Assignment 2

1. What is process control block? Explain operations on process.
Ans:When creating a process, the operating system undertakes multiple activities. To identify processes, each one is assigned a process identification number (PID).

To support multi-programming, the operating system must track all processes.
The process control block (PCB) is used to monitor the process's status.
Each memory block stores information such as the process state, program counter, stack pointer, file status, and scheduling algorithms.
This information must be saved while the process transitions between states.
When a process transitions from one state to another, the operating system updates information in the process's PCB.

<u>Operations on process</u>
-Creation:This is the first step in the process execution activity. Process creation refers to the development of a new process for execution. This could be done by the system, the user, or the old process itself. Several events occur before the process is created. Some of these events include the following:

When we start the computer, the system initiates various background activities.
A user can request to start a new process.
A process can generate a new process while it is being executed.
The batch system initiates the batch job.

-Scheduling/Dispatching:The event or activity that causes the process's status to transition from ready to run to running. This signifies that the operating system moves the process from the ready state to the running state. The operating system dispatches when resources are available or the process has a higher priority than the current process. In several additional instances, the operating system preempts the running process and dispatches the ready process.

-Blocking:When a process calls an input-output system call that blocks the process, the operating system enters block mode. Block mode is essentially a mode in which the process waits for input and output. As a result, when the process requires it, the operating system blocks the process and sends another to the processor. As a result, during process-blocking actions, the operating system sets the process to 'waiting'.

-Preemption:When a timeout happens, it signifies that the process was not terminated within the specified time interval, and the following process is ready to execute, therefore the operating system preempts it. This operation is only applicable when CPU scheduling allows preemption. This occurs mostly in priority scheduling, when the incoming of a high priority process preempts the continuing process. As a result, while using process preemption, the operating system sets the process to'ready'.

-Process Termination:Process termination is the act of ending a process. In other terms, process termination refers to the release of computer resources used by the process for operation. Termination, like creation, can be the result of a series of events. Some of these include:
*The process completes its execution and notifies the operating system that it is

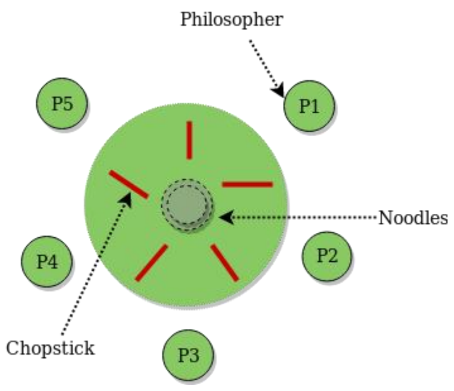
finished.
*The operating system stops the process owing to service problems.
*A hardware fault may have caused the process to terminate.
*A process can terminate another process.

2. Define the term semaphore. How does semaphore help in dining philosopher problem?
Ans: A semaphore is a synchronization component used in operating systems and concurrent programming to control access to shared resources among several processes or threads. It is essentially a variable that regulates access to a shared resource by numerous processes in a concurrent system, such as a multitasking operating system.

Semaphores come in two types:

a)Binary semaphores, also known as mutexes, have only two values (0 and 1). It ensures mutual exclusion.
b)Counting Semaphore: It accepts non-negative integer values. It is used to restrict access to a resource with a limited number of instances.

Dining Philosopher Problem:The Dining Philosopher Problem asserts that K philosophers are sitting around a circular table, one chopstick between each pair of philosophers. There is only one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks next to him. One chopstick may be picked up by any of its nearby followers, but not both.



Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
  {  THINK;
    PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
    EAT;
    PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
  }
```

The philosopher can be in three states: thinking, hungry, and eating. There are two semaphores here: Mutex and a philosophers' semaphore array. Mutex is used to ensure that no two philosophers access or put down the pickup at the same time. The array is used to direct the behavior of each philosopher. However, semaphores might cause deadlock due to programming flaws.

The steps for the Dining Philosopher Problem solution using semaphores are as follows

1. Set the semaphores for each fork to one (showing that they are available).

2. Set a binary semaphore (mutex) to 1 so that only one philosopher can try to pick up a fork at a time.

3. For each philosopher process, start a separate thread that runs the following code:

While true, consider a random amount of time.
Acquire the mutex semaphore so that only one philosopher can pick up a fork at a time.
Attempt to obtain the semaphore for the fork on the left.
If successful, try to obtain the semaphore for the fork on the right.
If both forks are properly acquired, consume for a random period of time before releasing both semaphores.

If you are unable to acquire both forks, release the semaphore for the fork to the left (if acquired), followed by the mutex semaphore, and return to thinking.

3. What is process? How does it differ form program? Explain.
Ans:A process is a program that is currently running on a computer.
It has a similar meaning to the term task, which is used in several operating systems.
In UNIX and comparable operating systems, a process begins when a program is started, such as by a user inputting a shell command or another application.
A program is not a process, but rather a passive entity like a file with instructions saved on disk. When an executable file is loaded and run, it transforms into a process.

A program is a set of ordered operations to be performed. A process refers to the execution of a program.
A program is passive and does nothing until executed, while a process is dynamic and performs particular actions.
A program has a longer lifespan as it is stored in memory until deliberately removed, whereas a process has a shorter lifespan as it is terminated upon task completion.
Processes demand more resources, such as processing, memory, and I/O, to run successfully. In contrast, a program only requires RAM for storage.

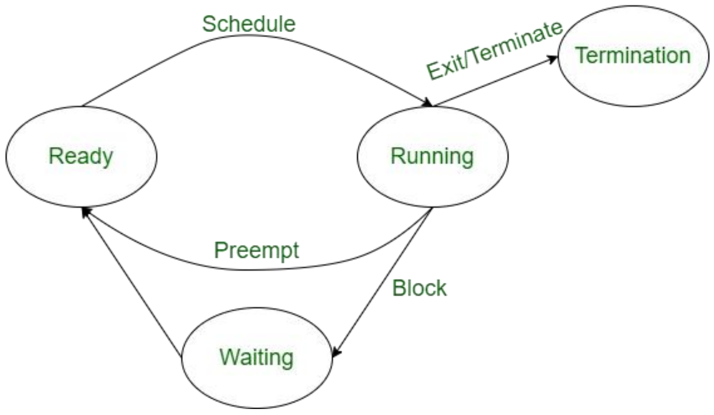
Aspect	Program	Process
Nature	Static entity (set of instructions)	Dynamic entity (instance of a program in execution)
Existence	Exists at rest (stored	Exists in execution (loaded in

	on disk)	memory)
State	No state (only code)	Has a state (running, waiting, ready, terminated)
Resources	Doesn't consume resources except disk space	Consumes resources (CPU, memory, I/O devices)
Multiplicity	Single copy on disk	Multiple instances can run simultaneously
Control Block	No PCB	Has a PCB (Process Control Block)
Interactivity	Passive (doesn't execute on its own)	Active (executes the program's instructions)
Example	Text editor program file	Running instance of the text editor

4. What are different state process models? Draw a state (block) diagram of process with different state and explain each briefly.

Ans:Processes in an operating system typically go through various states during their lifecycle. The most common process states are:

1. **New:** The process is being created.
2. **Ready:** The process is ready to run and is waiting for CPU time.
3. **Running:** The process is currently being executed by the CPU.
4. **Waiting (Blocked):** The process is waiting for some event (like I/O completion) to occur.
5. **Terminated:** The process has finished execution.



· **<u>Transition Between States</u>**

New to Ready:

- · Once the process is created and initialized, it moves to the ready state.

· **Ready to Running:**

- · The process scheduler selects a process from the ready queue and assigns it the CPU, moving it to the running state.

· **Running to Waiting:**

- · If the process needs to wait for an event (e.g., I/O operation), it moves to the waiting state.

· **Running to Ready:**

- · If the process's time slice expires or it is preempted by a higher-priority process, it moves back to the ready state.

· **Waiting to Ready:**

- · When the event the process was waiting for occurs, it moves from the waiting state back to the ready state.

· **Running to Terminated:**

- · When the process finishes its execution, it moves to the terminated state.

5 Describe how multithreading improve performance over a single-threaded solution.

Ans:Multithreading enables a program to do several operations concurrently by breaking tasks into smaller threads that can run in parallel. This method can greatly enhance performance over a single-threaded alternative. Here are some ways multithreading improves performance:
Increased responsiveness:
Multithreading can keep graphical user interfaces (GUIs) responsive. For example, one thread may handle user inputs while another does background computations or I/O tasks.
