEVEL THREADS:

- User threads are supported above the kernel and are managed without kernel support i.e., they are implemented by thread library at the user level.



KERNEL LEVEL THREADS:

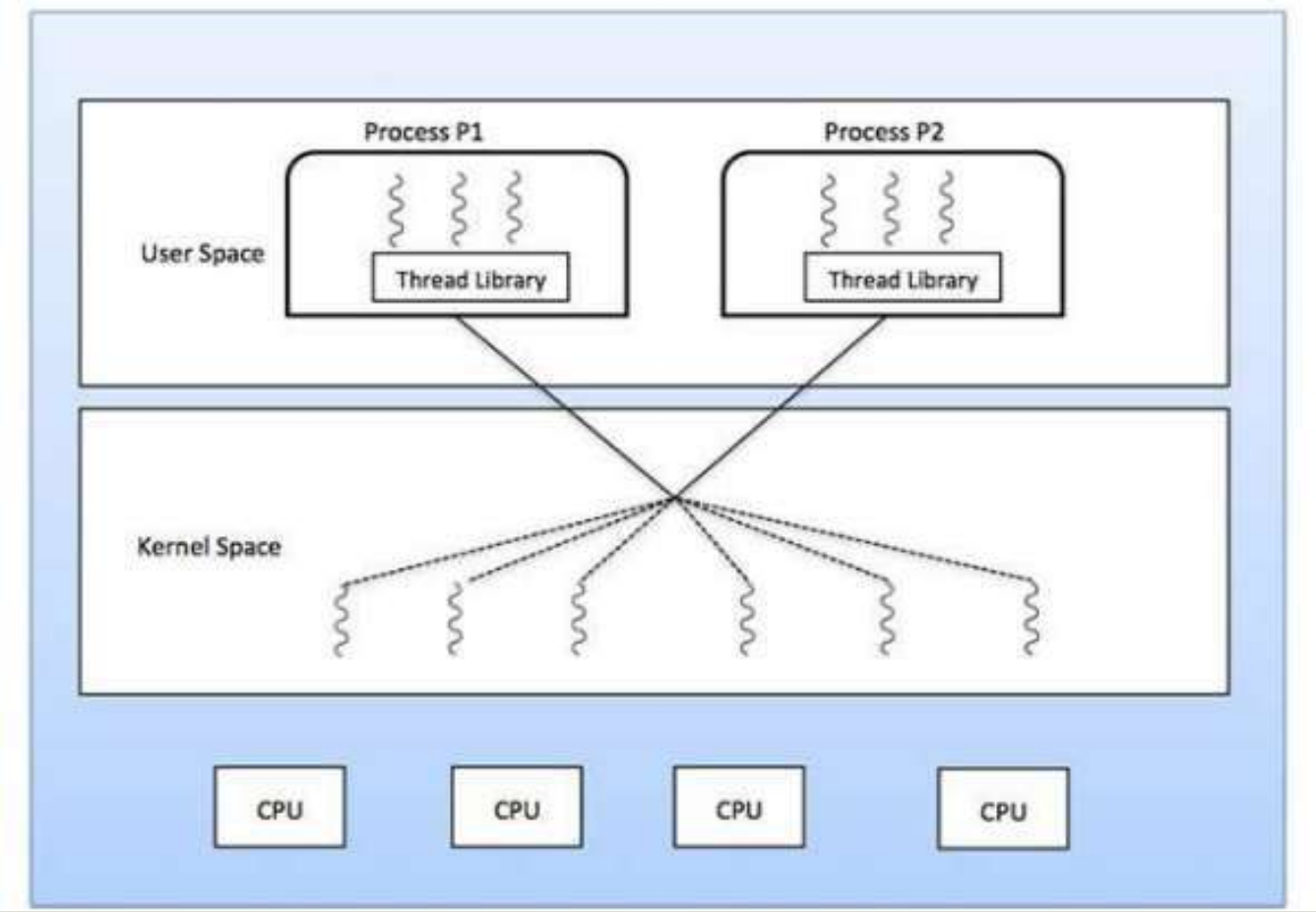
- Kernel threads are supported and managed directly by the operating system.
- Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process
- The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads

MULTITHREADING MODELS:

