Q. Describe the desirable features of a global scheduling algorithm.

1. No A Priori Knowledge About Processes:

- Often, we don't know the details of processes in advance (e.g., how much resource they need).
- A good scheduling algorithm should work without needing this prior information.

2. Dynamic in Nature:

- The workload on different nodes in a distributed system can change over time.
- The scheduling algorithm should be flexible enough to adapt to these changes.

3. Quick Decision-Making:

- The state of the system keeps changing, so the algorithm needs to make decisions quickly.
- It should be able to analyze the situation and allocate resources fast.

4. Balanced System Performance and Scheduling Overhead:

- Many global scheduling algorithms gather information about the entire system's state to make decisions.
- The algorithm should achieve good performance with minimal overhead from collecting this information.

5. Stability:

- If processes are moved too often (migration), it can lead to inefficiency, known as "thrashing."
- The scheduling algorithm should minimize unnecessary process migrations to maintain system stability.

6. Scalability:

- As the system grows (more nodes or processes), the scheduling algorithm should still perform well.
- It should be able to scale to handle systems of any size.

7. Fault Tolerance:

- If a node fails or crashes, it should not affect the overall system performance significantly.
- The algorithm should be able to handle such failures without major disruption.

8. Fairness of Service:

- When two users or processes request the same resources, they should be treated fairly.
- The algorithm should ensure that resources are assigned fairly to avoid any user being neglected or over-prioritized.

Q. Explain resource management in distributed system.

Resource Management in a distributed system means using all available resources (like computers, memory, storage, etc.) in the best possible way to get good performance and avoid delays or overload.

Key Points:

1. Resource management is about **using resources effectively and efficiently** when they are needed.
2. Resources can include **money, hardware, software, human skills, or data**.
3. A **resource manager** handles the scheduling of tasks and jobs across the distributed system.
4. The goal is to **optimize resource use**, reduce **response time**, and prevent **network congestion**.

Techniques for Resource Management:

I) Task Assignment Approach:

- Every process (or job) is seen as a set of smaller **tasks**.
- These tasks are assigned to the best available **nodes (computers)** to improve performance.

Goals of Task Assignment:

- Reduce communication cost between tasks (IPC cost).
- Complete the whole process quickly.
- Achieve good **parallelism** (many tasks running at the same time).
- Use system resources **efficiently**.

Example: If you have a large file to convert, break it into smaller parts and send each part to a different computer to finish faster.

II) Load Balancing Approach:

- Distribute the tasks evenly across all computers in the system.

- Aim is to **balance the load** so that no computer is too busy while others are free.

Example: If one computer is overloaded, move some tasks to others that are free.

III) Load Sharing Approach:

- Make sure that **no computer stays idle** while others are working.
- Share the tasks to make sure all available computers are doing some work.

Example: If a task is waiting and a computer is doing nothing, move the task there.

Q. Write a short note on load balancing technique.

Load Balancing is a technique used to manage processes in distributed systems. It involves distributing the workload evenly across all the nodes in the system to ensure that no single node is overloaded. The goal is to optimize the use of system resources, maximize throughput, minimize response time, and prevent any node from being overwhelmed.

Key Features of Load Balancing:

1. **Main Goal**:
   - Balance the workload across all nodes in the system.
   - Ensure efficient use of resources.
   - Maximize performance while minimizing delays.
2. **How It Works**:
   - Processes are distributed among the nodes to ensure that each node has a similar amount of work.
   - The algorithm transfers tasks from heavily loaded nodes to lightly loaded ones, improving overall system performance.

Types of Load Balancing Algorithms:

1. **Static Load Balancing**:
   - These algorithms do not consider the current state of the system.
   - For example, if a node is overloaded, a task is randomly transferred to another node.
   - Static algorithms are simpler but may not perform as well.
   - **Types**:

        - **Deterministic**: Uses characteristics of processes and processors to allocate tasks.
        - **Probabilistic**: Uses static attributes of the system, such as the number of nodes and their processing capability.

2. **Dynamic Load Balancing**:
    - These algorithms use the current state of the system to make load-balancing decisions.
    - They require more overhead to collect state information but perform better than static algorithms.
    - **Types**:
        - **Centralized**: One node collects all system information and makes scheduling decisions.
        - **Distributed**: Each node makes decisions based on its local state and information received from other nodes. This is the preferred approach.
            - **Cooperative**: Nodes work together to make decisions, leading to better stability but higher complexity.
            - **Non-Cooperative**: Nodes make independent decisions without consulting others, leading to less overhead but potentially less stability.

