Distributed System Management

Introduction

- Distributed systems have multiple resources and hence, there is a need to provide systems transparency
- Distributed systems management ensures that large distributed systems can function in accordance with the objectives of their users.
- System management can be categorized into resource management, process management, and fault tolerance.
- It also involves processor allocation, which deals with deciding which process should be assigned to which processor
- Once a set of processes are assigned to a particular processor the choice of a good scheduling algorithm will decide the order in which the processor will execute the processes

Types resource management techniques

- Task assignment approach
  - User processes are collections of related tasks
  - Tasks are scheduled to improve performance
- Load-balancing approach
  - Tasks are distributed among nodes so as to equalize the workload of nodes of the system
- Load-sharing approach
  - Simply attempts to avoid idle nodes while processes wait for being processed

Desirable Features of Global Scheduling Algorithm

- No A Priori knowledge about the Processes
- Ability to make dynamic scheduling decisions
- Flexible
- Stable
- Scalable
- Unaffected by system failures

No A Priori knowledge about the Processes

- In computing, scheduling is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). This is usually done to load balance and share system resources effectively or achieve a target quality of service.
- A good process scheduling algorithm should operate with absolutely no a priori knowledge about the processes to be executed. Since it places extra burden on the user to specify this information before execution

Ability to make dynamic scheduling decisions

- A good process scheduling algorithm should be able to take care of the dynamically changing load (or status) of the various nodes of the system.
- Process assignment decisions should be based on the current load of the system and not on some fixed static policy.

Flexible

- The algorithm should be flexible enough to migrate the process multiple times in case there is a change in the system load.
- The algorithm should be able to make quick scheduling decisions about assigning processes to processors.

Stability

- The algorithm must be stable such that processors do useful work, reduce thrashing overhead and minimize the time spent in unnecessary migration of the process.

- **Example:** it may happen that node *n*1 and *n*2 both observe that node *n*3 is idle and then both offload a portion of their work to node *n*3 without being aware of the offloading decision made by the other. Now if node *n*3 becomes overloaded due to the processes received fro both nodes *n*1 and *n*2 , then it may again start transferring its processes to other nodes. This entire cycle may be repeated again and again, resulting in an unstable state. This is certainly not desirable for a good scheduling algorithm.

Stability

- **process migration** is a specialized form of process management whereby processes are moved from one computing environment to another.

A scheduling algorithm is said to be unstable if it can enter a state in which all the nodes of the system are spending all of their time migrating processes without accomplishing any useful work in an attempt to properly schedule the processes for better performance.

Unaffected by system failures

- The algorithm should not be disabled by system failures such as node or link crash and it should have decentralized decision making capability

Scheduling in distributed systems

Local scheduling discipline: for single cpu

Global Scheduling discipline :for entire distributed system

The **transfer policy** - to determine when it is necessary to transfer a process from one node to another.

The **selection policy** - to determine which process of the selected node should be transferred.

The **location policy** - to determine to which node a selected process should be transferred.

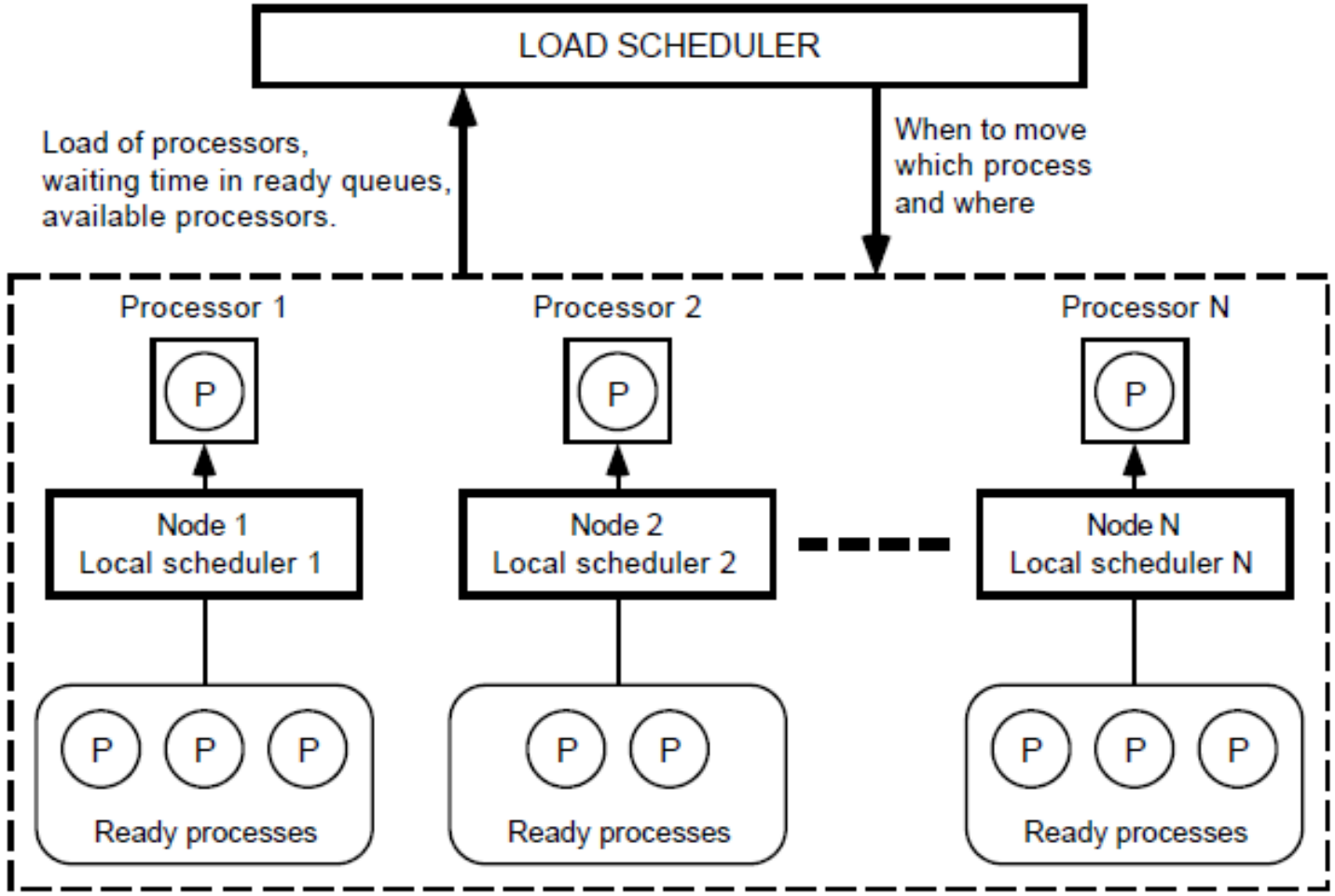


Fig. 1.2 Local and global scheduling in a distributed system.

Task Assignment Approach

- Each process is divided into multiple tasks. These tasks are scheduled to suitable processor to improve performance. This is not a widely used approach because:
- It requires characteristics of all the processes to be known in advance.
- This approach does not take into consideration the dynamically changing state of the system.
- In this approach, a process is considered to be composed of multiple tasks and the goal is to find an optimal assignment policy for the tasks of an individual process. The following are typical assumptions for the task assignment approach:

Assumptions For Task Assignment Approach

- A process has already been split into pieces called tasks.
- The amount of computation required by each task and the speed of each processor are known.
- The cost of processing each task on every node of the system is known.
- The Interprocess Communication (IPC) costs between every pair of tasks is known.
- Other constraints, such as resource requirements of the tasks and the available resources at each node, precedence relationships among the tasks, and so on, are also known.

Points to be considered while task assignment

- Low IPC Cost
- Less Turnaround time
- High degree of parallelism
- Efficient resource utilization

A tabular Example of Task Assignment Approach

Inter-task Communication Cost (c_{ij})				
	t1	t2	T3	t4
t1	0	35	3	8
t2	35	0	6	4
t3	3	6	0	23
t4	8	4	23	0

An infinite cost for a particular task against a particular node indicate that the task cannot be executed on that node due to the task's requirement of specific resources that are not available on that node.

Execution Costs(x_{ab})			
Task	Nodes		
	n1	n2	n3
t1	31	4	14
t2	1	5	6
t3	2	4	∞
t4	3	28	10

Arbitrary Assignment	
Task	Node
t1	n2
t2	n2
t3	n2
t4	n3

Optimal Assignment	
Task	Node
t1	n2
t2	n2
t3	n1
t4	n1

Cost of arbitrary assignment (Total execution and communication cost) = $x_{12} + x_{22} + x_{32} + x_{43} + c_{14} + c_{24} + c_{34} = 4 + 5 + 4+10 + 8 + 4 + 23 = 58$

Cost of optimal assignment (total execution and communication cost) = $x_{12} + x_{22} + x_{31} + x_{41} + c_{13} + c_{23} + c_{14} + c_{24} = 4 + 5 + 2 + 3 + 3 + 6 + 8 + 4 = 38$

Inter-task Communication Cost (c_{ij})				
	t1	t2	T3	t4
t1	0	35	3	8
t2	35	0	6	4
t3	3	6	0	23
t4	8	4	23	0

Execution Costs(x_{ab})			
Task	Nodes		
	n1	n2	n3
t1	31	4	14
t2	1	5	6
t3	2	4	∞
t4	3	28	10

Arbitrary Assignment	
Task	**Node**
t1	**n2**
t2	**n2**
t3	**n2**
t4	**n3**

Cost of arbitrary assignment (Total execution and communication cost) =

$$x_{12} + x_{22} + x_{32} + x_{43} + c_{14} + c_{24} + c_{34} = 4 + 5 + 4 + 10 + 8 + 4 + 23 = 58$$

Optimal Assignment	
Task	**Node**
t1	**n2**
t2	**n2**
t3	**n1**
t4	**n1**

Cost of optimal assignment (total execution and communication cost) =

$x_{12} + x_{22} + x_{31} + x_{41} + c_{13} + c_{23} + c_{14} + c_{24} = 4 + 5 + 2 + 3 + 3 + 6 + 8 + 4 = 38$

