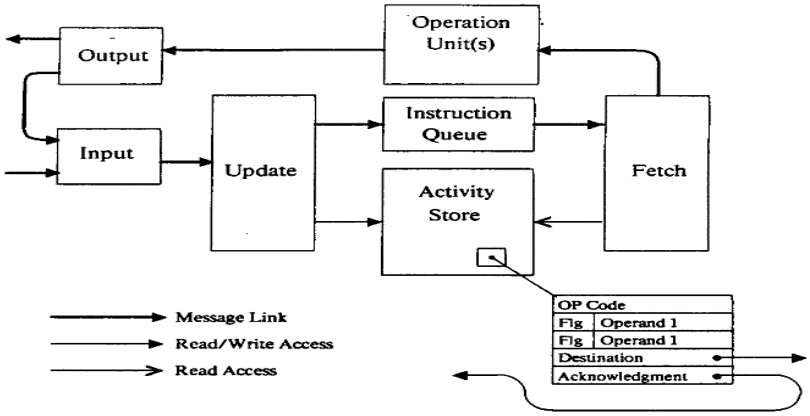
1. STATIC DATA FLOW ARCHITECTURE [UNIT 5]

Static dataflow architecture is a type of dataflow architecture where the ***flow of data*** is determined by the ***static dataflow graph (SDF).*** An SDF is a ***directed acyclic graph (DAG)*** that represents the ***data dependencies*** between the different operations in the system. The ***nodes*** in the SDF represent the ***different operations*** that are performed on the data, and the ***edges*** represent the ***data dependencies*** between the operations.



Components of Static Dataflow Architecture:

1. **Input:** Represents the ***initial data or operands*** required for processing instructions.
2. **Update Unit:** ***Manages the availability of input data, updating the activity status*** of instructions. Connected to the ***Instruction Queue*** and ***Activity Store.***
3. **Fetch Unit:** Retrieves instructions from the Instruction Queue and forwards them to the Operation Unit for execution. Facilitates the ***seamless transfer of instructions.***
4. **Operation Unit:** Executes instructions, carrying out ***computational and operational tasks*** specified by the program. Where the actual processing takes place.
5. **Activity Store:** Repository for storing the ***status and availability of instructions and associated input data.*** Coordinates the ***flow of instructions and data*** within the architecture.
6. **Output:** Represents the output data generated as a result of instruction execution. Signifies the ***completion of processing*** and the ***production of the desired output.***

Working of Static Dataflow Architecture:

1. ***Graph Initialization:***
   - The static dataflow graph (SDF) is pre-established, representing ***data dependencies and operations.***
2. ***Input Processing:***
   - Input component provides initial data or operands for processing instructions.
3. ***Instruction Flow:***
   - Update Unit manages the ***availability of input data.***
   - Fetch Unit retrieves instructions from the Instruction Queue.
   - Operation Unit executes instructions based on ***predetermined data dependencies.***

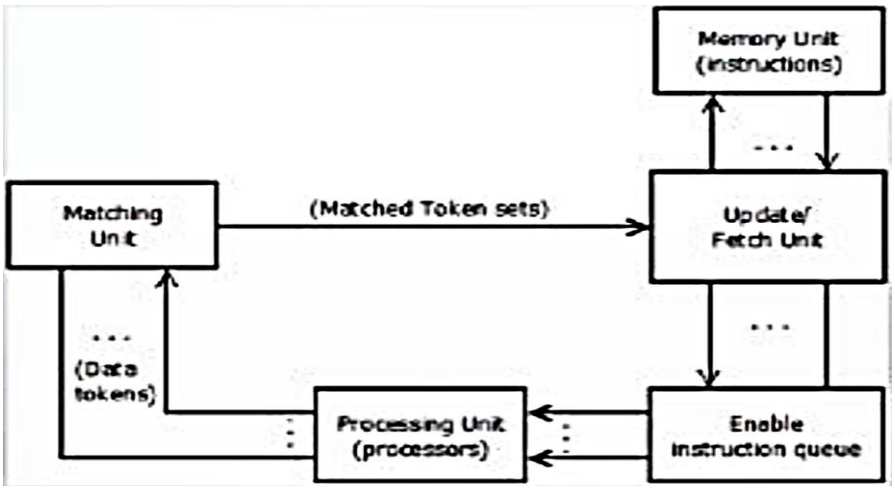
4. ***Activity Status Management:***
    - Update Unit ensures that the necessary input data is available for instruction execution.
    - Activity Store maintains the ***status and availability of instructions and associated input data.***
5. ***Output Generation:***
    - Operation Unit processes instructions, producing output data.
    - Output component represents the final result of the executed instructions.