- Many systems provide support for both user and kernel threads, resulting in different multithreading models. Three common ways of establishing this relationship are:
- Many to many relationship.
- Many to one relationship.
- One to one relationship

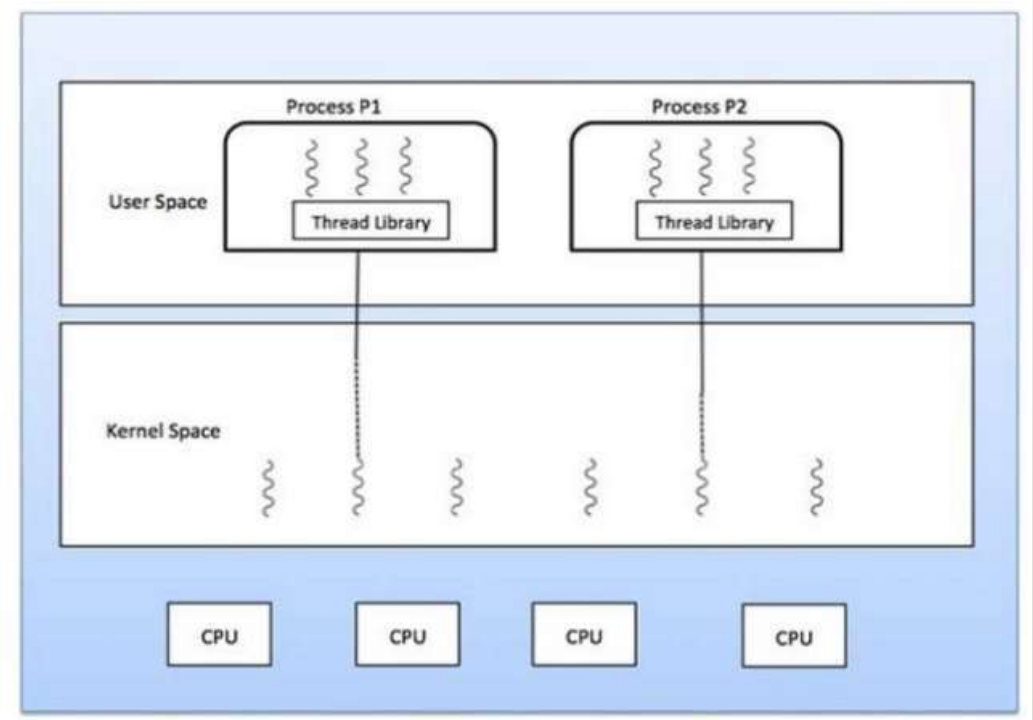
MANY TO MANY MODEL:

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads.
- In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine.



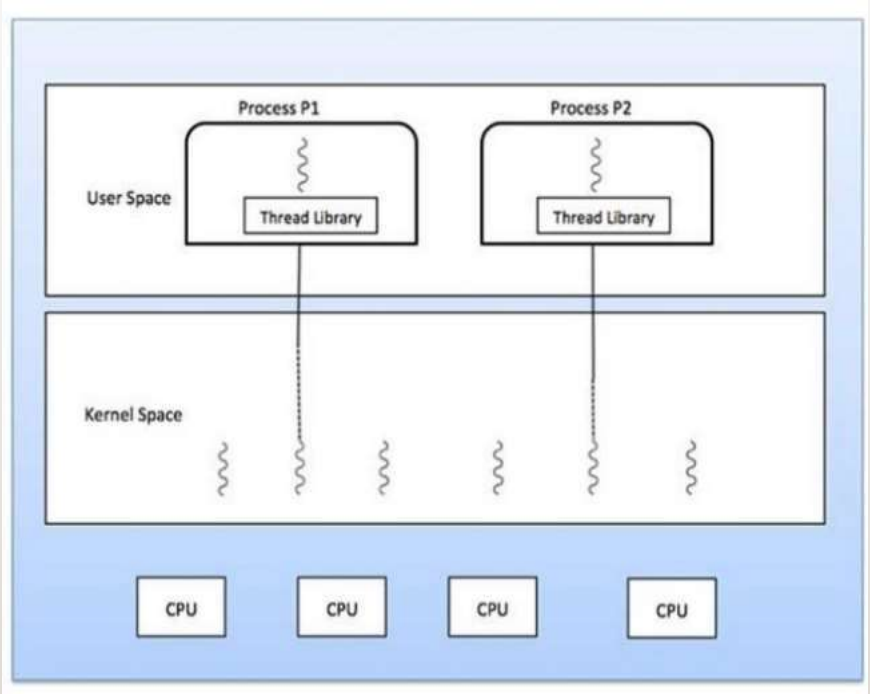
MANY TO ONE MODEL:

- The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call



ONE TO ONE MODEL:

- There is one-to-one relationship of user-level thread to the kernel-level thread.
- This model provides more concurrency than the many-to-one model.
- It also allows another thread to run when a thread makes a blocking system call.



S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

CONTEXT SWITCH BETWEEN THREADS

- **Each thread has its own program counter and set of registers.**
- if there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch must take place.**
- The context switch between threads is quite similar to the context switch between processes, as the register state of T1 must be saved and the register state of T2 restored before running T2.

CONTEXT SWITCH BETWEEN THREADS

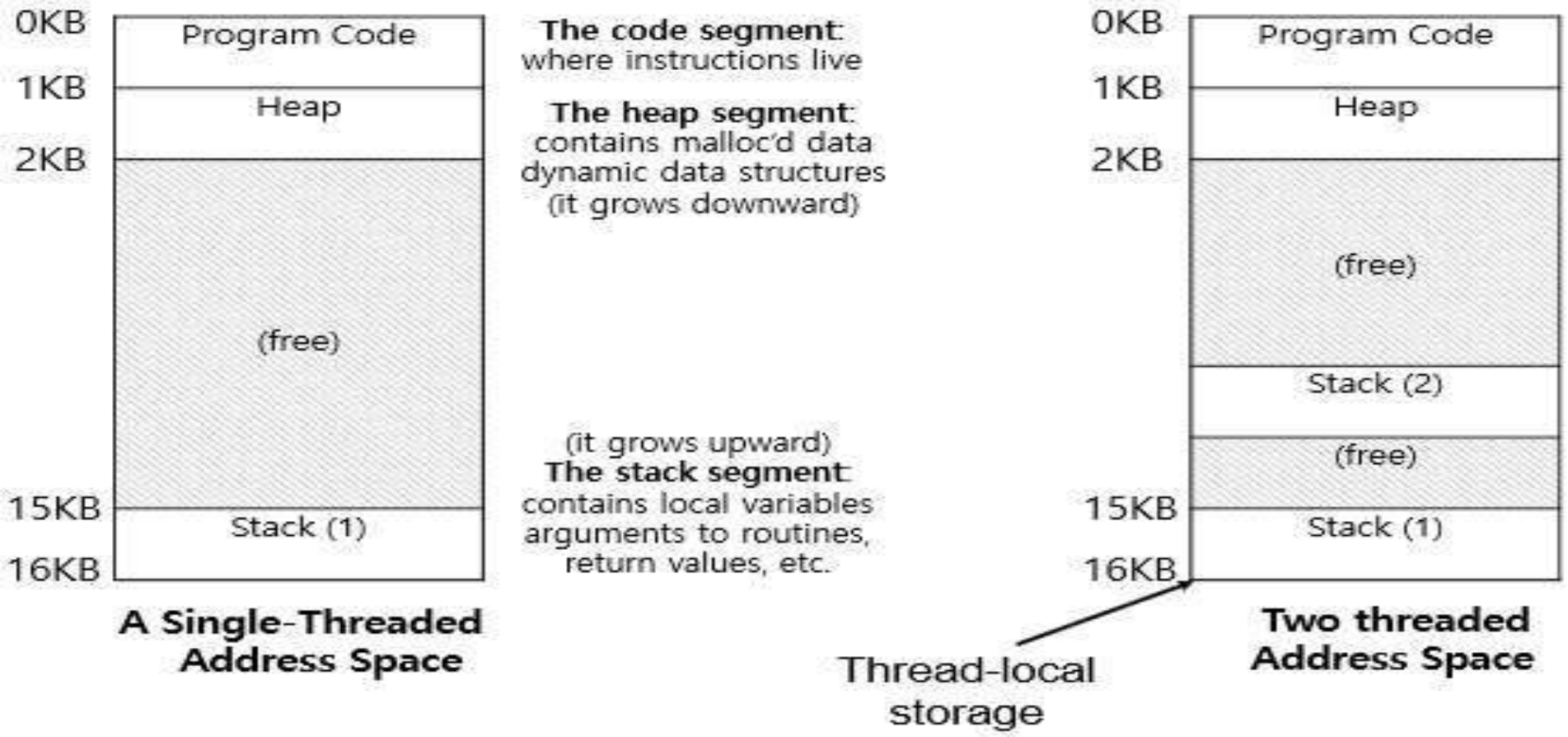
- With processes, we saved state to a **process control block (PCB); now, we'll need one or more thread control blocks (TCBs)** to store the state of each thread of a process.
- There is one major difference, though, in the context switch we perform between threads as compared to processes: the address space remains the same (i.e., there is no need to switch which page table we are using).