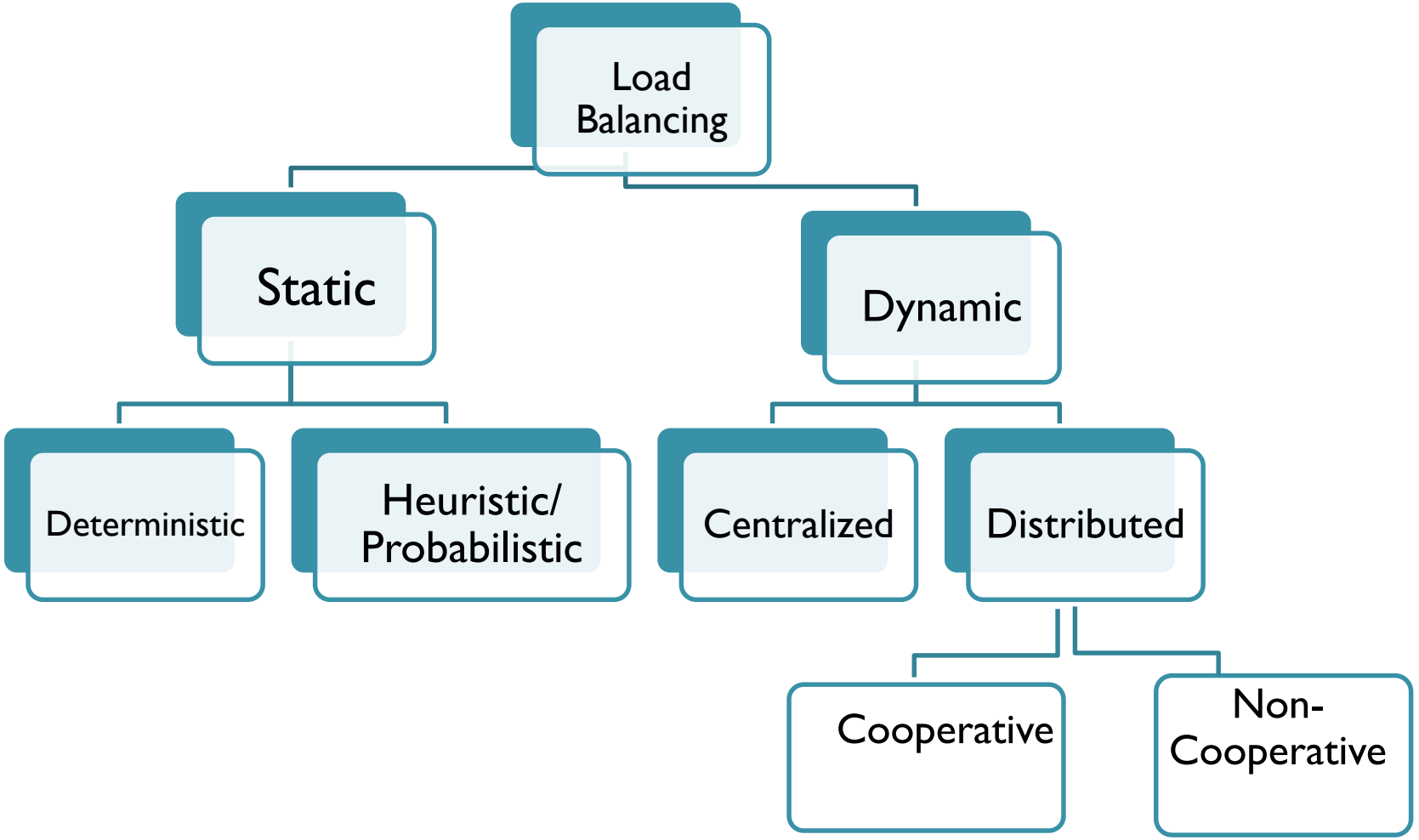
Load Balancing Approach

- In distributed computing thousand of processors can be connected either by wide area network or across a large number of systems which consists of cheap and easily available autonomous systems like workstations or PCs.
- The distribution of loads to the processing elements is simply called the load balancing.
- The goal of the load balancing algorithms is to maintain the load to each processing element such that all the processing elements become neither overloaded nor idle that means each processing element ideally has equal load at any moment of time during execution to obtain the maximum performance (minimum execution time) of the system.

Benefits of Load Balancing

1. Load balancing improves the performance of each node and hence the overall system performance.
2. It reduces the job idle time.
3. Here, small jobs do not suffer from long starvation.
4. It makes maximum utilization of resources.
5. Response time becomes shorter.
6. It gives higher throughput.
7. It gives higher reliability.
8. It has low cost but high gain.
9. It gives extensibility and incremental growth.

Taxonomy Of Load Balancing



Static Load Balancing

- In static algorithm the processes are assigned to the processors at the compile time according to the performance of the nodes.
- Once the processes are assigned, no change or reassignment is possible at the run time.
- Number of jobs in each node is fixed in static load balancing algorithm. Static algorithms do not collect any information about the nodes .

Static Load Balancing

- The assignment job is done to the processing nodes on the basis of the following factors:

1. Incoming time.
2. Extent of resource needed.
3. Mean execution time.
4. Inter-process communications.

Sub Classes of SLB

- The static load balancing algorithms can be divided into two sub classes:
  - Optimal static load balancing (Deterministic)
  - Sub optimal static load balancing (Probabilistic)

Optimal Static Load Balancing Algorithm:

- If all the information and resources related to a system are known optimal static load balancing can be done such as the list of processes, computing requirements, file requirements and communication requirements

Sub Optimal Static Load Balancing Algorithm

- Sub-optimal load balancing algorithm will be mandatory for some applications when optimal solution is not found.
- In case the load is unpredictable or variable from minute to minute or hour to hour.

Dynamic Load Balancing

- In dynamic load balancing algorithm assignment of jobs is done at the runtime.
- In DLB jobs are reassigned at the runtime depending upon the situation that is the load will be transferred from heavily loaded nodes to the lightly loaded nodes.
- In dynamic load balancing no decision is taken until the process gets execution.
- This strategy collects the information about the system state and about the job information.
- As more information is collected by an algorithm in a short time, potentially the algorithm can make better decision.

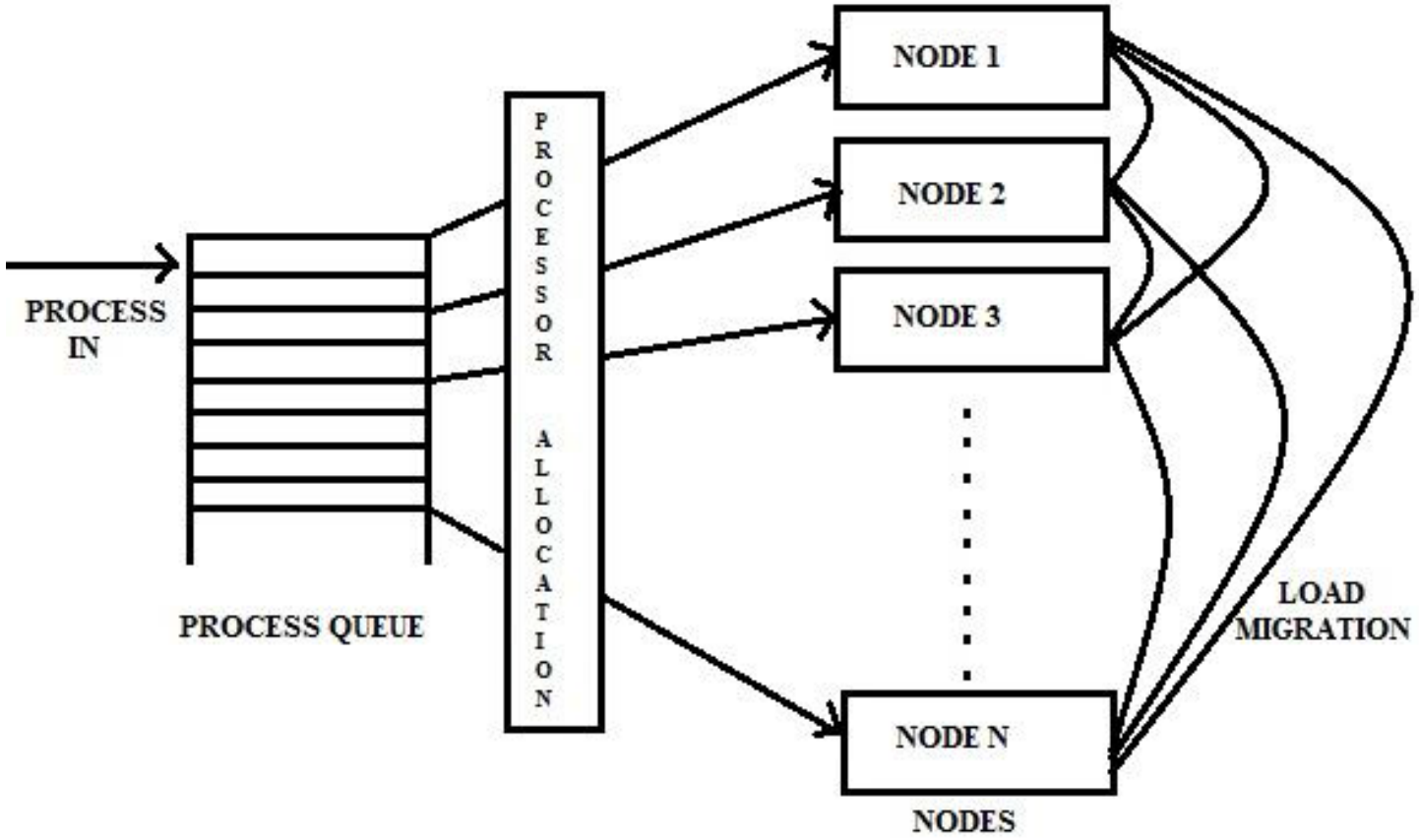


Figure: Job Migration in Dynamic Load Balancing Strategy

Centralized Vs Distributed

- A **centralized** dynamic scheduling algorithm means that the scheduling decision is carried out at one single node called the centralized node. This approach is efficient since all information is available at a single node.
- Drawback for **centralized** approach is it leads to a bottleneck as number of requests increase .
- In a **distributed** algorithm the task of processor assignment is physically distributed among various nodes .
- In **distributed algorithm** scheduling decisions are made on individual nodes.

Cooperative Vs non-cooperative

- **Cooperative Algorithm** distributed entities cooperate with each other to make scheduling decisions
- **Non-Cooperative algorithm** the individual entities make independent scheduling decisions and hence they involve minor overheads
- Cooperative algorithms are more complex than non-cooperative ones
- Non-cooperative algorithms may not be stable

Issues in designing in load balancing algorithms

- Deciding policies for:
  - **Load estimation:** determines how to estimate the workload of a node in a distributed system.
  - **Process transfer:** decides whether the process can be executed locally or there is a need for remote execution.
  - **Static information exchange:** determines how the system load information can be exchanged among the nodes.
  - **Location Policy:** determines the selection of a destination node during process migration
  - **Priority assignment:** determines the priority of execution of a set of local and remote processes on a particular node

Policies for Load estimation

- Parameters:
- number of processes running on the machine
- Resource demand of these processes
- Instruction mix of processes
- Architecture & speed of the node
- Sum of remaining service time of all the processes.

Policies for Process transfer

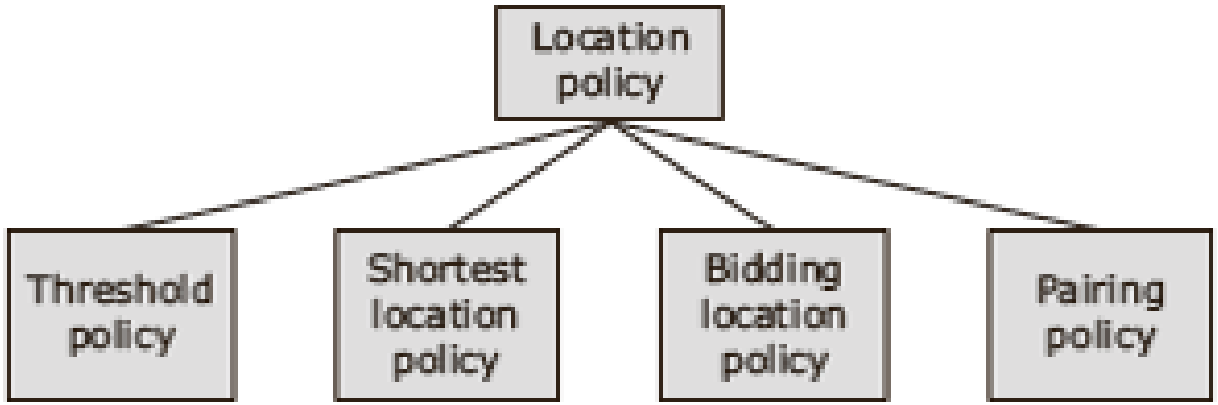
- Load balancing strategy involves transferring some processes from heavily loaded nodes to lightly nodes
- So there is a need to decide a policy which indicates whether a node is heavily or lightly loaded, called threshold policy.
- This threshold is a limiting value which decides whether a new process, ready for execution or transferred to a lightly loaded node
- Threshold policy
  - Static
  - Dynamic

Threshold Policies

- Threshold policy
  - **Static:** each node has a predefined threshold value depending on its processing capability, which does not vary with load. The advantage of this method is that there is no need for any exchange of state information to decide the threshold value
  - **Dynamic:** the threshold value is calculated as an average workload of all nodes. This policy gives a realistic value but involves the overhead of state information exchange among nodes to determine the threshold value.