Q. Explain designing issues in load balancing approach

Load Balancing is used in distributed systems to make sure all nodes (computers) share work equally. The goal is to avoid overloading any single node and to improve system performance.

Main Goals of Load Balancing:

- Use resources effectively
- Increase system speed (throughput)
- Reduce response time
- Avoid overloading any one node

Designing Issues in Load Balancing:

1. Load Estimation Policy:

- This policy decides **how to measure the workload** on a node (computer).
- It is hard to measure workload perfectly, but it can be estimated using:
    - Number of running processes
    - Resource needs of those processes
    - Type of tasks being run
    - CPU speed and architecture of the node

2. Process Transfer Policy:

- This policy decides **when a process should be moved** from one node to another.
- If a node is **too busy**, some processes are sent to **less busy** nodes.

Types:

- **Static policy**: Fixed limit to decide when a node is overloaded.
- **Dynamic policy**: The limit changes based on system conditions.

Threshold Policies:

- **Single threshold**: One limit used for both sending and receiving processes.
- **Double threshold (High-Low policy)**: Two limits — below one, node accepts work; above the other, node offloads work. More stable than single threshold.

3. Location Policy:

- Once it's decided that a process should move, this policy **selects the best node** to move it to.

Methods:

- **Threshold**: Try random nodes until one accepts the process.
- **Shortest method**: Pick a few random nodes and choose the one with the least load.
- **Bidding**: Nodes act like managers and contractors — one sends, another accepts.
- **Pairing**: Nodes form pairs and balance load between each other.

4. State Information Exchange Policy:

- This policy handles **how and when nodes share their load status** with each other.

Types:

- **Periodic broadcast**: Every node sends its status after fixed time intervals.
- **On state change**: Nodes send updates only when their load changes.
- **On demand**: Nodes ask for others' status only when they need to offload work.

- **Polling**: A node asks other nodes one by one until it finds a suitable one.

5. Priority Assignment Policy:

- Decides which process (local or remote) should get **higher priority** for execution.

Types:

- **Selfish rule**: Local processes get more priority.
- **Altruistic rule**: Remote processes get more priority.
- **Intermediate rule**: Priority depends on how many local and remote processes are on the node.

6. Migration Limiting Policy:

- This policy sets **how many times a process can move** from one node to another.

Types:

- **Uncontrolled**: Process can move as many times as needed.
- **Controlled**: Number of migrations is limited using a counter.

Q. Explain Load sharing approach in distributed system.

Of course! Here's the explanation **without any symbols** and written in a simple, student-friendly way:

Q. Explain Load Sharing Approach in Distributed Systems (Simple Explanation)

What is Load Sharing?

- In a distributed system, it's not always necessary to divide the workload equally.
- It is more important to make sure that no node is idle when others are heavily loaded.
- This approach is called **dynamic load sharing**, and it is often simpler and more efficient than full load balancing.

Why Load Sharing is Important

- Helps in better use of system resources.
- Keeps all the nodes active and useful.
- Easier to implement than complex load balancing.

Designing Issues in Load Sharing

1. **Load Estimation Policy**
   - It checks whether a node is busy or idle.
   - The most common and simple method is counting how many processes are running on a node.
   - This helps decide if the node should send or receive work.
2. **Process Transfer Policy**
   - Follows an "all-or-nothing" strategy.
   - If a node has more than one task, it becomes a sender.
   - If a node has no task, it becomes a receiver.
   - To avoid overuse of idle nodes, some systems use two thresholds.
3. **Location Policy**
   - Decides who starts the search for another node to send or receive tasks.

a. **Sender-Initiated**

- A busy node looks for an idle one.
- It can broadcast or check nodes one by one.

b. **Receiver-Initiated**

- An idle node looks for a busy one to receive tasks from.
- It can also broadcast or check nodes one by one.

4. **State Information Exchange Policy**
   - Nodes don’t need to regularly send updates.
   - They only need to know the state of other nodes when they become overloaded or underloaded.

a. **Broadcast When State Changes**

- Node sends its status when it becomes busy or idle.
- In receiver-initiated systems, this is called "broadcast-when-idle".

b. **Poll When State Changes**

- Node randomly asks other nodes for their status.
- In receiver-initiated systems, this is called "poll-when-idle".