CONTEXT SWITCH BETWEEN THREADS

- In our simple model of the address space of a classic process (which we can now call a **single-threaded process), there is a single stack,** usually residing at the bottom of the address space.
- In a multi-threaded process, each thread runs independently.
- Thus, any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called **thread-local storage, i.e., the stack of the relevant** thread.

THE STACK OF THE RELEVANT THREAD

There will be one stack per thread.



WHY DO WE NEED THREAD?

- Threads run in parallel improving the application performance.
- Each such thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads can share common data so they do not need to use inter-process communication. Like the processes, threads also have states like ready, executing, blocked, etc.
- Priority can be assigned to the threads just like the process, and the highest priority thread is scheduled first.
- Each thread has its own Thread Control Block (TCB). Like the process, a context switch occurs for the thread, and register contents are saved in (TCB).

THREAD API

- Thread API (Application Programming Interface) provides a set of functions for creating, managing, and synchronizing threads within a program. Threads are the smallest unit of execution within a process and allow for concurrent execution of code.
- **Thread Creation and Management**
- **Creating Threads:** Most Thread APIs provide functions to create new threads. For example, in POSIX threads (pthreads) on Unix-like systems, pthread_create is used.
- **Exiting Threads:** Functions like pthread_exit allow a thread to terminate itself.
- **Joining Threads:** Functions like pthread_join allow one thread to wait for the termination of another.

THREAD SYNCHRONIZATION

- **Mutexes:** Mutexes (mutual exclusions) are used to prevent multiple threads from accessing shared resources simultaneously. Functions include pthread_mutex_init, pthread_mutex_lock, and pthread_mutex_unlock.
- **Condition Variables:** These allow threads to wait for certain conditions to be met. Functions include pthread_cond_wait and pthread_cond_signal.
- **Semaphores:** Semaphores are signaling mechanisms used to control access to shared resources. Functions include sem_init, sem_wait, and sem_post.

KEY POINTS

- **Concurrency:** Threads enable concurrent execution within a single process.
- **Synchronization:** Proper synchronization mechanisms (mutexes, condition variables, semaphores) are crucial to avoid race conditions and ensure thread-safe operations.
- **Portability:** While the concepts are similar, the API functions and their usage may differ across operating systems.

WHAT IS SEMAPHORES ?

- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.
- **Wait**The wait operation decrements the value of its argument **S**, if it is positive. If **S** is negative or zero, then no operation is performed.
  - **wait(S)**

```
{
while (S<=0);
S--;
}
```

WHAT IS SEMAPHORES ?