Location policies

- Once the threshold transfer policy decides to transfer a process from a node, the next step is to use a location policy to select the destination node where the process can be executed.



Threshold policy

- In the threshold policy, the destination node is selected at random and a check is made to verify whether the remote process transfer would load that node, If not, the process transfer is carried out; else another node is selected at random and probed. This process continues till a suitable destination node is found or the number of nodes probed exceeds a probe limit(defined by the system)

Shortest Location policy

- Nodes are chosen at random and each of these nodes is polled to check for load.
- The node with the lowest load value is selected as the destination node.
- Once selected the destination node has to execute the process irrespective of its state at the time the process arrives.
- In case non of the polled nodes can accept the process it will be executed at the source node itself

Bidding Location Policy

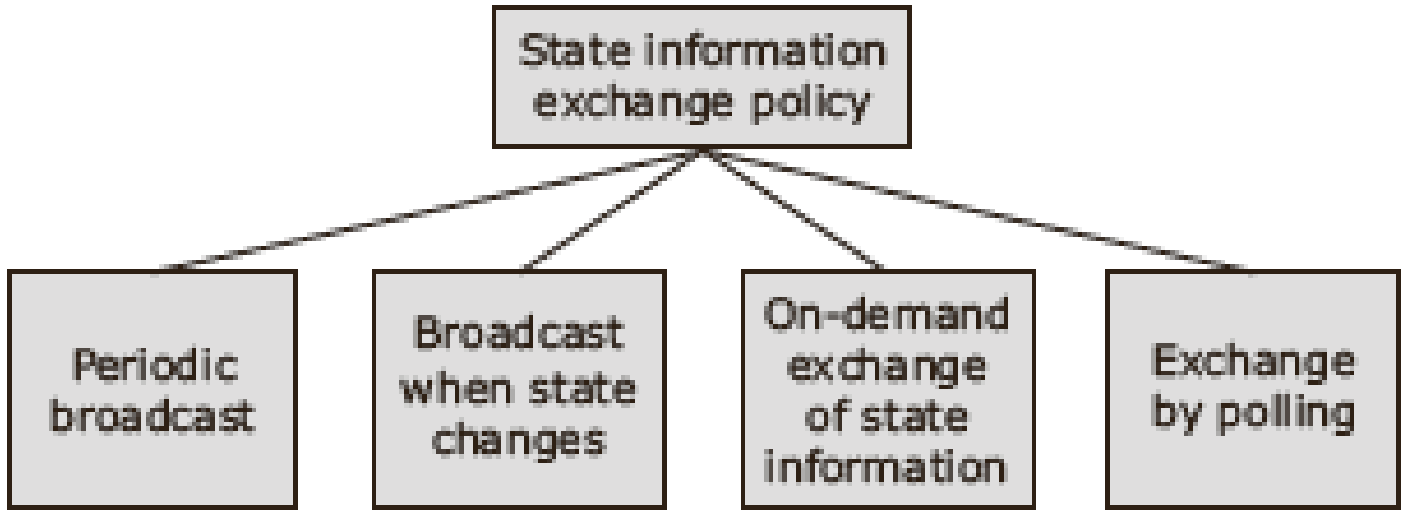
- This policy transforms the system into a market scenario with buyers and sellers of services.
- Each node is assigned two roles, namely the **manager and the contractor**.
- The manager is an under loaded node having a process which needs a location and the contractor is a node which can accept remote processes.
- The manager broadcasts a request for a bid message to all nodes and the contractors send their bids to the manager node
- The bid contains information about processing power, and memory size
- The manager chooses the best bid which is the cheapest, and fastest then the process transferred to the winning contractor node.
- If the contractor won too many bids at a time from many managers this may become overload, so when the best bid is selected a message sent to the owner of the bid which can weather accept or reject the process

Pairing Policy

- The policies discussed earlier focus on load balancing across the systems, while the pairing policy focuses on load balancing between a pair of nodes
- Two nodes which have a large difference of load balancing between a pair of nodes are paired together temporarily
- The load balancing is carried out between the nodes belonging to the same pair by migrating processes from the heavily loaded node to the lightly loaded node
- Several pairs of nodes can exist in the system simultaneously

State information exchange

- Dynamic policies require frequent exchange of state information among the nodes of a system
- Decision based on state information



Periodic Broadcast

- In this policy, each node broadcasts its state information at time interval t .
- This method generates heavy traffic and there may be unnecessary messages transmitted in case where the node state has not changed in time t .
- Scalability is also an issue, since the number of messages generated for state information exchanges will be too large for a network with many nodes.

Broadcast when state changes

- **Broadcast when state changes:** In this method, a node broadcasts its state information only when a node's state changes (when a process arrives at a node or a process departs from a node).

On-demand exchange of state information

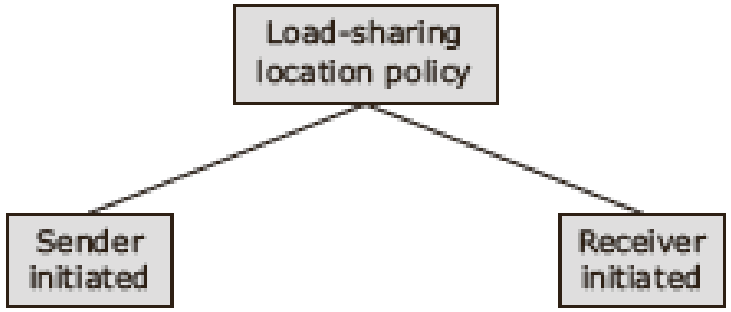
- **On-demand exchange of state information:** The method of on-demand exchange of state information is based on the fact that a node needs to know the state of other nodes, only when it is either **under-loaded or overloaded.**
  - A node will broadcast $Stateinformationrequest$ message then the nodes send their current state to the requesting node

Exchange by polling

- Broadcasting occurs only when a node needs cooperation from another node for load balancing
- It search for a suitable partner by polling all nodes one by one and exchanging state information
- The polling process is stopped when a suitable pair is found or a predefined poll limit is reached

Load Sharing Approach

- Load balancing approaches attempt to equalize the workload on all the nodes of a system by gathering state information
- Load sharing algorithms do not attempt to balance the average workload on all nodes, they only ensure that no node is idle or heavily loaded
- Policies for load sharing approach are the same as load balancing polices, they include load estimation policy, process transfer policy, location policy and state information exchange, they differ in location policy



Location sharing policies

Location policy for load sharing approach

- **Sender Initiated algorithm** uses sender of the process to decide where to send the process
  - The heavily loaded node search for lightly loaded nodes where the process can be transferred
  - When a load on a node increases beyond the threshold , it probes nodes to find a lightly loaded node .
  - A node can receive a process only if its transfer will not increase its load beyond the threshold .
  - If a suitable node is not found the process will be executed on the same node.

Location policy for load sharing approach cont.

- **Receiver initiated location policy**
  - In this policy lightly loaded nodes search for heavily loaded nodes from which processes can be accepted for execution .
  - When the load on a node falls below a threshold value, it broadcasts a probe message to all nodes or probes nodes one by one to search for a heavily loaded node
  - A node can transfer one of its processes if such a transfer does not reduce its load below normal threshold

Process Management in a Distributed Environment

Process Management in a Distributed Environment

- Main goal of process management in DS is to make best possible use of existing resources by providing mechanism and polices for sharing them among processors

- **This achieve by providing :**
  - **Process allocation** : decide which processor should be assign to which process in any instance of time for better utilization of resources .
  - **Process migration** : move process to new node for better utilization of resources
  - **Thread facilities** : provide mechanism for parallelism for better utilization of processor capabilities.

Types of process migration

Process Management in a Distributed Environment

Main Functions of distributed process management

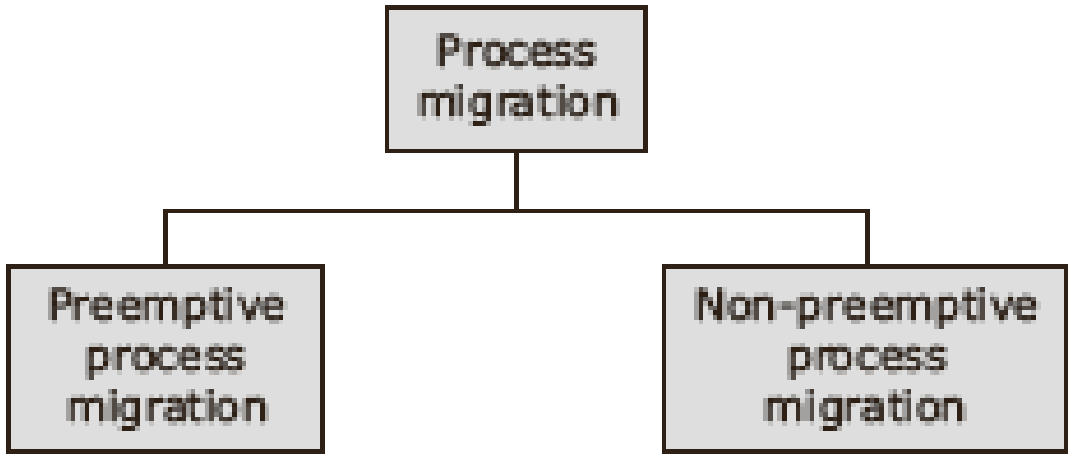
- **Process allocation :**decide which process should assign to which process in any instance of time
- **Process Migration:** Change of location and execution of a process from current processor to the destination processor for better utilization of resources and balancing load in distributed system.

Types of process migration

Process Management in a Distributed Environment

Process Migration classified into :

- **Non-preemptive : process is migrate before start execution in source node .**
- **Preemptive : process is migrate during its execution .**



Types of process migration

Desirable features of a good process migration mechanism

- Transparency:
- Minimal interference:
  - To the progress of process and system
  - Freezing time : time period for which the execution of the process is stopped for transferring its info to destination node
- Minimal residual dependencies:
  - Migrated process should not depend on its previous node

Desirable features of a good process migration mechanism Cont.

- Efficiency:
  - Issues
    - Time required to migrate process
    - Cost of locating the object
    - Cost of supporting remote execution once the process is migrated
- Robustness :
  - Failure of any other node should not affect the accessibility or execution of the process
- Ability to communicate between co processes of the job:
  - Communication directly possible irrespective of location

3.7 Process migration

3.7 Process Migration

- In computing, **process migration** is a specialized form of **process** management whereby **processes** are moved from one computing environment to another.