Key Characteristics:

- ***Predetermined Flow:*** The dataflow graph is static and predetermined before program execution.
- ***Fixed Dependencies:*** Data dependencies between operations are fixed throughout the program.

Static Dataflow Architecture provides a ***deterministic and structured*** approach to data processing, ensuring that the ***order of execution and data dependencies remain constant throughout program execution.***

2. WHAT IS DYNAMIC DATAFLOW ARCHITECTURE?



Dynamic Dataflow Architecture is a computing framework where the ***flow of data determines the execution of tasks,*** and this ***flow can adapt and change during program execution.*** Unlike static dataflow architectures with fixed structures, dynamic dataflow allows the ***creation and modification of the dataflow graph*** as the program runs.

Components and Working of Dynamic Dataflow Architecture:

1. **Matching Unit:**
    - Receives data tokens from input channels
    - Matches data tokens to corresponding processing units
2. **Processing Units:**
    - Perform various operations on the received data tokens

- ***Store intermediate results*** in the memory unit

3. **Enable Instruction Queue:**
   - Queue of instructions ready for being processed by processing units
   - Guides processing units on what operations to perform
4. **Memory Unit:**
   - Stores intermediate results produced by processing units
   - ***Temporary storage for data tokens*** awaiting further processing
5. **Updater:**
   - ***Updates the enable instruction queue*** based on the ***availability of data tokens*** in the memory unit
   - Informs enable instruction queue when processing units have completed their operations and new data tokens are available

Applications:

Dynamic Dataflow Architecture finds applications in scenarios where the structure of data processing tasks changes during runtime. Some examples include:

1. **Media Processing:**
   - Efficient handling of dynamic data streams in multimedia applications.
2. **Digital Signal Processing:**
   - Real-time signal processing where the flow of data can vary.
3. **Adaptive Algorithms:**
   - Implementing algorithms that dynamically adjust based on changing data requirements.

This architecture's ***flexibility and adaptability*** make it well-suited for scenarios where computational requirements are dynamic and variable, allowing for ***efficient resource management and responsive task execution.***

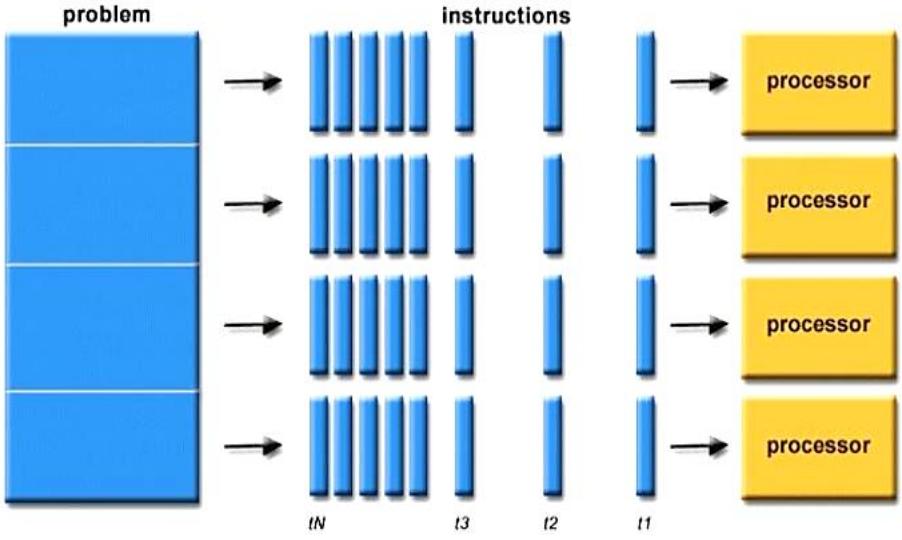
03. DISTINGUISH BETWEEN STATIC AND DYNAMIC DATAFLOW ARCHITECTURE.