- **Signal -** The signal operation increments the value of its argument **S**.
  - **signal(S)**

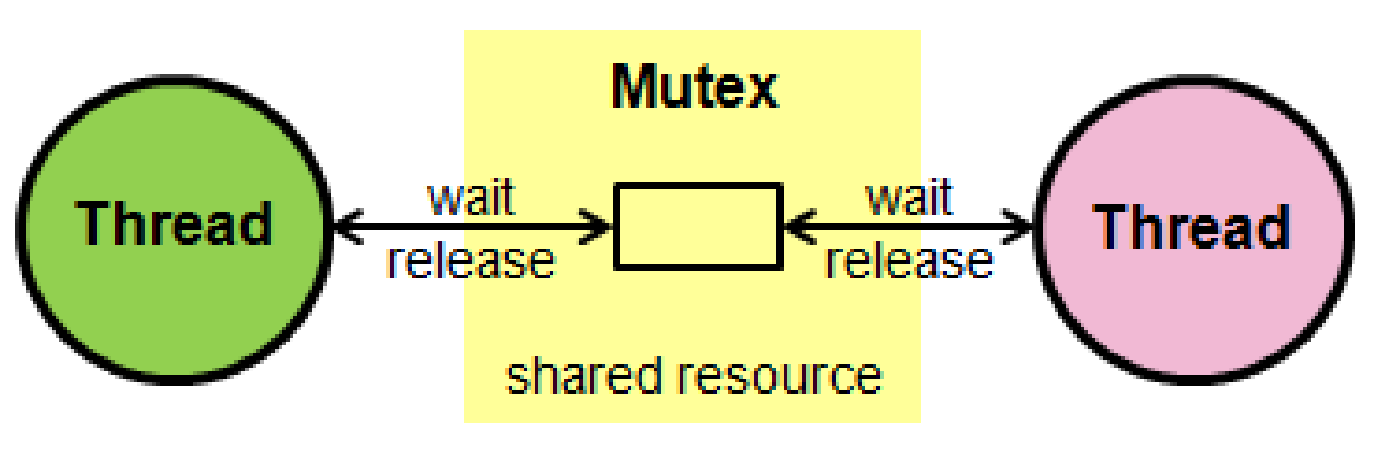
```
{
S++;
}
```

TYPES OF SEMAPHORES

- **Counting Semaphores**
- These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources.
- **Binary Semaphores**
- The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.

MUTEX

- A mutex is a binary variable used to provide a locking mechanism. It offers mutual exclusion to a section of code that restricts only one thread to work on a code section at a given time.