Steps involved in process migration

- Freezing the process on its source and restarting it on its destination
- Transferring the process's address space (program code – data –stack program) from its source to destination
- Forwarding messages meant for the migrant process
- Handling communication between cooperating process

Mechanism

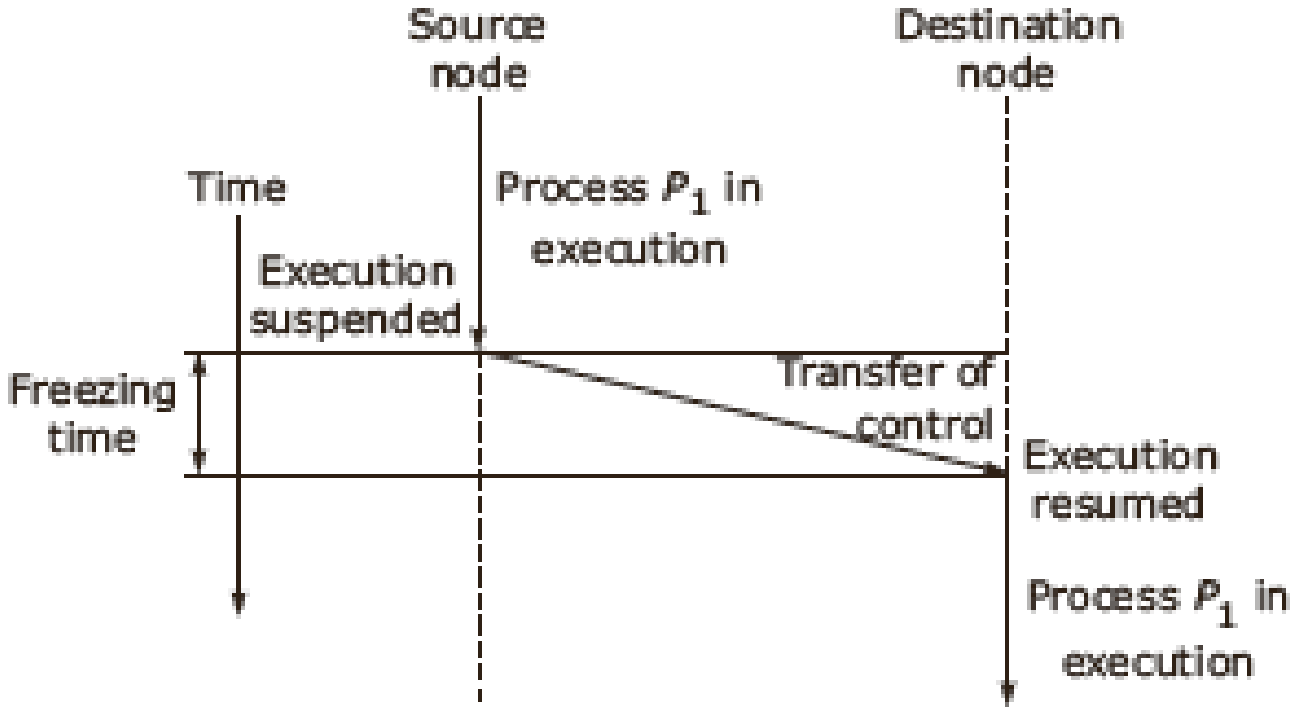


Figure 6-15 Process migration mechanism

Process migration mechanism

Freezing process on source node

- Take snapshot of process's state on its source node
  - Reinstate the snapshot on the destination node
  - Issues
    - Immediate and delayed blocking of the process
    - Fast and slow I/O operations
      - After process is blocked wait for completion of fast I/O operations , then process is frozen
      - Slow I/O performed after migration
    - Information about open files
      - Use of Links to remote files
      - Use of local files as far as possible
    - Reinstating the process on its destination node
    - Constructing a new copy in the destination

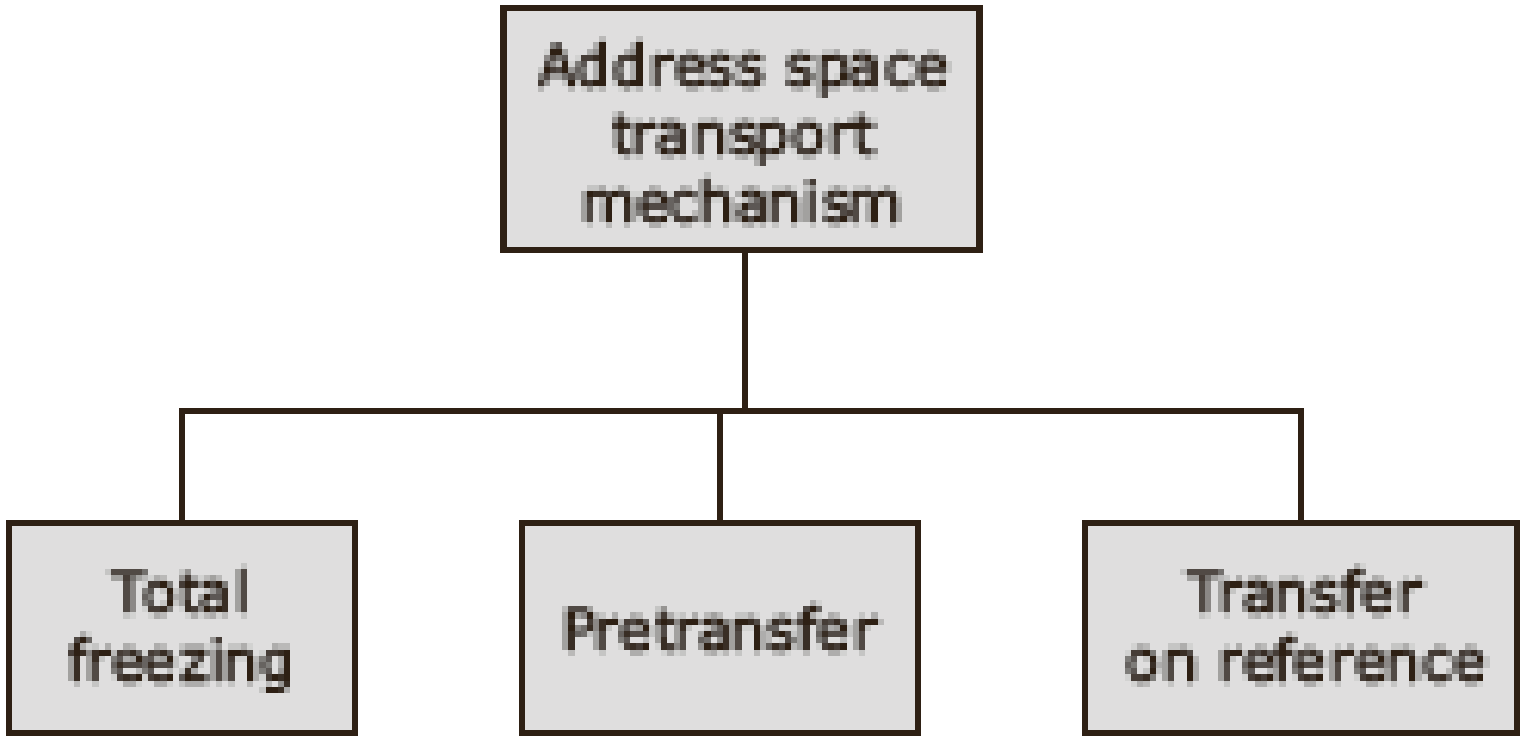
Address space transport mechanisms- I

Information to be transferred

- **Process's state**
  - Execution status
  - Scheduling info
  - Info about RAM
  - I/O states
  - Objects for which the process has access rights
  - Info about files opened etc.
- **Process's address space**

  (code, data & stack)
  - Higher in size than process's state info
  - Can be transferred after migration, before or after process starts executing
  - **Address space transfer mechanisms**
    - Total freezing
    - Pre transferring
    - Transfer on reference

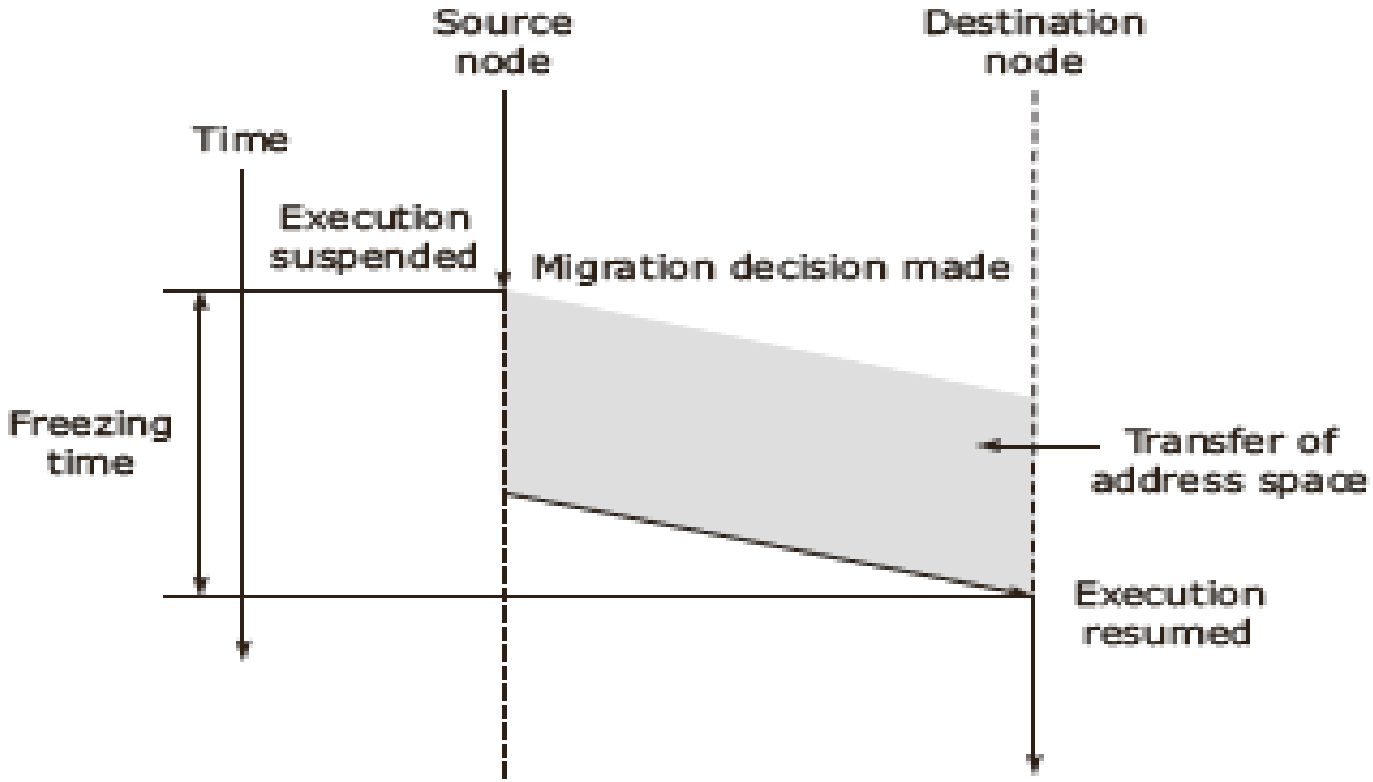
Address space transport mechanisms-I



Address space transport mechanism

Address space transport mechanisms-2

- Total Freezing
  - Process's execution is stopped while transferring the address space
  - Disadvantage that process may be suspended for a long time