Aspect	**Static Dataflow Architecture**	**Dynamic Dataflow Architecture**
Graph Structure	Static dataflow graph (SDF), a fixed DAG.	The dataflow graph is ***dynamic*** and ***can change during program execution***.
Components	Input - Update Unit - Fetch Unit - Operation Unit - Activity Store - Output	Matching Unit - Processing Units - Enable Instruction Queue - Memory Unit - Updater
Initialization	Graph is ***pre-established*** before program execution.	Graph can adapt and change during runtime.
Data Dependency	***Fixed data dependencies*** between operations.	***Flexible data dependencies*** that can change dynamically.

Aspect	Static Dataflow Architecture	Dynamic Dataflow Architecture
Processing Order	Order of execution and data dependencies remain ***constant.***	***Order of execution can change*** based on dynamic dataflow.
Use Cases	Well-suited for scenarios with ***static computational requirements.***	Suited for scenarios where ***data processing tasks change dynamically during runtime.***
Applications	Signal Processing, Scientific Computing, Business Data Processing	Digital Signal Processing, Graphics Processing Units, AI, Network Routing

04. EXPLAIN DIFFERENT TYPES AND LEVELS OF PARALLELISM.

Parallelism is the concept of executing multiple tasks or instructions simultaneously, either within a single processor or across multiple processors, to improve ***computational efficiency and speed.*** It involves ***breaking down a task into smaller parts*** that can be performed simultaneously, allowing for faster execution and increased throughput.



Different Types and Levels of Parallelism

1. **Instruction Level Parallelism:**
    - This type of parallelism involves ***breaking down a program into smaller units of instructions*** that can be executed concurrently.
    - It operates at a ***fine-grained level***, typically involving ***less than 20 instructions per grain.***
    - The benefits are having ***many options for parallel execution*** and enabling compilers to recognize and utilize this parallelism.

2. **Loop-level Parallelism:**
    - Loop-level parallelism focuses on parallelizing loops within a program.
    - A common loop has less than 500 instructions, and ***if the loop operations are independent between iterations,*** they can be executed simultaneously.
    - Loop-level parallelism is considered ***fine-grained computation*** and can be efficiently handled by ***pipelines or SIMD (Single Instruction, Multiple Data) machines.***
    - However, certain types of loops, such as ***recursive loops,*** can be challenging to parallelize.
3. **Procedure-level Parallelism:**
    - Procedure-level parallelism involves parallelizing ***medium grains of code,*** typically comprising ***fewer than 2000 instructions.***
    - Detecting parallelism at this level is more challenging than with fine grains because of ***intraprocedural dependence analysis.***
    - ***Communication requirements are lower*** compared to instruction level parallelism.
    - ***Multitasking systems*** often fall into this level of parallelism.
4. **Subprogram-level Parallelism:**
    - Subprogram-level parallelism deals with ***larger grains of code,*** typically consisting of ***thousands of instructions.***
    - At this level, parallelism is often achieved through ***multiprogramming,*** where ***multiple programs or subprograms are executed simultaneously.***
    - Multi-programming systems make use of parallelism at the subprogram level to improve overall system efficiency.
5. **Job or Program-Level Parallelism:**
    - Job or program-level parallelism involves ***executing independent jobs or programs concurrently*** on a parallel computer.
    - This level of parallelism is practical for machines with a small number of powerful processors.
    - However, it becomes impractical for machines with a large number of simple processors because each processor would take too long to process a single job.