Multithreading is useful for games, real-time simulations, and other interactive applications because it separates graphics, physics calculations, and input handling into various threads.

Improved Resource Utilization:
CPU Utilization: Multithreading improves CPU utilization by keeping multiple cores engaged. Threads on a multi-core CPU can be distributed across numerous cores, allowing for real parallel execution.
I/O Operations: Multithreading can increase the performance of applications that rely heavily on I/O operations. While one thread waits for I/O operations to complete, other threads can continue to run, reducing idle time.

Improved throughput:
Task Parallelism: Multithreading can boost throughput by running numerous separate processes concurrently. A web server, for example, can process numerous client requests at the same time by assigning each one to a separate thread.
Pipeline Processing: In some applications, work can be separated into stages, each handled by a separate thread. This pipeline strategy can increase overall processing performance by overlapping the execution of many steps.

Scalability:
Scalable Applications: Multithreading-enabled applications scale more effectively with more CPU cores. As the number of cores increases, the application can establish new threads to make use of the extra processing capability.
Concurrent Data Processing: Multithreading enables concurrent processing of data chunks in data-intensive applications, making the application more scalable and faster when processing huge datasets.
Reduced latency:

Real-Time Systems: Multithreading can help meet timing limitations by allowing vital processes to execute alongside less critical tasks. This lowers the latency for high-priority tasks.
Batch Processing: Multithreading can reduce the time required to complete large batches of work by dividing them across numerous threads.

6 Differentiate between kernel level thread and user level thread. Which one has a better performance?

Ans:

Aspect	Kernel-Level Threads (KLT)	User-Level Threads (ULT)
Management	Managed by the operating system kernel	Managed by user-level libraries
Creation and Termination	Requires system calls, which are slower	Quicker as it does not involve system calls
Switching	Thread switching requires context switching by the kernel, which is slower	Thread switching is done by the thread library, which is faster
Blocking	If a kernel-level thread blocks (e.g., on I/O), the entire process does not block	If a user-level thread blocks, the entire process may block
Portability	Less portable, depends on the operating system	More portable, as it is managed by the user-level library
Concurrency	True concurrency can be achieved on multiprocessor systems	Concurrency is limited to the user-level library's scheduling
Complexity	Simpler for the application developer, as the kernel	More complex for the developer, as they must manage scheduling

	handles scheduling and management	and synchronization
Resource Usage	Generally uses more resources (kernel resources)	More efficient in terms of resource usage (memory, etc.)
Examples	Windows threads, Linux threads (pthreads)	Java threads (managed by the JVM), user-level thread libraries

<u>Which has better Performance</u>

Due to the lack of kernel participation, user-level threads (ULT) typically perform better in terms of creation, termination, and context switching. They are suitable for applications in which threads frequently perform short tasks and non-blocking activities predominate.

Kernel-Level Threads (KLT): Improve performance for applications that require true parallelism on multiprocessor systems and improved handling of blocking operations. They are better suited for I/O-intensive, high-performance computing applications that require blocking operations and real concurrency.

7. Differentiate between process and thread.

Aspect	Process	Thread
Definition	An independent program in execution with its own memory space	A smaller unit of a process that shares the same memory space
Memory	Has its own memory space, including code, data, and stack	Shares memory space with other threads of the same process (code, data)
Creation	More resource-intensive, involves allocating separate memory space	Less resource-intensive, shares existing memory of the process
Communication	Requires inter-process communication mechanisms (pipes, message queues, shared memory)	Communicates directly through shared memory
Overhead	Higher overhead due to separate memory and resource allocation	Lower overhead due to shared resources
Context Switching	Slower due to more complex state saving	Faster due to less state saving and restoring

	and restoring	
Independence	Processes are independent of each other	Threads are dependent on each other and share the same resources

8. What resources are used when a thread is created? How do they differ from those used when a process is created?

Ans: When a thread is created, the resources listed below are frequently used:

Thread Control Block(TCB):
The TCB includes information about the thread, including its status, program counter, stack pointer, register values, thread ID, and scheduling information.

Stack:
Each thread has its own stack that contains local variables, return addresses, and control information for function calls. To store thread-specific data, each thread has its own stack.