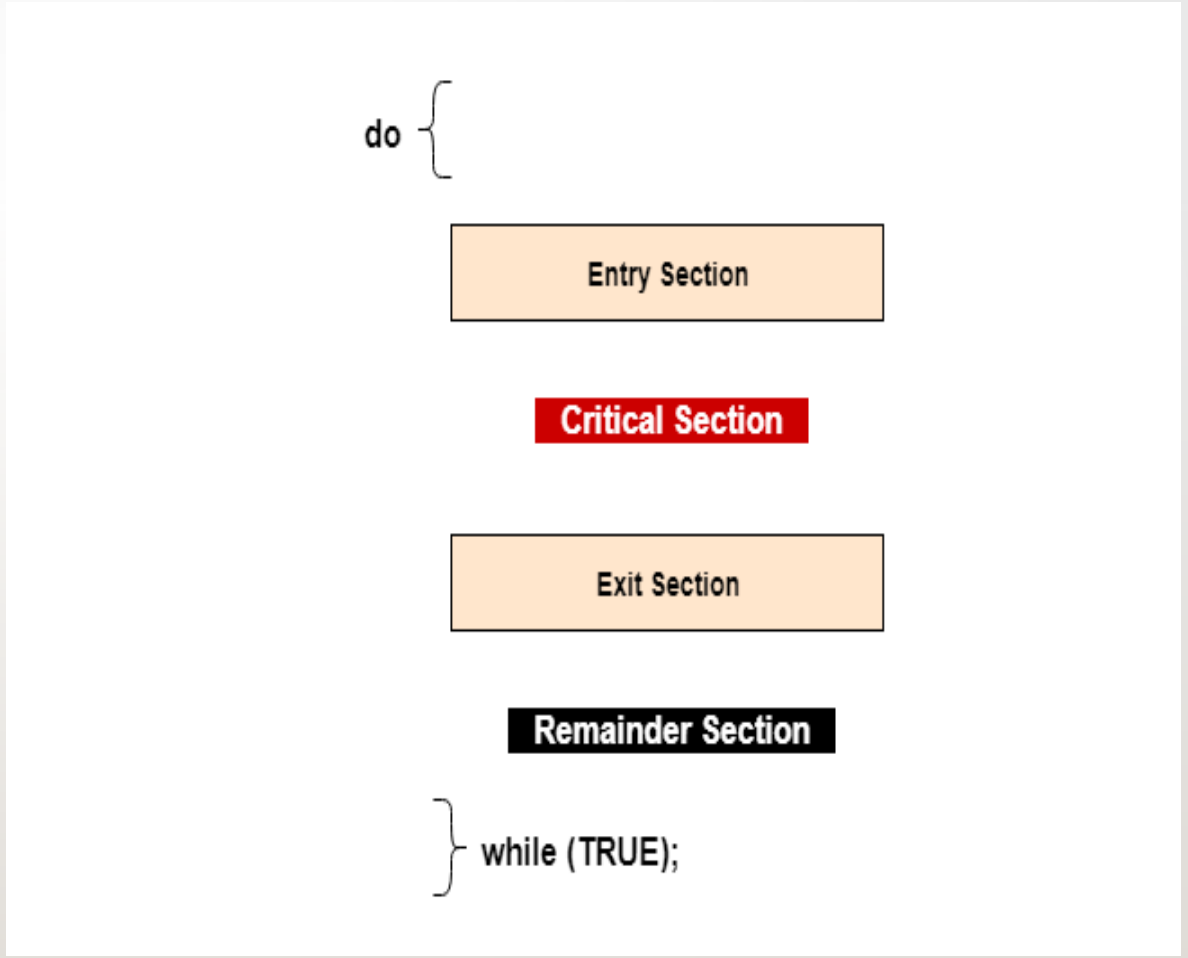
WORKING OF MUTEX

- Suppose a thread executes a code and has locked that region using a mutex.
- If the scheduler decides to perform context switching, all the threads ready to execute in the same region are unblocked.
- Out of all the threads available, only one will make it to the execution, but if it tries to access the piece of code already locked by the mutex, then this thread will go to sleep.
- Context switching will take place again and again, but no thread will be able to access the region that has been locked until the lock is released.
- Only the thread that has locked the region can unlock it as well.

Mutex	Semaphore
A mutex is an object.	A semaphore is an integer.
Mutex works upon the locking mechanism.	Semaphore uses signaling mechanism
Operations on mutex: • Lock • Unlock	Operation on semaphore: • Wait • Signal
Mutex doesn't have any subtypes.	Semaphore is of two types: • Counting Semaphore • Binary Semaphore
A mutex can only be modified by the process that is requesting or releasing a resource.	Semaphore work with two atomic operations (Wait, signal) which can modify it.
If the mutex is locked then the process needs to wait in the process queue, and mutex can only be accessed once the lock is released.	If the process needs a resource, and no resource is free. So, the process needs to perform a wait operation until the semaphore value is greater than zero.

CRITICAL SECTION PROBLEM

- The critical section is a code segment where the shared variables can be accessed.
- An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time.
- All the other processes have to wait to execute in their critical sections.



WHAT IS CRITICAL SECTION PROBLEM?

- A critical section is a segment of code which can be accessed by a signal process at a specific point of time. The section consists of shared data resources that required to be accessed by other processes.
- The entry to the critical section is handled by the wait() function, and it is represented as P().
- The exit from a critical section is controlled by the signal() function, represented as V().
- In the critical section, only a single process can be executed. Other processes, waiting to execute their critical section, need to wait until the current process completes its execution.