Q. Explain Task Assignment and Job Scheduling

Task Assignment

- Think of a **process** as a **group of small tasks**.
- In **task assignment**, we assign each task to a processor that can handle it best.

Why it's not used often:

- You must know all the details of each process **beforehand**.
- It doesn't handle changes in the system **during runtime**.

Goals of Task Assignment:

1. **Reduce communication cost** between tasks (like less data transfer).
2. **Use resources** (CPU, memory) efficiently.
3. **Finish the process faster** (quick turnaround time).
4. Run many tasks **in parallel**.

Job Scheduling

- **Job scheduling** is about deciding **which task runs next** and **for how long**.
- Jobs are placed in **queues** and wait their turn to use the CPU.

Main Goals:

- Make sure every job is handled **fairly**.
- Keep everything running **on time and smoothly**.

How it works:

- Most systems (like **Windows** or **Linux**) have built-in job schedulers.
- These job schedulers can start, stop, and manage jobs **automatically**.
- Jobs can also be tracked or controlled **manually** if needed.

Job scheduling considers factors like:

- **Priority** of the job.
- Whether the needed **resources** (like CPU or memory) are free.
- **How much time** the user has been given.
- **How many jobs** a user is allowed to run.
- **How long** the job will take or has already taken.

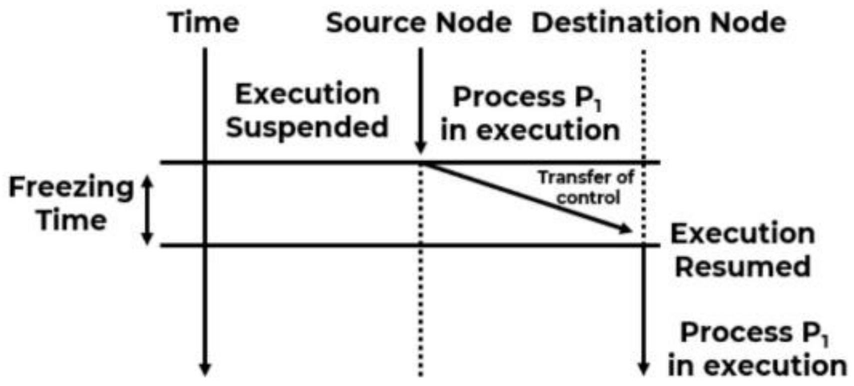
Q. Compare load sharing, task assignment, and load balancing strategies for scheduling processes in a distributed system.

Parameter	**Load Balancing**	**Load Sharing**	**Task Assignment**
Goal	Balance the load across all nodes in the system.	Prevent nodes from being idle while others are overloaded.	Assign tasks to the best processors for better performance.
Focus	Equal distribution of workload.	Ensure no node is idle while others are overloaded.	Assign each process's tasks to suitable processors.
Main Objective	- Optimize resource use. - Maximize throughput. - Minimize response time. - Avoid overloading nodes.	- Prevent idle nodes. - Keep all resources in use.	- Minimize communication costs. - Efficient resource use. - Fast turnaround. - High parallelism.
Process Distribution	Distributes tasks evenly to balance load.	Prevents under-utilized nodes, but doesn't fully balance load.	Assigns individual tasks of a process to different nodes.
Type of Algorithms	Static or Dynamic.	Mainly Dynamic.	Task-based, typically static but can adapt to changes.
Overhead	High (especially for dynamic algorithms).	Lower than load balancing (simple estimation of idle vs busy nodes).	High due to task-specific decisions and advanced planning.
System State Information	Uses system state information for better distribution.	Only needs to know if a node is idle or busy.	Requires knowledge of all process characteristics in advance.
Efficiency	More efficient at balancing load.	Focuses on keeping all nodes active, not balancing load exactly.	Efficient resource utilization for each process, but may not adapt to system changes.