Registers:
Registers store the thread's current working variables. When a context switch happens, the thread's register state is preserved in the TCB.
Thread-Specific Data:

Any thread-specific data that needs to be kept separate from other threads.

Aspect	**Thread**	**Process**
Control Block	Thread Control Block (TCB)	Process Control Block (PCB)
Memory	Shares memory space with other threads in the same process (shared code, data, heap)	Separate memory space (own code, data, heap, stack segments)
Stack	Separate stack for each thread	Separate stack, along with separate code, data, and heap segments
Registers	Own set of registers saved in the TCB	Own set of registers saved in the PCB
Address Space	Shares address space with other threads of the same process	Separate address space
Creation Overhead	Lower overhead as it shares resources with the process	Higher overhead due to allocation of separate memory space and resources
Communication	Easier and faster through shared memory	More complex, often requires inter-process communication mechanisms
Context Switching	Faster due to less state information to save and resto	Slower due to more state information to save and restore

9. Compare the use of monitor and semaphore operations.

Ans:

Aspect	Monitor	Semaphore
Definition	A high-level synchronization construct that combines mutual exclusion and condition variables	A low-level synchronization primitive that uses counters to manage access to resources
Type	High-level construct	Low-level construct
Structure	Encapsulates shared resources and the synchronization code (methods) in a single module	Uses a counter and two atomic operations (wait and signal)
Operations	- wait(): Puts the calling thread to sleep until it is notified - notify(): Wakes up one waiting thread - notifyAll(): Wakes up all waiting threads	- wait(P): Decrements the counter and possibly blocks the calling process if the counter is negative - signal(V): Increments the counter and possibly wakes up a blocked process
Usage Complexity	Simpler to use and understand, as it encapsulates synchronization within methods	More complex and error-prone due to manual handling of counters
Mutual Exclusion	Automatically ensures mutual exclusion when accessing shared resources within its methods	Needs to be explicitly managed by the programmer using the semaphore operations
Condition Variables	Provides condition variables to handle complex synchronization scenarios	Does not have built-in condition variables; requires additional logic for similar functionality
Blocking	Automatically handles thread blocking and waking based on the condition variables	Requires explicit blocking and waking up of processes/threads based on the semaphore value

10. Define mutual exclusion and critical section.

Ans: • Sequences of instructions that may get incorrect results if executed simultaneously are called critical sections

• (We also use the term race conditionto refer to a situation in which the results depend on timing)

• Mutual exclusionmeans "not simultaneous"–A < B or B < A–We don't care which

• Forcing mutual exclusion between two critical section executions is sufficient to ensure correct execution guarantees ordering

• One way to guarantee mutually exclusive execution is using locks

11. What are CPU scheduling criteria? For the processes listed below, draw Gantt chart illustrating their execution, calculate average waiting and average turnaround time using:

a) FCFS

b) SRTN

c) Priority

d) Round Robin (quantum=1 sec)

Procees	Arrival Time	Burst Time	Priority
A	0.00	7	3
B	2.01	7	1
C	3.01	2	4
D	3.02	2	2

Ans: In multiprogramming systems, numerous processes often fight for CPU resources at the same time.
-When multiple processes are ready at the same time, a decision must be made regarding which process to run next.

-The operating system's Scheduler uses the Scheduling algorithm.

-Process execution is divided into CPU execution and I/O wait cycles.

-Processes switch between these two states.

-Process execution starts with a CPU burst, followed by an I/O burst, and so on.

Eventually, the CPU burst stops with a system request to stop execution.

Scheduling Criteria

CPU utilization
We need to keep the CPU as active as possible. It may range from 0 to 100%. Real-world systems should have a range of 40-90% for both low and heavy loads.

Throughput measures the number of processes done per unit time. For extended processes, the rate may be 1 per hour, whereas brief transactions may have a throughput of 10 per second.

TurnaroundTime
The total time spent waiting in memory, waiting in the ready queue, executing on the CPU, and performing I/O.
Turnaround time refers to the time it takes from process submission to completion.

Waiting time plus service time.
Turnaround time equals the difference between the time the project was completed and when it was submitted.
(waiting time plus service time/burst time)
Waiting time is the total of waiting durations in the ready queue.

Response time
In an interactive system, turnaround time is not the most important criterion. Response time refers to the time it takes to initiate a response, rather than the time it takes to output that answer.