Total freezing mechanism

Address space transport mechanisms-3

- Pre-transfering or Pre-copying
  - Address space transferred while process is running on the source node

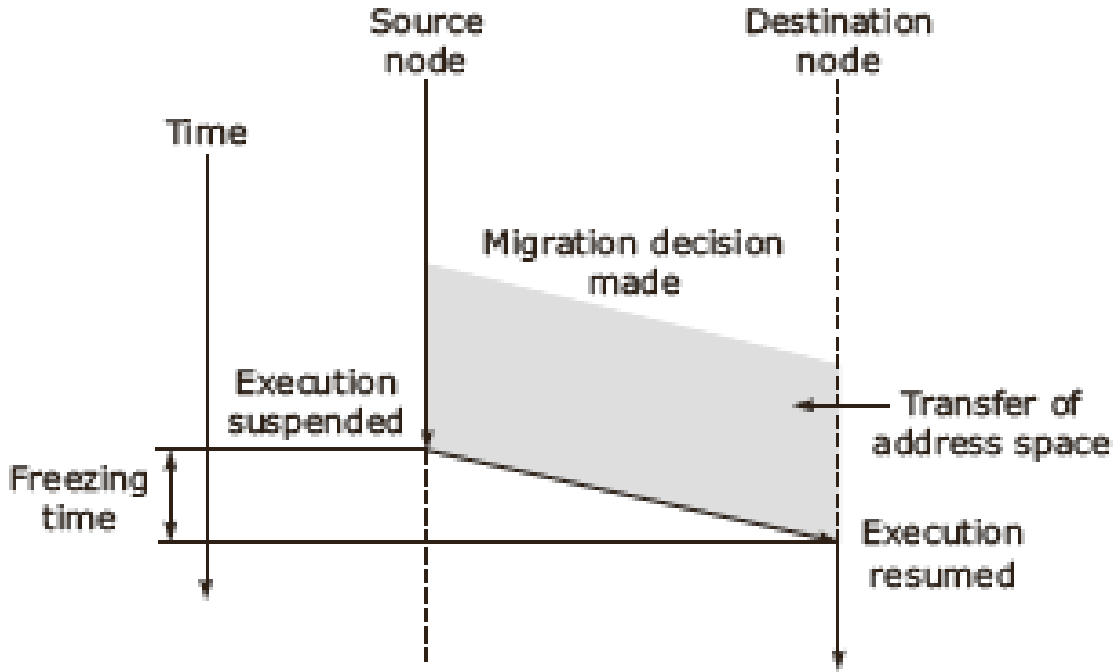


Figure 6-18 Pretransfer

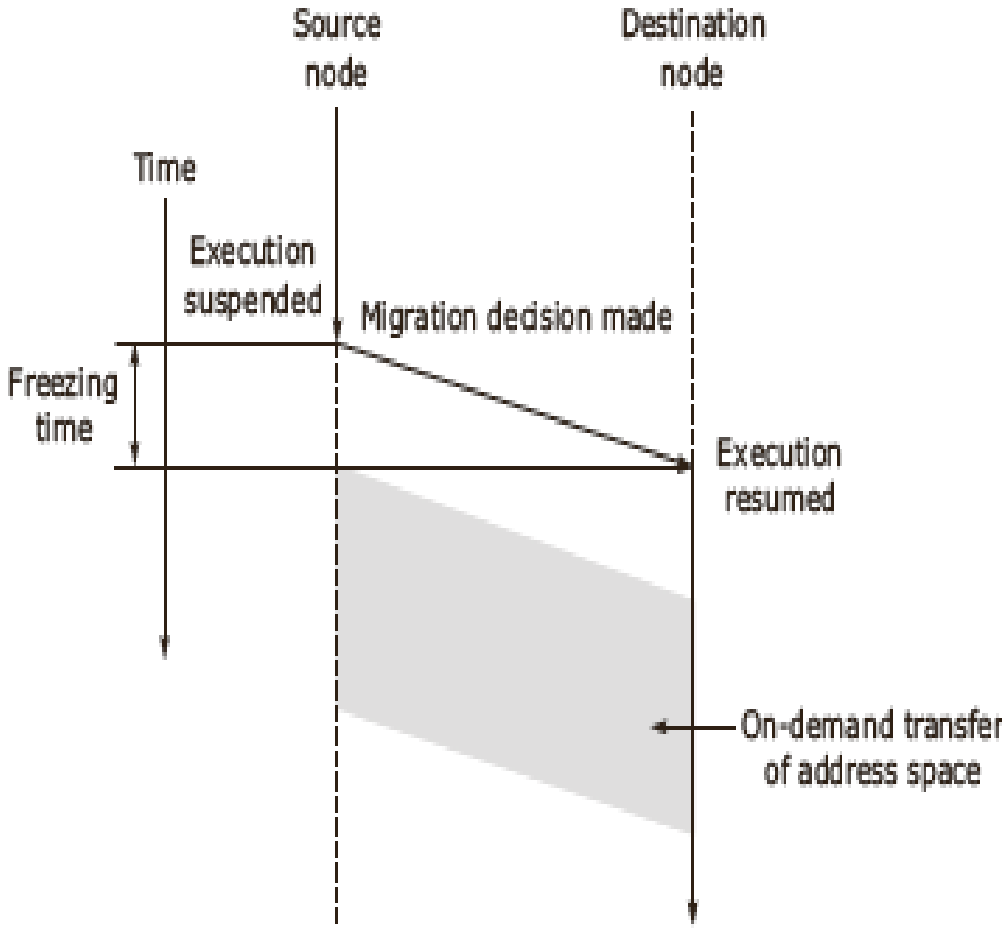
Pretransfer mechanism

Address space transport mechanisms-3

- Pretransfering or Precopying
  - Address space transferred while process is running on the source node
  - After decision for migration is made process continues to execute on source node until address space is has been transferred
  - Initially entire address space is transferred followed by repeated transfers of pages modified during previous transfer so on until no reduction in number of pages is achieved
  - The remaining pages are transferred after the process is frozen for transferring its state info
  - Freezing time is reduced
  - Migration time may increase due to possibility of redundant transfer of same pages as they become dirty while pretransfer is being done

Address space transport mechanisms-4

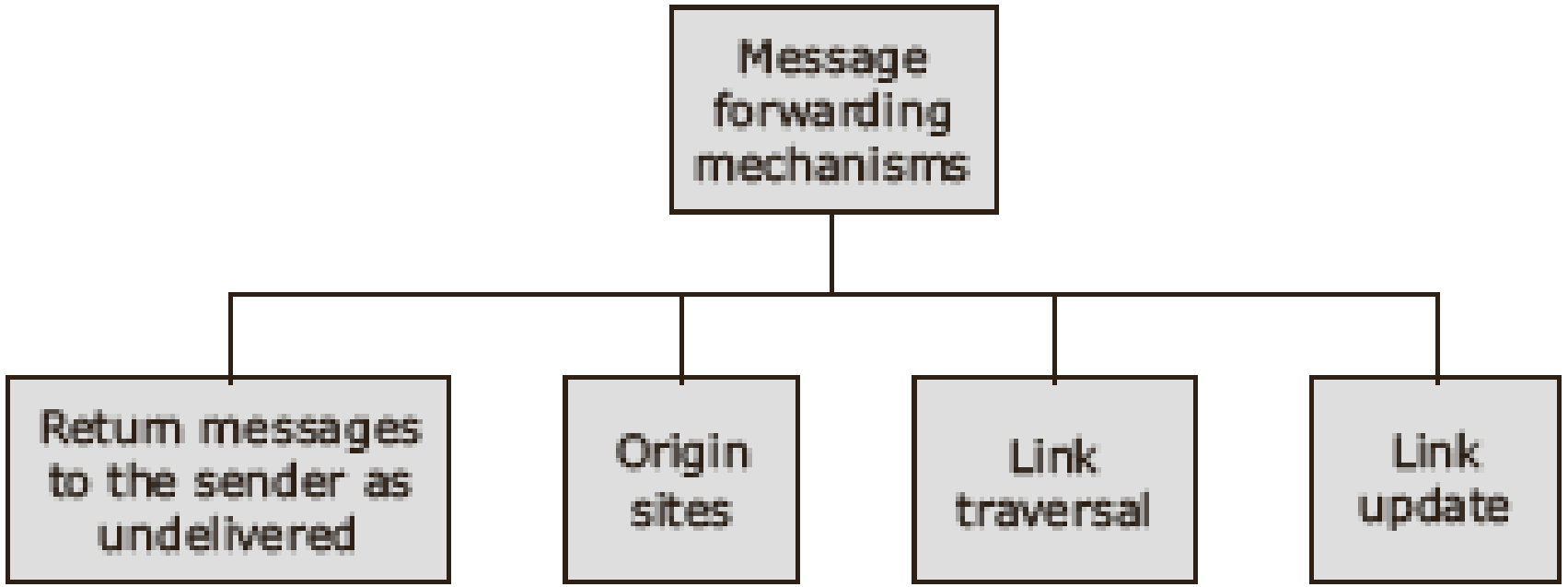
- Transfer on Reference
  - Process executes on destination
  - Address space is left behind in source node
  - Desired blocks are copied from remote locations as and when required
  - Failure of source node results in failure of process



Transfer on Reference mechanism

Messages Forwarding

- Track and forward messages which have arrived on source node after process migration



Messages Forwarding

- Messages
  - Messages received at source node after the process stopped on its source node and not started on the destination node
  - Messages received at source node after execution
  - Messages of process started at destination node

Message forwarding Mechanisms

- Return message to sender as undeliverable
  - Message type 1 and 2are returned to sender or dropped
  - Sender retries after locating the new node (using locate operation)
  - Type 3 message directly sent to new node
- Origin site mechanism
  - All messages are sent to origin site
  - Origin site forwards the messages
  - If origin site fails forwarding mechanism fails
  - Continuous load on the origin site

Messages Forwarding Message forwarding Mechanisms

- Link traversal mechanism
  - Message queue is generated at origin
  - Message Forwarded to destination node
  - After process is migrated link is left on the previous node
  - Process address has two parts process id, last known location of destination node

Advantages of process migration

- Reduce average response time of heavily loaded nodes
- Speed up of individual jobs
- Better utilization of resources
- Improve reliability of critical processes