SOLUTION TO THE CRITICAL SECTION PROBLEM

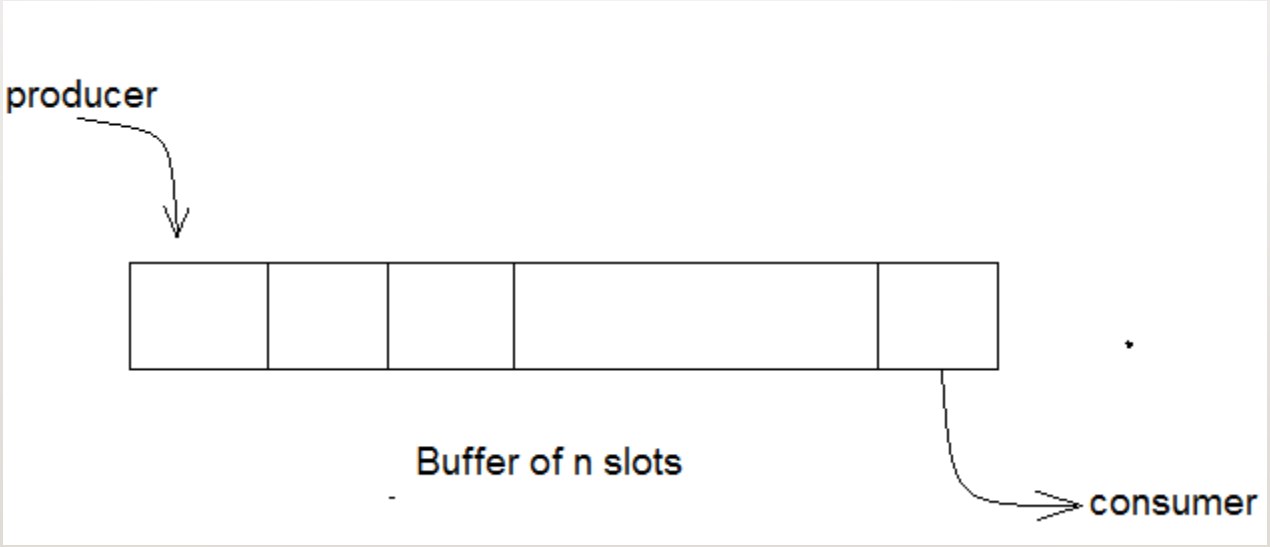
- The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions –
- **Mutual Exclusion**
- Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.
- **Progress**
- Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it.

SOLUTION TO THE CRITICAL SECTION PROBLEM

- **Bounded Waiting**
- Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

BOUNDED BUFFER PROBLEM / PRODUCER CONSUMER PROBLEM

- **What is the Problem Statement?**
- There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer.



PRODUCER CONSUMER PROBLEM

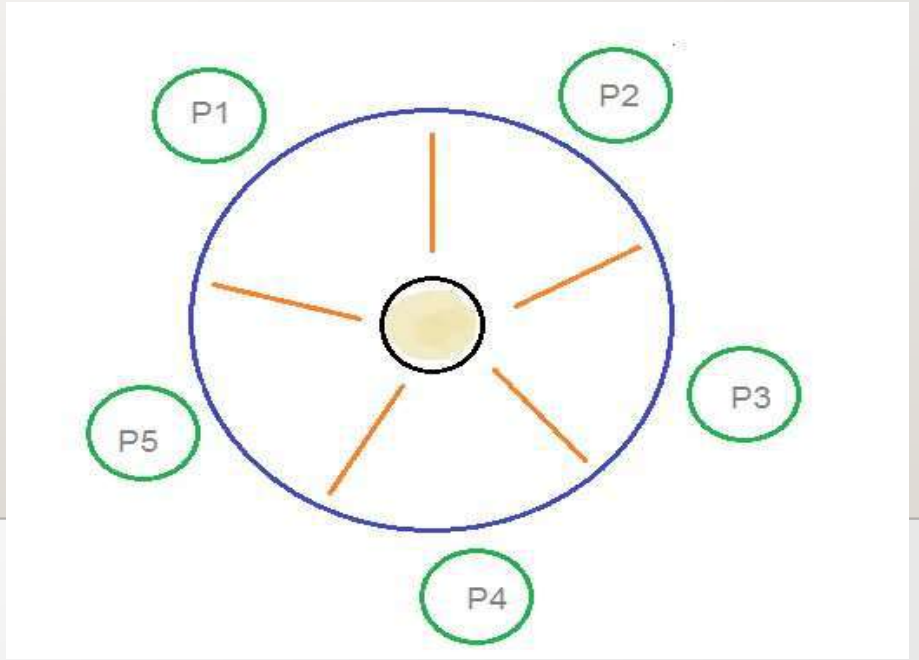
- A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.
- There needs to be a way to make the producer and consumer work in an independent manner.