Parameter	Load Balancing	Load Sharing	Task Assignment
Example Use Cases	Data center management, cloud systems.	Distributed computing with minimal load balancing.	High-performance computing, multi-task parallelism.
Communication	Often requires communication between nodes to adjust load.	Less frequent communication but uses a simple strategy to avoid idle nodes.	Needs communication between tasks for optimal assignment.
Complexity	Can be complex (especially dynamic types).	Simpler and less complex.	More complex due to the need for detailed task knowledge.
Adaptability	Adapts to system state changes (especially in dynamic versions).	Adapts to prevent idle nodes but doesn't strive for perfect balance.	Doesn't adapt well to changing system conditions (requires prior knowledge).

Q. What is the need for process migration and explain the role of resource to process and process to resource binding in process migration

Process migration is the act of moving a process from one computer (node) to another in a distributed system. This is done to improve resource usage, balance the load, and enhance system performance.



Why is Process Migration Needed?

1. **To balance the load**: Avoid overloading one system while others remain idle.

2. **To improve resource utilization**: Move processes closer to the resources they need.
3. **Fault recovery**: Move processes away from a failing or overloaded system.
4. **Performance optimization**: Optimize system performance by distributing tasks.

Types of Process Migration:

Non-preemptive process migration: The process is moved before it starts running. This is simpler and less costly.

Preemptive process migration: The process is moved while it is running. This is more complex and costly because the state of the process must be transferred with it.

Steps in Process Migration:

1. **Select a process**: Choose which process to migrate.
2. **Choose a destination node**: Decide which node the process will move to.
3. **Transfer the process**: Move the process to the new node and restart it there.

What Needs to Be Handled During Migration:

- **Freezing the process**: Pause the process on the source node.
- **Transfer the process state**: Move the process's memory, data, and execution context to the new node.
- **Forward messages**: Ensure messages meant for the process are forwarded correctly.
- **Maintain communication**: Keep communication intact with other processes that are involved.

Features of a Good Process Migration System:

1. **Transparency**: The user should not notice that the process was moved.
2. **Minimal interference**: The migration should not disrupt the overall system.
3. **Efficiency**: The system should efficiently use resources during migration.
4. **Robustness**: The system should handle migration reliably even during failures or complex situations.

Resource-to-Process and Process-to-Resource Binding:

Resource-to-process binding: A resource (such as a device or file) is associated with a specific process.

Process-to-resource binding: A process requires certain resources to run.

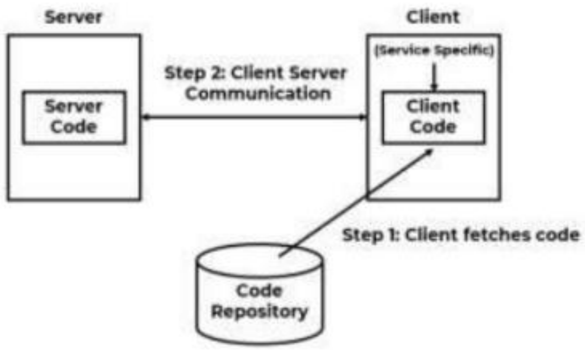
When migrating a process, the system must check if the required resources can also move with the process, or if they can be accessed from the new node. This ensures the migration is successful.

Q. Describe code migration in distributed systems.

Code Migration refers to the process of transferring a program or part of a program from one machine (node) to another in a distributed system. This is different from **process migration**, where an entire process is moved from one machine to another.

How Code Migration Works:

1. **Instead of moving the entire process**, only a part of the program (the code) is transferred.
2. **Example**: In a client-server system, where the server manages a large database, if the client needs to perform many database operations, it is more efficient to **move part of the client's code to the server** rather than sending the data back and forth.
3. This **reduces communication overhead** and can make the system more efficient.
4. Code migration can improve overall system performance by **exploiting parallelism** (by performing tasks on different machines at the same time).
5. **Advantages**:
   - Clients don't need to install all software; it can be transferred only when needed and discarded when not in use.



Issues in Code Migration:

1. **Communication in Distributed Systems**: The primary challenge is ensuring smooth data exchange between different machines.
2. **Process Segments**:
   - **Code Segment**: Contains the actual program code.
   - **Resource Segment**: Contains references to the resources the program needs.
   - **Execution Segment**: Stores the execution state of the process.

3. There are different models for code migration:
    - **Client-Server** (CS)
    - **Remote Evaluation** (REV): Used to perform operations like database processing.
    - **Code-on-Demand (CoD)**: Moves part of the server's code to the client.
    - **Mobile Agents (MA)**: A program that moves from one site to another to perform tasks.