Threads

Process v/s threads

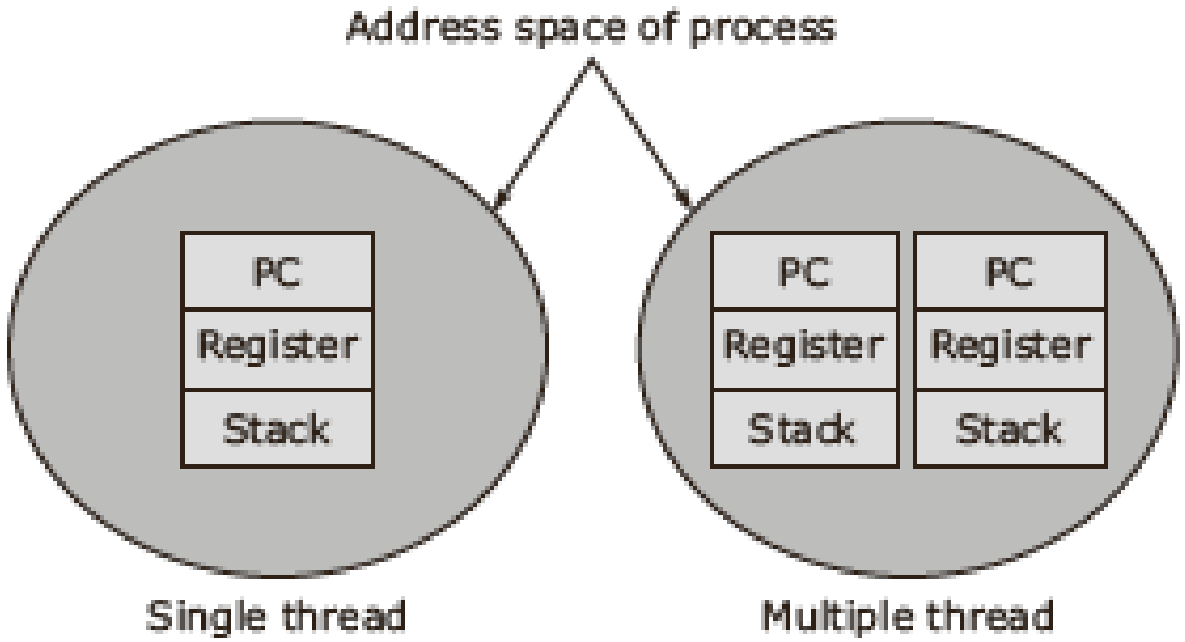
- Programs are divided to process , which that can be independently executed .
- Process can block itself , while waiting for some other operation to be complete , and program execution can slow down .
- We cannot use multiple processes , since processes don’t share address space. Instead if this process is divided into threads, one of the threads can go to sleep , while other threads may continue execution. thus , system throughput is increased and this in turn improve system performance .

Motivation for using threads

- Overhead involved is less
- Switching between the easy and cheaper as they belobg to same address space
- High degree of parallelism.
- More resource sharing.

Process v/s threads

- Analogy:
  - Thread is to a process as process is to a machine



Process address space

Comparison

Criteria	Process	Thread
Control block	Process Control Block (PCB): program counter, stack, and register states; open files, child processes, semaphores, and timers	Thread Control Block (TCB): program counter, stack, and register states
Address space	Separate for different processes, provides protection among processes	Share process address space, no protection between threads belonging to the same process
Creation overhead	Large	Small
Context switching time	Large	Small
Objective of creation	Resource utilization, to be competitive	Use pipeline concept, to be cooperative

Comparison of processes and threads

Thread models

- Dispatcher worker model
- Team model
- Pipeline model

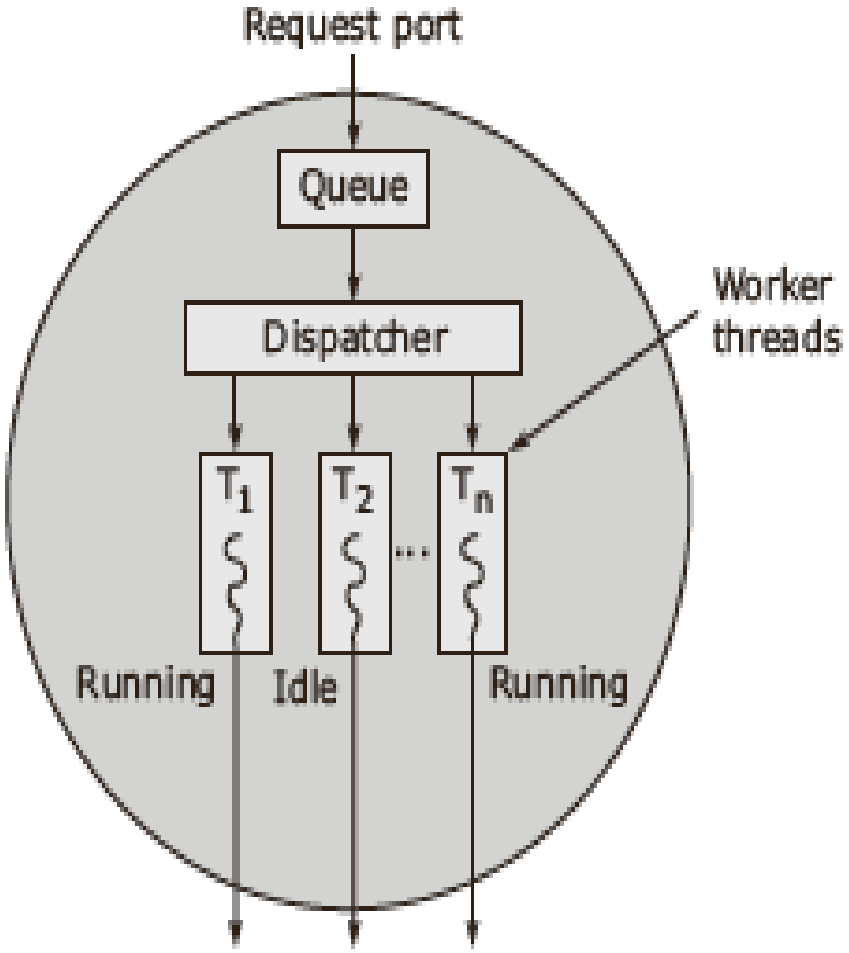
Thread: Dispatcher worker model

Typical example of this model is server process such as file server that :

1. Accept request from client .
2. Check for access permission.
3. Accordingly services the request.

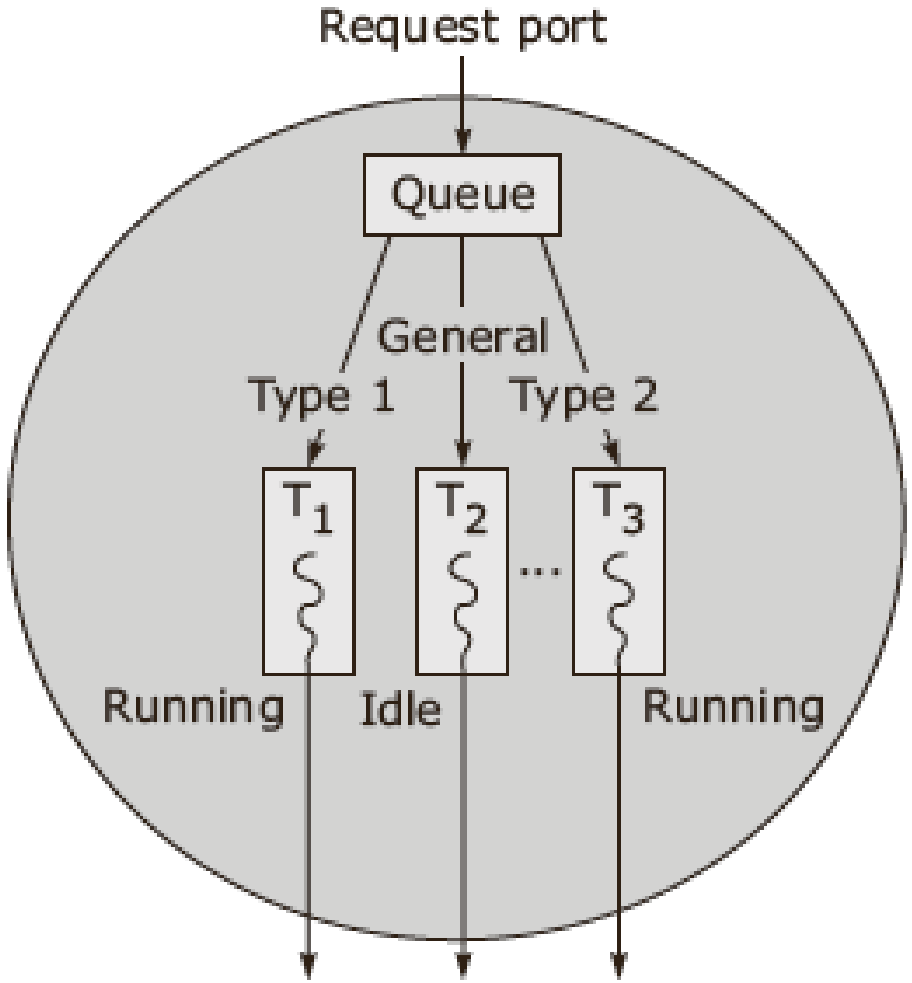
Thread: Dispatcher worker model

- Assume that Single process is divided into one dispatcher and multiple worker
- Dispatcher thread accept incoming request from client request queue.
- Examine request and choose idle worker thread to handle request.
- Thus worker thread change its state to running and dispatcher change state to ready .
- Since each worker thread processes different client request , multiple request can be processed in parallel.



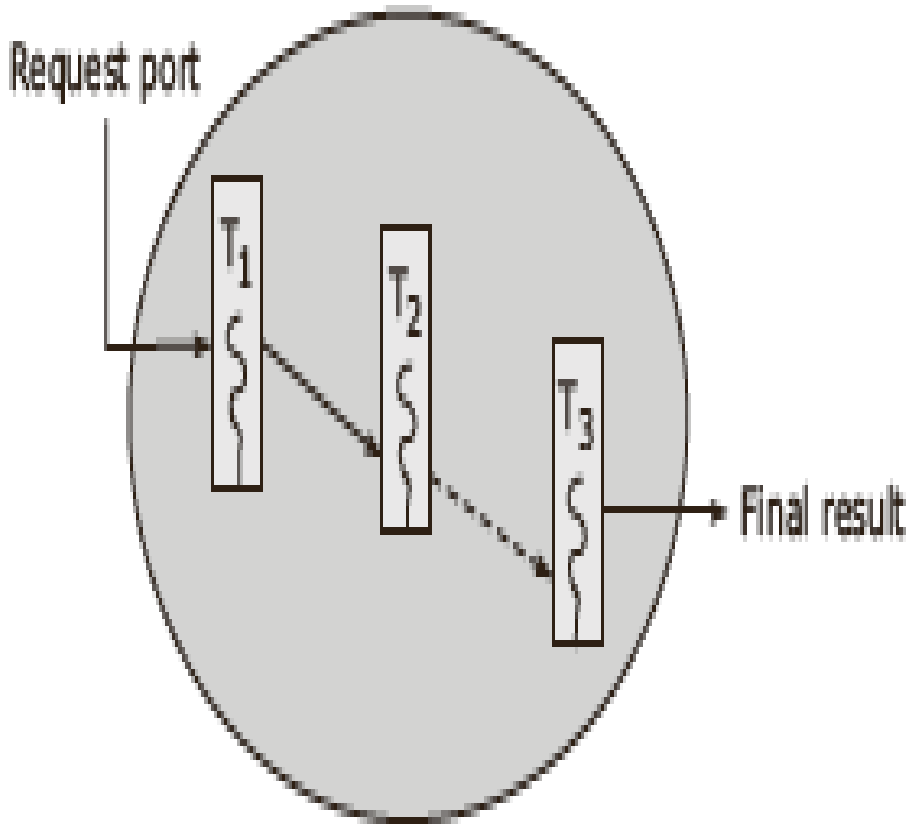
Thread: Team model

- All threads are treated equal , such that each one handle request on its own .
- In case threads are capable of performing specific distinct function , a queue can be maintained .
- When thread change state from running to idle , it take new request from the job queue and stars execution .



Thread: Pipeline model

- Used pipeline concept that used in CPU instructions executions.
- The tasks assigned to the threads are completed and the result generated by first thread are passed to next thread .
- It take this as input and start running .
- Data pass across multiple threads with each one processing it partly and the last thread giving the final result .