Grain Size: At a very basic level, **grain size** in parallel computing refers to the ***size of the tasks that can be done at the same time.*** It's like breaking a big task into smaller parts.

- If the parts are small, it's called **fine grain**.
- If the parts are a bit bigger, it's **medium grain**.
- If the parts are large, it's **coarse grain**.
- **Fine-grained parallelism:** This refers to parallelism at the ***level of individual instructions or small code segments.*** It offers a ***high level of parallelism*** but may ***additional overhead due to synchronization and communication*** between parallel units.
- **Medium-grained parallelism:** This involves parallelism at the level of ***loops, procedures, or subprograms.*** It strikes a balance between fine-grained and coarse-grained parallelism, offering a ***moderate level of parallelism*** while reducing synchronization and communication overhead.
- **Coarse-grained parallelism:** This refers to parallelism at the level of larger program segments or ***independent jobs.*** It involves ***executing independent programs or jobs concurrently,*** offering ***lower parallelism*** but reducing the need for synchronization and communication between parallel units.

05. EXPLAIN TYPES OF DATA DEPENDENCIES WITH EXAMPLE.

Data Dependencies: Data dependencies refer to the relationships between instructions in a program based on the data they share. when a program instruction refers to the data of a previous instruction. This means that an instruction depends on the results of a previous instruction.

let's delve into the different types of data dependencies that exist in parallel computing:

1. **Flow dependence/ True dependence/ Read After Write (RAW)**
    - **Description:** A flow dependence occurs when a ***statement S2 depends*** on the ***result of statement S1.*** Essentially, an output of S1 is used as an input by S2.
    - **Example:** In the case of

      S1: Load R1, A

      S2: ADD R2, R1

      S2 is flow dependent on S1 because it uses the result of S1 (the value A loaded into R1).
2. **Anti-dependence/ Write After Read (WAR)**
    - **Description:** An anti-dependence occurs when a ***statement S2 modifies a value that S1 reads,*** and S2 is executed after S1.
    - **Example:** Given

      S1: ADD R2, R1 ***(where R1 is an input)*** and

S2: Move R1, R3 (where R1 is an output), S2 is anti-dependent on S1 because it modifies R1, which was read by S1.

3. **Output dependence/ Write After Write (WAW)**

   - **Description:** An output dependence exists when ***two or more statements write to the same memory location or variable.*** The order of these statements can impact the final result.
   - **Example:** Given

     S1: Load R1, A and

     S2: Move R1, R3, both S1, and S2 write to the same output variable (R1), causing output dependence.

4. **I/O dependence**

   - **Description:** I/O dependence arises not from the use of the same variable but from the use of the ***same file or device by different I/O statements.***
   - **Example:** With

     S1: Read(4), A(I) and

     S2: Write(4), A(I), S1 and S2 depend on each other because of they are using same ***file descriptor (4)*** and the same ***array element (A(I)).***

5. **Unknown dependence**

   - **Description:** An unknown dependence exists when the ***system cannot determine the relationship between two statements,*** frequently due to a ***lack of specific data*** or ***complex memory access patterns.***
   - **Example:**

     ```
     x = 0
     y = 0

     # Statement 1: Update x based on y
     x = y + 1

     # Statement 2: Update y based on x
     y = x * 2
     ```

In conclusion, understanding data dependencies is vital ***when parallelizing computations,*** as they can affect computation correctness. ***The primary goal is to ensure that the parallel execution of program reproduces correct result, as a sequential version would.***

06. EXPLAIN WHAT IS DATA DEPENDENCY? EXPLAIN BERNSTEIN'S CONDITION FOR ANALYSIS OF DATA DEPENDENCY.

Data Dependency: Data dependency refers to the relationship between different instructions or statements in a program based on the data they share. It determines the order in which instructions need to be executed to ensure correct results.