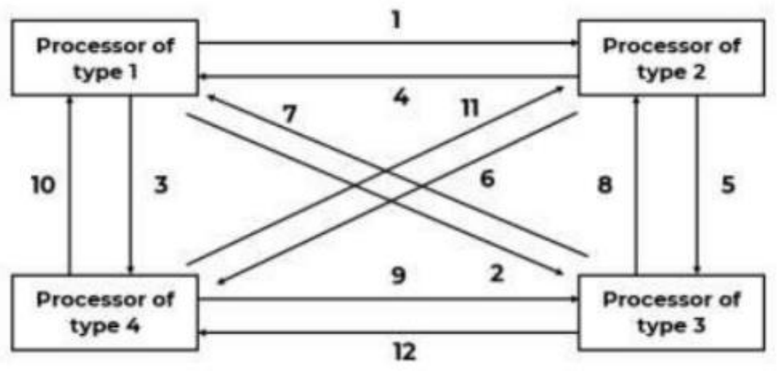
Example: In a code-on-demand system, part of the server's code can be migrated to the client for processing, making the system more efficient by reducing the amount of data transfer.

Q. Explain process migration in heterogeneous system

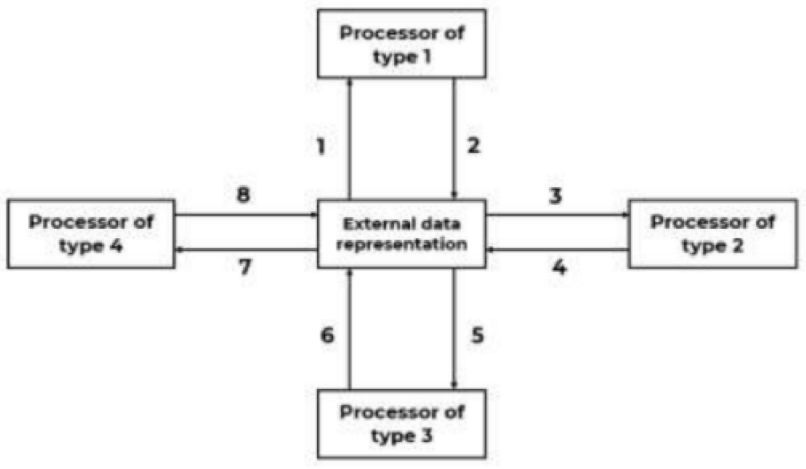
In a **heterogeneous system**, different computers (nodes) may use different CPU types or architectures. When migrating a process from one node to another, the system faces additional challenges compared to a **homogeneous system** where all nodes are the same.

Key Points of Process Migration in Heterogeneous Systems:

1. **Homogeneous systems**: In a homogeneous system (where all computers are identical), data interpretation is consistent, and no data translation is required when migrating processes.
2. **Heterogeneous systems**: In heterogeneous systems, data must be translated from the source CPU format to the destination CPU format before it can be executed.
3. **Translation challenge**: If a system has two types of CPUs, each processor needs to be able to convert data from one format to another. If more CPUs are added, more data translations are needed, leading to more overhead.
4. **Overhead due to data translation**: For example, if there are three CPU types in a system, processor 1 would need to translate data three times before sending it to the other processors. Each processor then needs to translate data back to its own format, adding to the overhead.
5. **Translation software**: To handle the conversion between different CPU formats, a heterogeneous system may need **n(n-1)** pieces of translation software, where **n** is the number of different CPU types in the system.



6. **Reducing complexity with external data representation**: To simplify the translation process, systems can use an **external data representation**. This is a standard format used for transferring data between nodes. Each process must convert its data to and from this standard format.
7. **Design issues**: When designing this translation system, issues like handling **signed infinity** and **signed zero** representations must be addressed.



Q. Explain threads

A **thread** is a smaller unit of a process that can execute tasks concurrently. It is often referred to as a **lightweight process** because it shares resources with other threads in the same process.

Key Points:

1. **What is a thread?**
   A thread is a flow of control within a process's address space. Multiple threads can exist within the same address space and share the same resources (like memory), but each has its own execution path.

2. **Multiple control flows**
   A single process can have multiple threads running simultaneously. Each thread has access to the entire address space, allowing it to share resources with other threads.