Design issues in threads

- Thread semantics
  - Thread creation, termination
- Thread synchronization
- Thread scheduling

Design issues in threads :Thread semantics

- The first step before using threads is **thread creation** that can be
  - Static .
  -Dynamic .

  Static : number of threads to be created is fixed when program is written or when it is complied , and memory space is allocate to each thread.

  Dynamic : threads are created as and when it is needed during the process life cycle .and they exit when task is completed .Here the stack size for the threads is specified as parameter to the system call for thread creation.

Design issues in threads : Thread semantics

- Threads termination : threads follow the same steps for termination as processes either :

  - ***EXIT*** call command : thread destroys itself on task completion by making an *EXIT* call .

  -***KILL*** call command :or thread is killed from outside using *KILL* command with thread id as parameter.

Design issues in threads

- Thread semantics
  - Thread creation, termination
- **Thread synchronization**
- Thread scheduling

Thread synchronization

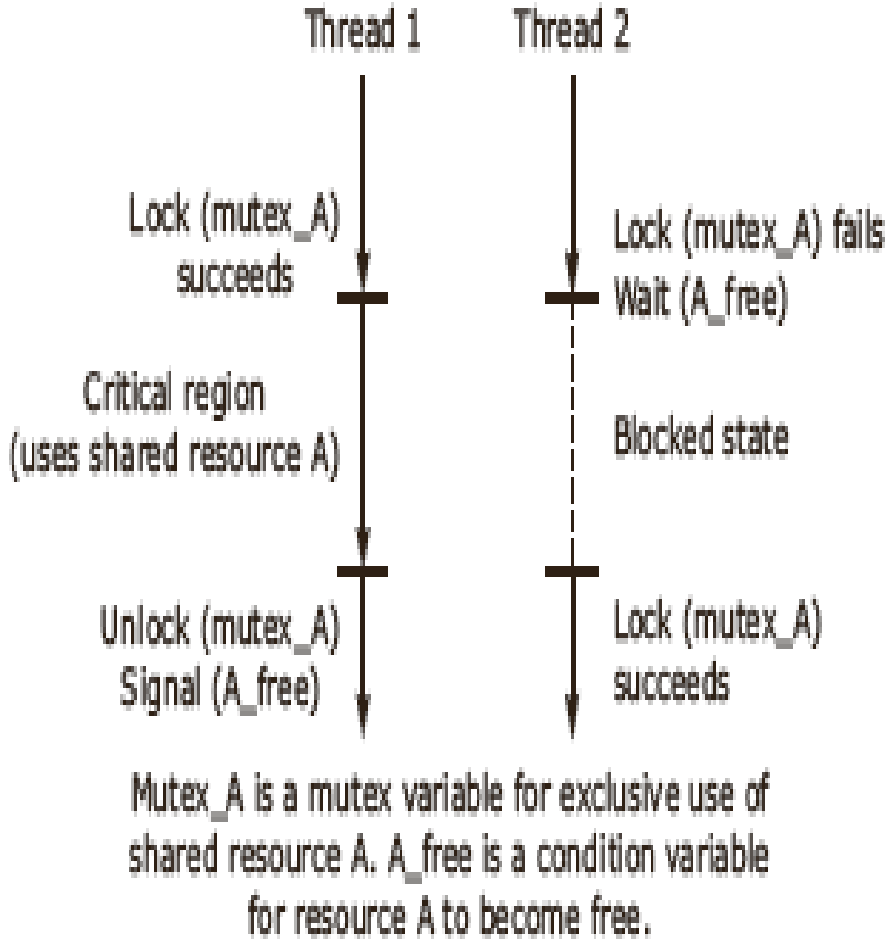
- Since threads belong to process share the same address space , thread synchronization are required to ensure that multiple threads don’t access the same data simultaneously.
- For example , if two threads want to double the same global variable , it is best done one after another.

Thread synchronization : example

- One thread should exclusive access to shared variable , double it , and pass control to the other thread.
- To provide exclusive access to shared variables , we define critical region .It mean that only one thread can execute in critical region at any instance of time.
- Critical region is implemented using mutex variable , which is binary semaphore: *locked* and *unlocked*

Thread synchronization

- Execution in Critical region
  - Use binary semaphore
- The lock operation attempts to lock the mutex. It successes if unlocked m and mutex become locked in single atomic action.
- If two threads try to lock the same mutex at the same time , only one successes , while the other thread is blocked .



Implementation of critical region

Design issues in threads

- Thread semantics
  - Thread creation, termination
- Thread synchronization
- **Thread scheduling**

Threads scheduling

- Another important issue in designing threads package is to decide an appropriate scheduling algorithm.
- The Threads packages provide application programmer with calls to specify scheduling policy to be used for application execution .

Threads scheduling policies/algorithms

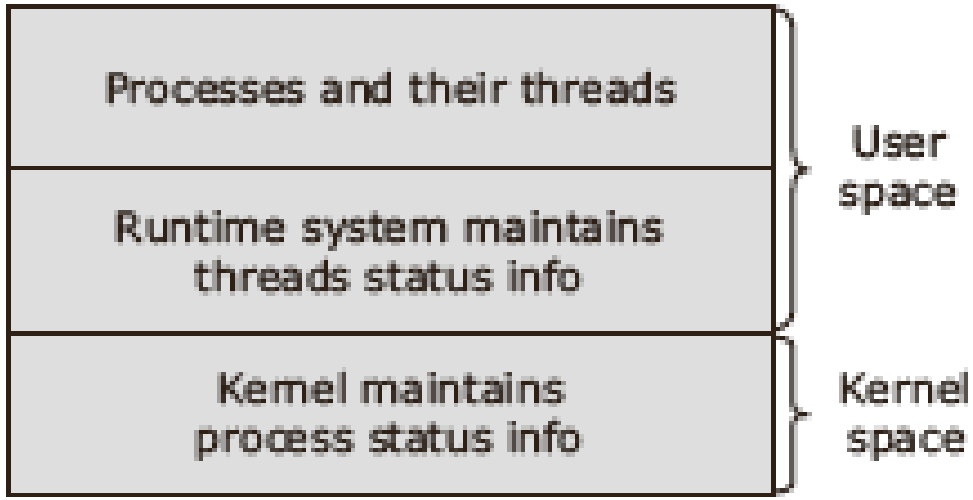
- Priority assignment facility
- Choice of dynamic variation of quantum size
- Handoff scheduling scheme
- Affinity scheduling scheme
- Signals used for providing interrupts and exceptions

Implementing thread package

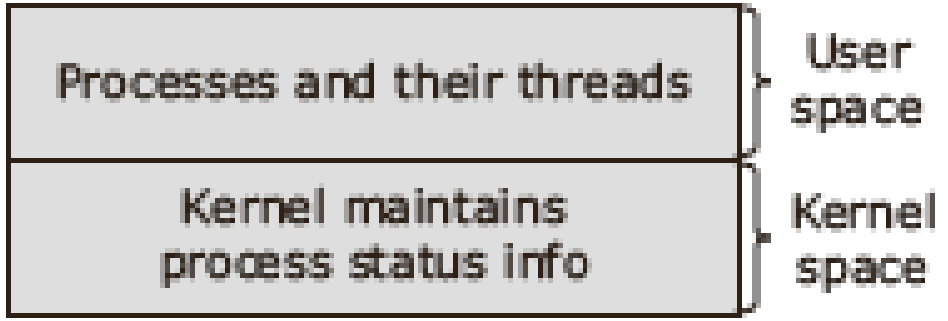
- Typically any OS divides memory into users and kernel space to store programs and data .
- **Thread package can be implemented either in**
  - user space
  - kernel space .

Implementing thread package

- **User level approach**



Kernel level approach



Comparison of thread implementation-I

User- level vs. kernel-level thread implementation

Criteria	User-level approach	Kernel-level approach
Thread package implementation	Can be implemented even on the OS which does not support threads.	Can be implemented only in the OS which supports threads because it needs to be integrated into the kernel design.
Flexibility to use customized scheduling algorithms	Users can design the algorithms which suit the application because of the use of two-level scheduling.	Users can only specify priorities for selecting a new thread because only one-level scheduling is used.
Context switching	Faster because it is managed by the runtime system.	Slower because the trap has to be made to the kernel.
Scalability	Scalable since the status information table is maintained by the runtime system.	Poorly scalable because this status information table is maintained by the kernel.

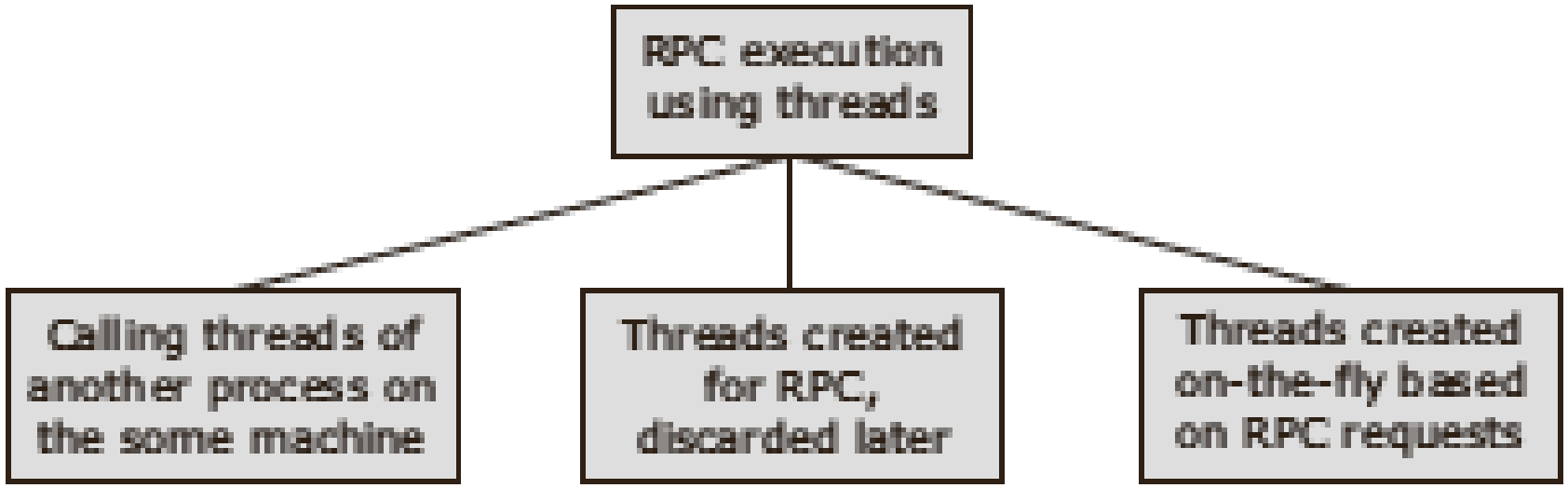
Comparison of thread implementation-2

Blocking system call implementation	If a thread makes a blocking system call, all threads of the process will be trapped. The kernel schedules another process to run, the objective of the thread will be lost. The solution is to use a jacket routine; extra code before a blocking system call. It checks if the call causes a trap to the kernel. Call is allowed to be made if it is safe; else the thread is suspended. The entire operation is done atomically.	Easy to implement if a thread makes a blocking system call. The sequence of operations are: • Thread makes a call • Trap to kernel • Thread is suspended • Kernel starts a new thread

Threads and Remote execution

There are two different ways of remote execution of thread :

- RPC : distributed systems commonly use RPC (remote procedure call).
- RMI (remote method invocation) and Java threads



Types of RPC execution in distributed system

Clients and Servers

- What's a client?
- *Definition*: "A program which **interacts** with a human user and a remote server."
- Typically, the user interacts with the client via a GUI.
- Of course, there's more to clients than simply providing a UI. Remember the multi-tiered levels of the Client/Server architecture from earlier …

What's a Server?