There are three main types of data dependencies: ***flow dependence, anti-dependence, and output dependence.***

Bernstein's Conditions: Bernstein's conditions are a set of conditions that must be satisfied for two processes to be able to execute in parallel. These conditions help determine if parallel execution of instructions is safe and will not lead to incorrect results. The ***conditions are based on data dependencies between processes.***

Notation:

- ***Ii represents the set of all input variables for a process Pi.***
- ***Oi represents the set of all output variables for a process Pi.***

Bernstein's conditions state that if two processes, P1 and P2, can execute in parallel (P1 || P2), the following conditions must hold:

1. **Flow Independence:**
    - P1 and P2 should not have any flow dependence between them.
    - Flow dependence occurs when the output of one process is used as input by the other process.
    - Mathematically: $O_1 \cap I_2 = \emptyset$
2. **Anti-Independence:**
    - P1 and P2 should not have any anti-dependence between them.
    - Anti-dependence occurs when the ***output of one process overlaps with the input of the other process.***
    - Mathematically: $O_2 \cap I_1 = \emptyset$
3. **Output Independence:**
    - P1 and P2 should not have any output dependence between them.
    - Output dependence occurs when ***both processes write to the same output variable.***
    - Mathematically: $O_1 \cap O_2 = \emptyset$

It is important to note that the ***parallelism relation (||) is commutative,*** meaning that if P1 can execute in parallel with P2, then P2 can also execute in parallel with P1. However, the ***parallelism relation is not transitive,*** meaning that if P1 can execute in parallel with P2 and P2 can execute in parallel with P3, it does not imply that P1 can execute in parallel with P3.

By satisfying Bernstein's conditions, programmers can ensure that ***parallel execution of processes is safe and does not introduce data dependencies*** that could lead to incorrect results. It allows for efficient utilization of parallel computing resources and improved performance in parallel programs.

07. EXPLAIN HARDWARE AND SOFTWARE PARALLELISM.

Hardware Parallelism:

1. **Definition:**
    - Hardware parallelism is a ***characteristic of machine architecture*** that determines the ***simultaneous execution of multiple instructions.***
2. **Characterization:**
    - It is often characterized by the ***number of instructions*** a processor can issue in a ***single machine cycle.*** For instance, a processor that can issue k instructions per cycle is referred to as a k-issue processor.
    - Conventional processors are typically one-issue processors, meaning they can execute one instruction per machine cycle.
3. **Examples:**
    - ***Intel i960CA:*** This processor is an example of a ***three-issue processor.*** It can simultaneously handle ***arithmetic, memory access, and branch instructions*** in a single cycle.
    - ***IBM RS-6000:*** This processor is a ***four-issue processor,*** capable of executing ***arithmetic, floating-point, memory access, and branch instructions*** concurrently.

Software Parallelism:

1. **Definition:**
    - Software parallelism is determined by the ***control and data dependencies*** within programs. It is revealed in a ***data flow graph*** and is influenced by the ***algorithm, programming style,*** and ***compiler optimizations.***
2. **Control and Data Dependencies:**
    - **Control dependence** is associated with the order of execution of instructions based on conditional branches and control flow within the program.
    - **Data dependence** is related to the ordering relationship between statements and the use of variables.
3. **Data Flow Graph:**
    - Software parallelism can be visualized in a data flow graph, illustrating the dependencies within the code.

4. **Factors Influencing Software Parallelism:**

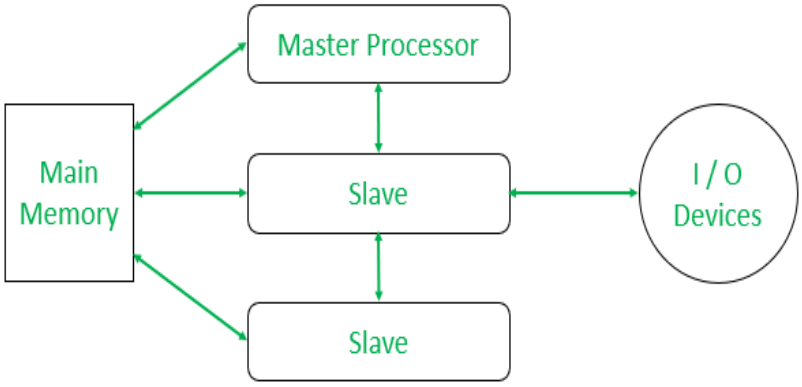
    - ***Algorithm:*** The design of the algorithm plays a crucial role in determining the potential for parallelism.
    - ***Programming Style:*** The way code is written and organized can impact the identification and exploitation of parallelism.
    - ***Compiler Optimization:*** Compilers can analyse code to identify opportunities for parallel execution and optimize accordingly.