Why Use Threads?

1. **Easier creation**
   It's easier to create threads than to create multiple processes because threads share resources within the same process.
2. **Faster switching**
   Switching between threads is faster than switching between processes because threads share the same address space.
3. **Parallelism**
   Threads enable parallelism, meaning that different threads can perform tasks at the same time, improving performance.
4. **Better resource utilization**
   Since threads share resources, they are more efficient and require fewer resources compared to processes.

Thread Models

There are different models for managing threads in a process:

1. **Dispatcher-Worker Model:**
   - **How it works:**
     In this model, a process has one **dispatcher thread** and multiple **worker threads**.
     The dispatcher thread handles incoming client requests and assigns them to worker threads for processing.
     Each worker thread processes a different client request, so multiple requests can be handled simultaneously.

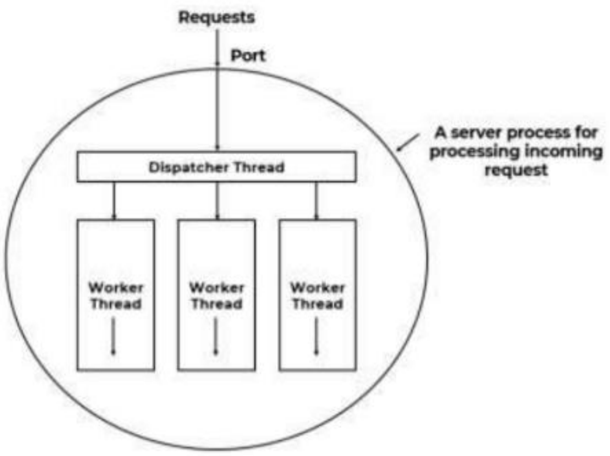


Figure 4.6: Dispatcher Worker Model

- **Example:**
  A web server could use this model to handle multiple client requests at the same time.

2. **Team Model:**
   - **How it works:**
     In the team model, all threads are equal, with no dispatcher-worker relationship.
     Each thread independently handles its own client requests and processes them.
     This model is useful when each thread has a specialized role.
   - **Example:**
     A process that handles different types of requests like file reading, data processing, and database queries could use this model, with each thread handling a specific task.

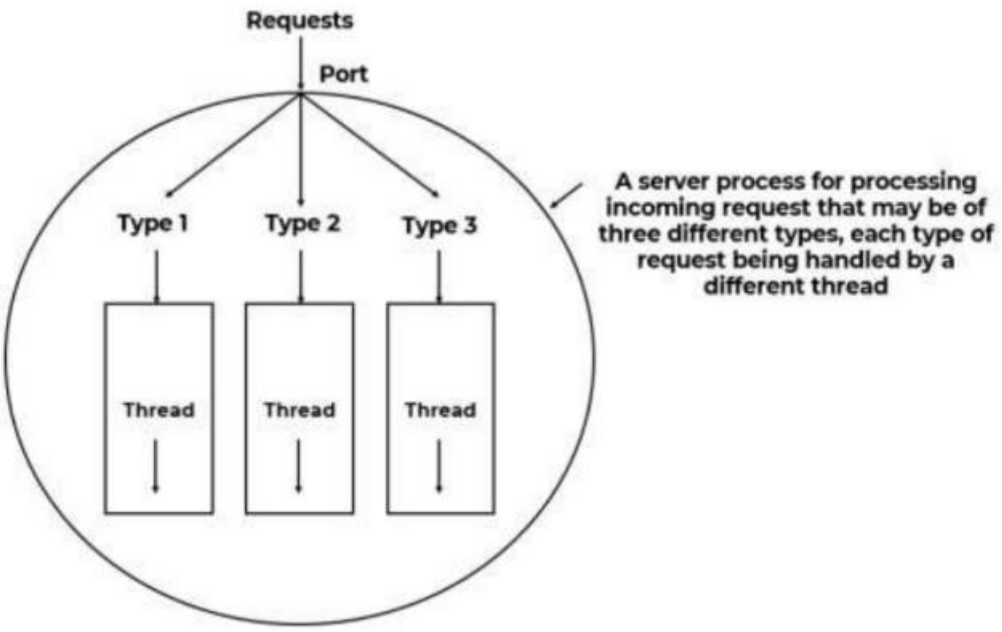

Requests
Port
Type 1
Type 2
Type 3
Thread
Thread
Thread
A server process for processing incoming request that may be of three different types, each type of request being handled by a different thread

Figure 4.7: Team Model

3. **Pipeline Model:**
   - **How it works:**
     The pipeline model is often used in **producer-consumer** scenarios.
     Each thread in the pipeline performs a part of a task, and the output of one thread becomes the input of the next.
     This allows for efficient processing, where each thread is like a stage in a production line.
   - **Example:**
     A video processing application where one thread decodes a video, another applies filters, and a third thread outputs the final result.