HERE'S A SOLUTION

- One solution of this problem is to use semaphores. The semaphores which will be used here are:
- m, a binary semaphore which is used to acquire and release the lock.
- empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- full, a counting semaphore whose initial value is 0.
- At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

DINING PHILOSOPHERS PROBLEM

- The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.
- What is the Problem Statement?
- Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



HERE'S THE SOLUTION

- From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.
- An array of five semaphores, stick[5], for each of the five chopsticks.
- When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.
- But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.

POSSIBLE SOLUTION

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

READER WRITER PROBLEM

- Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.
- **The Problem Statement**
- There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context.
- They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

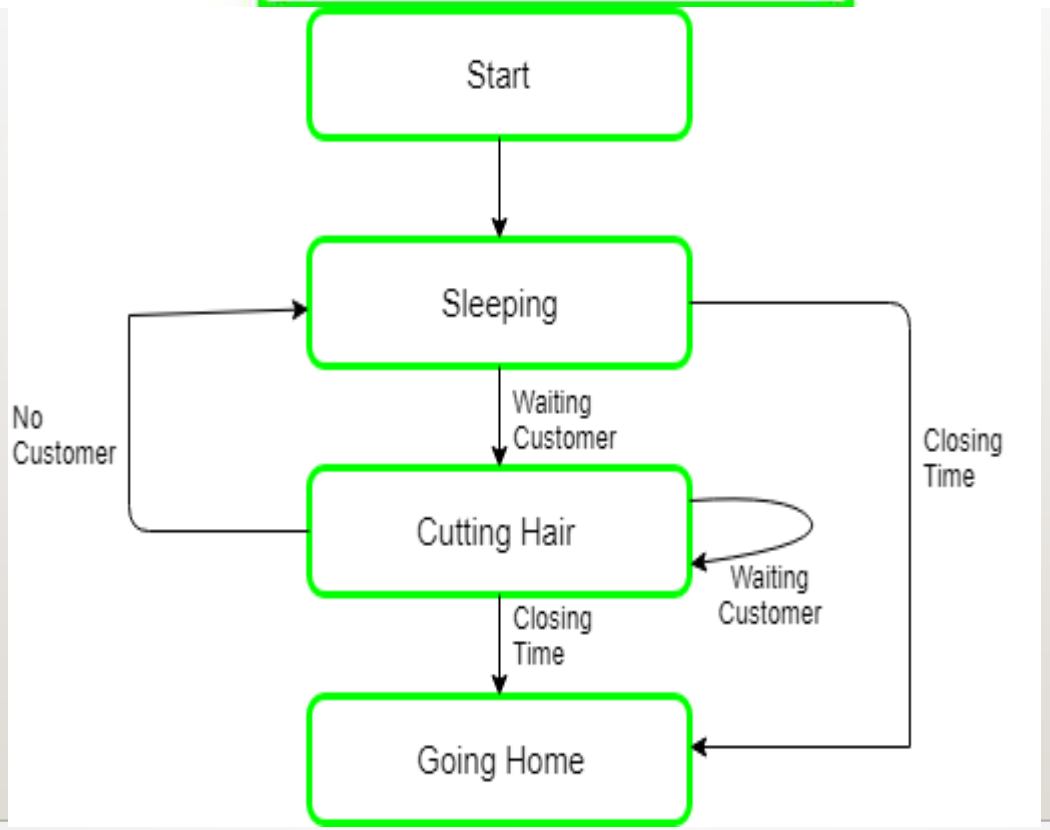
THE SOLUTION

- From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.
- Here, we use one mutex m and a semaphore w. An integer variable read_count is used to maintain the number of readers currently accessing the resource.

SLEEPING BARBER PROBLEM IN PROCESS SYNCHRONIZATION

- There is a barber shop with one barber and a number of chairs for waiting customers.
- Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available.
- If there are no chairs available, the customer leaves.
- When the barber finishes with a customer, he checks if there are any waiting customers.
- If there are, he begins cutting the hair of the next customer in the queue. If there are no customers waiting, he goes to sleep.


Waiting Chairs
Customer is leaving
Sleeping Barber
Customer is entering


Start
Sleeping
Waiting Customer
No Customer
Closing Time
Cutting Hair
Waiting Customer
Closing Time
Going Home

THANK YOU

Team – System Design & Introduction to Cloud