In summary, hardware parallelism is a property of the ***machine architecture,*** while software parallelism is influenced by the ***design of algorithms, coding practices,*** and the ***ability of compilers to identify and exploit parallel execution opportunities.*** Both aspects are essential for achieving efficient parallel processing in computing systems.

08. DISCUSS IN DETAIL MODELS OF PARALLEL OPERATING SYSTEMS.

There are three models of parallel operating systems that have been employed in the design of operating systems for multiprocessors:

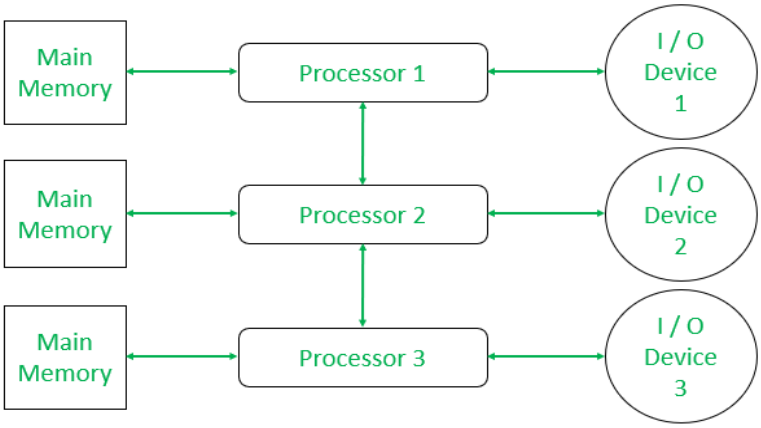
1. **Master-Slave Configuration:**



- In the master-slave model, one processor, known as the ***master, maintains the status of all the other processors*** in the system and ***distributes work to the slave processors.***
- The operating system runs only on the master processor, while the other processors are treated as ***schedulable resources.***
- Other processors needing ***executive services*** must request the master, which acknowledges the request and performs the services.
- This scheme is a simple ***extension of a uniprocessor operating system*** and is relatively easy to implement.
- However, this scheme makes the system highly susceptible to failures, as the system heavily relies on the master processor. If the master fails, the entire system may be affected.

- Additionally, ***many of the slave processors have to wait for the master's work to be completed*** before their requests can be served.

2. **Separate Supervisor Configuration:**



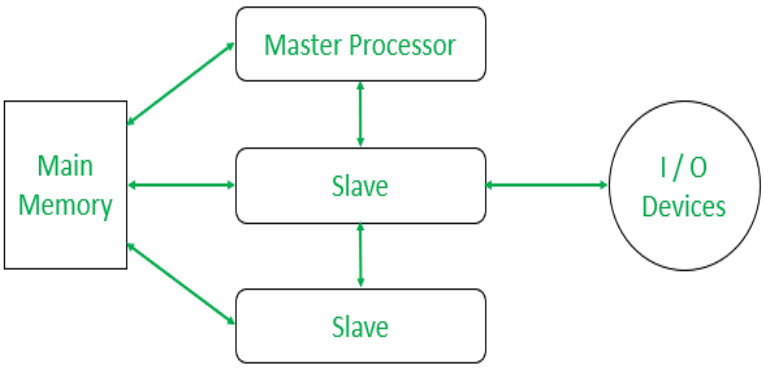
- In this approach, each processor contains a copy of the kernel or operating system.
- ***Resource sharing occurs through shared memory blocks.***
- Each processor ***services its own needs*** and ***executes its own instance of the operating system.***
- This configuration is ***less susceptible to failures*** compared to the master-slave configuration.
- However, this approach also has some drawbacks. First, it ***requires more memory*** because each processor has its own copy of the kernel. Second, it can be more difficult to manage the system because ***each processor needs to be kept up-to-date with the state of the other processors.***