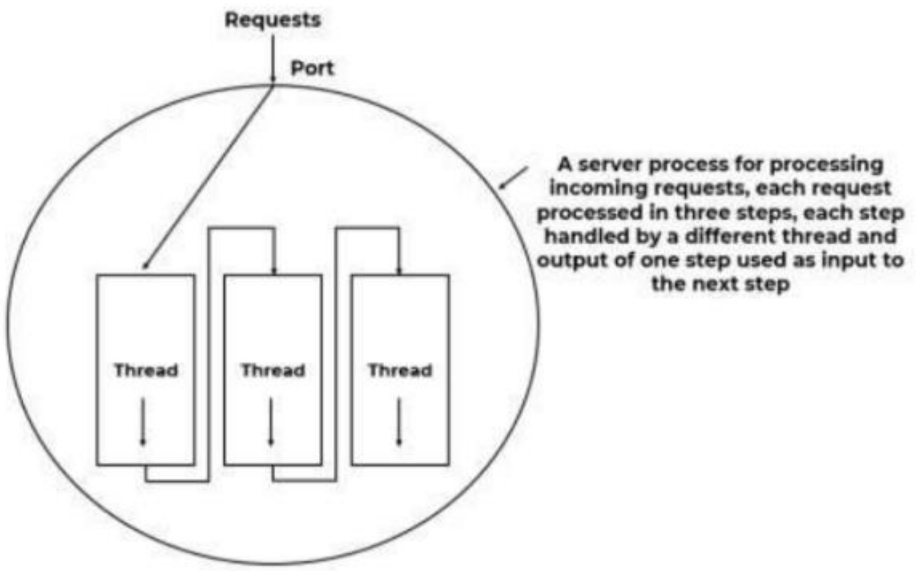

Requests
Port
A server process for processing incoming requests, each request processed in three steps, each step handled by a different thread and output of one step used as input to the next step
Thread
Thread
Thread

Figure 4.8: Pipeline Model

Q. Compare processes and threads.

Parameter	Process	Thread
Definition	A process is the basic unit of CPU work in traditional operating systems.	A thread is the basic unit of CPU work in systems with multi-threading.
Memory and Resources	Each process has its own program counter, registers, stack, and memory space.	Each thread has its own program counter, registers, and stack, but shares memory space with other threads in the same process.
Protection	Processes need protection because they do not share memory.	Threads do not need protection within the same process since they share memory.
Type	Processes are heavyweight because they use more resources.	Threads are lightweight and use fewer resources.
Resource Sharing	Less efficient in sharing resources, as processes do not share memory directly.	More efficient because threads share memory and resources within the same process.

Parameter	Process	Thread
Switching Cost	Switching between processes is more expensive due to independent memory and resources.	Switching between threads is cheaper as threads within a process share memory.
Overhead	Creating a new process requires more resources and time due to its independent nature.	Creating a new thread requires fewer resources and time because threads share resources.
Communication	Inter-process communication (IPC) is needed and is more complex and slower.	Communication between threads is easier and faster since they share the same memory space.
Concurrency	Multiple processes can run concurrently but are isolated from each other.	Multiple threads within a process can run concurrently and share data easily.
Fault Isolation	Processes are isolated, so if one process crashes, others are unaffected.	If a thread crashes, it can affect other threads in the same process since they share the same memory space.
Control	Processes are independent and can run in different address spaces.	Threads are part of a process and cannot run outside the context of that process.
Execution	Processes can run on different machines or systems.	Threads within the same process run on the same machine/system.
Context Switching	Context switching between processes is costly, as it involves saving and restoring the entire process state.	Context switching between threads is cheaper, as only the thread-specific state (like program counter) needs to be saved and restored.

Q.