3. **Floating Supervisor Control:**

- In this model, the ***supervisor routine floats or moves from one processor to another,*** while several processors may be executing their own supervisory service routines simultaneously.
- This model offers better ***load balancing,*** as the supervisor routine can ***dynamically distribute tasks*** among processors.
- There is also a certain degree of ***code sharing*** in this model, ***some parts of supervisor code shared,*** which must be executed by multiple processors concurrently without interference.

These models of multiprocessors operating systems aim to provide ***efficient resource utilization, workload distribution,*** and ***fault tolerance*** in a multiprocessor environment. The choice of the model depends on factors such as the system's ***scalability requirements, fault tolerance needs,*** and the ***level of resource sharing*** among processors.

09. DESCRIBE MASTER SLAVE CONFIGURATION OPERATING SYSTEM FOR PARALLEL PROCESSING.



In a master-slave configured operating system for multi-processors, one processor is designated as the master, while the remaining processors are considered as slaves. The master processor takes on the role of ***managing the system and distributing tasks*** to the slave processors.

Here is a detailed description of the master-slave configuration operating system:

1. **Master Processor:**
    - The master processor is responsible for maintaining the overall ***system status*** and ***controlling the execution of tasks.***
    - It runs the operating system and handles ***system-level*** operations, such as ***task scheduling, resource allocation, and synchronization.***
    - The master processor receives requests from the slave processors and ***provide necessary executive services*** to them.
    - It distributes tasks among the slave processors based on ***workload balancing strategies.***
2. **Slave Processors:**
    - The slave processors are considered as ***schedulable resources*** in this configuration.
    - They perform the assigned tasks received from the master processor.
    - The slave processors ***execute their tasks independently*** without direct involvement from the master processor.
    - They may communicate with the master processor to ***request services or report task completion.***
3. **Task Distribution:**
    - The master processor determines the distribution of tasks among the slave processors.

- It considers factors such as ***workload, processor capabilities,*** and ***task dependencies*** when assigning tasks.
- The master processor may use ***load balancing algorithms*** to distribute tasks evenly across the slave processors, ensuring ***optimal resource utilization.***

4. **Communication and Synchronization:**
    - Communication between the master and slave processors is essential for ***coordination and exchanging information.***
    - The master processor may use ***message passing or shared memory mechanisms*** to communicate with the slave processors.
    - Synchronization mechanisms, such as ***locks, barriers, or semaphores,*** are employed to ensure proper coordination and avoid conflicts between processors.

Advantages of Master-Slave Configuration:

- **Simplicity:** The master-slave configuration is relatively straightforward to implement compared to other parallel system models.
- **Task Management:** The master processor has ***centralized control over task distribution,*** allowing for efficient workload balancing.
- **System Monitoring:** The master processor can monitor the system's status and performance, making it ***easier to detect and handle failures.***
- **Scalability:** The master-slave configuration can scale well with the addition of more slave processors, increasing overall system performance.

Disadvantages of Master-Slave Configuration:

- **Single Point of Failure:** The master processor represents a single point of failure. If it fails, the entire system may be affected.
- **Waiting Time:** Slave processors may have to wait for the master's services, leading to potential delays in task execution.
- **Limited Parallelism:** As the master processor handles system-level operations, it may introduce overhead and limit the overall parallelism achievable in the system.

Overall, the master-slave configuration operating system offers a simple and ***centralized approach for managing parallel tasks,*** but it also presents challenges related to fault tolerance and potential performance limitations.