- *Definition*: "A process that implements a specific service **on behalf of** a collection of clients".

- Typically, servers are organized to do one of two things:
  1. Wait
  2. Service

… wait … service … wait … service … wait …

Servers: Iterative and Concurrent

- *Iterative*: server handles request, then returns results to the client; any new client requests *must wait* for previous request to complete (also useful to think of this type of server as *sequential*).

- *Concurrent*: server does not handle the request itself; a separate thread or sub-process handles the request and returns any results to the client; the server is then free to immediately service the next client (i.e., there's no waiting, as service requests are processed in *parallel*).

Server "States"

- **Stateless servers** – no information is maintained on the current "connections" to the server. The web is the classic example of a *stateless service*. As can be imagined, this type of server is **easy** to implement.

- **Stateful servers** – information is maintained on the current "connections" to the server. Advanced file servers, where copies of a file can be updated "locally" then applied to the main server (as the server knows the state of things). These are more **difficult** to implement.

Code Migration

- Under certain circumstances, in addition to the usual passing of data, *passing code* (even while it is executing) can greatly simplify the design of a distributed system.

- However, code migration can be inefficient and very costly.

- So, why migrate code?

Reasons for Migrating Code

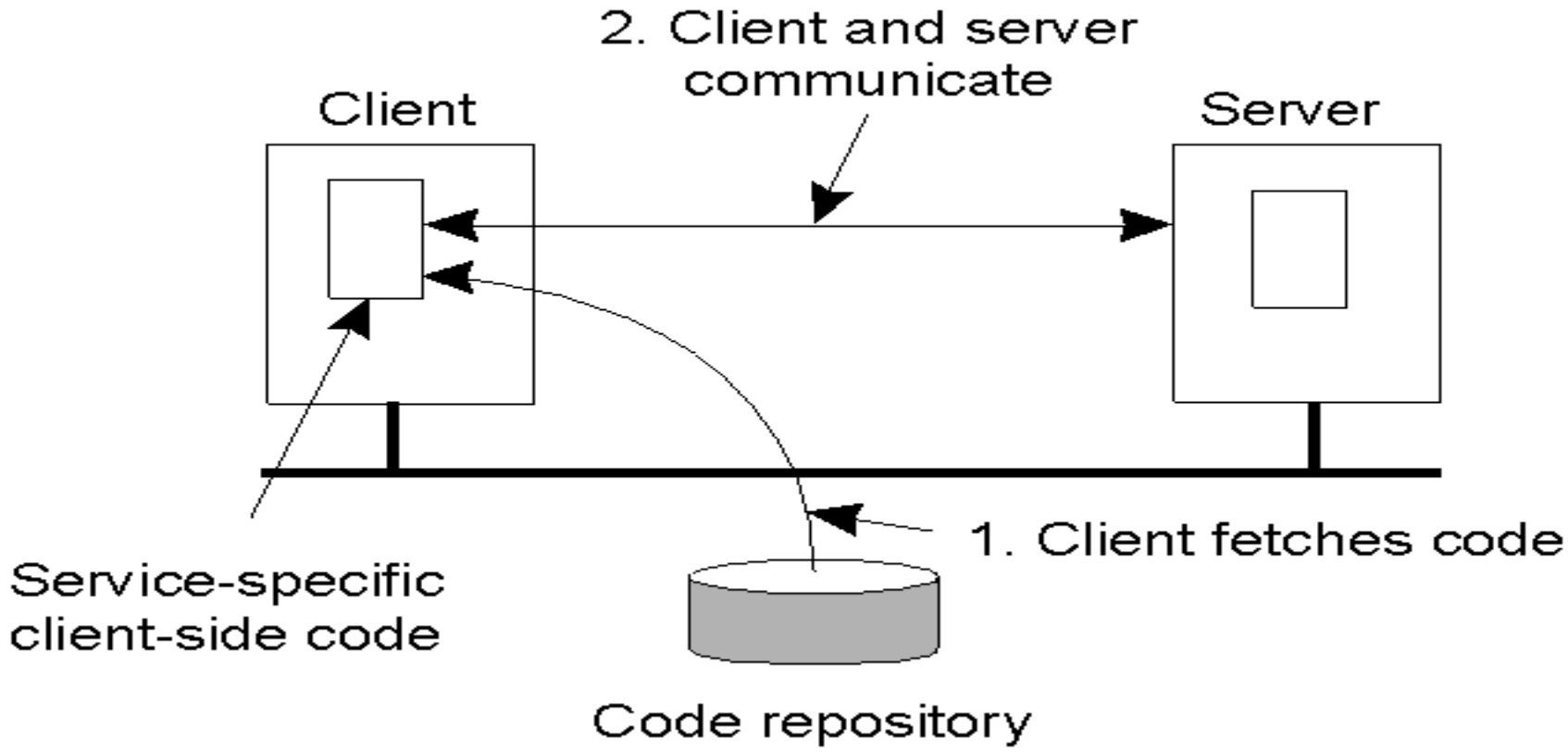
- Why?
- Biggest single reason: **better performance**.
- The big idea is to move a compute-intensive task from a *heavily loaded* machine to a *lightly loaded* machine "on demand" and "as required".

Code Migration Examples

- *Moving (part of) a client to a server* – processing data close to where the data resides. It is often too expensive to transport an entire database to a client for processing, so move the client to the data.

- *Moving (part of) a server to a client* – checking data prior to submitting it to a server. The use of local error-checking (with JavaScript) on web forms is a good example of this type of processing. Error-check the data close to the user, not at the server.

- The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.



Major Disadvantage

- **Security Concerns**.

- "Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard-disk and does not send the juiciest parts to heaven-knows-where may not always be such a good idea".

Code Migration Models

- A running process consists of three "segments":
  - *Code Segment* – instructions
    - The part that contains the set of instructions that make up the program that is being executed.
  - *Resource Segment* – external references.
    - The part that contains references to external resources needed by the process, such as files, printers, devices, other processes, and so on.
  - *Execution Segment* – current state.
    - Used to store the current execution state of a process, consisting of private data, the stack, and the program counter.

Code Migration Characteristics

- **Weak Mobility**: just the code is moved – and it always restarts from its initial state.
- e.g. Java Applets.
- Comment: simple implementation, but limited applicability.

- **Strong Mobility**: the code *and* the state is moved – and execution restarts from the next statement.
- e.g. D'Agents.
- Comment: very powerful, but hard to implement.

More Characteristics

- Sender- vs. Receiver-Initiated.

- Which side of the communication starts the migration?

- The machine currently executing the code (known as *sender-initiated*), or
- The machine that will ultimately execute the code (known as *receiver-initiated*).

How Does the Migrated Code Run?

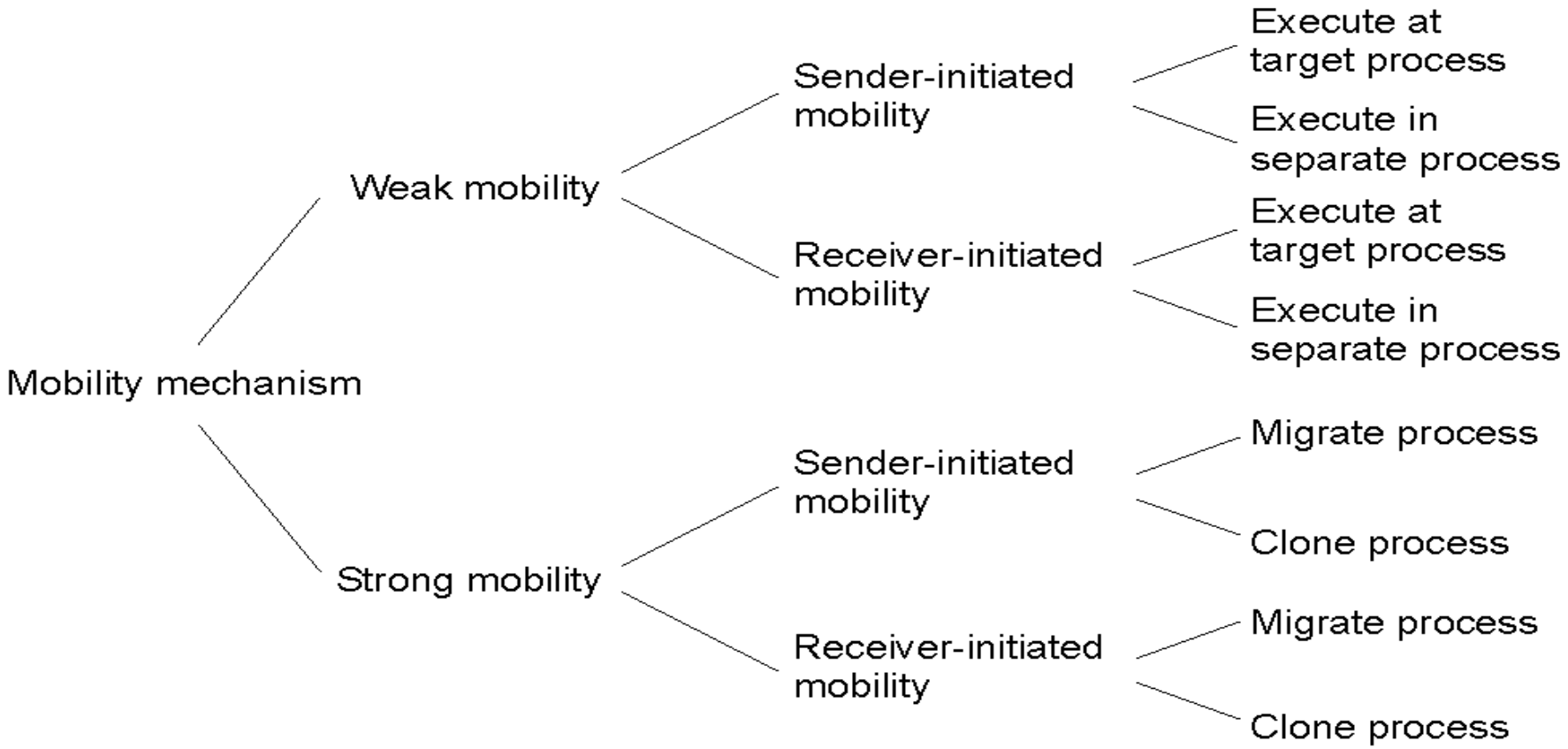
- Another issue surrounds where the migrated code executes:

- Within an existing process (possibly as a thread) or Within it's own (new) process space.

- Finally, strong mobility also supports the notion of "remote cloning": *an exact copy of the original process, but now running on a different machine*.

Models for Code Migration

- Alternatives for code migration.



What About Resources?

- What makes code migration difficult is the requirement to migrate resources.
- Resources are the *external references* that a process is currently using, and includes (but is not limited to):
- Variables, open files, network connections, printers, databases, etc.

Types of Process-to-Resource Binding

- **Strongest**: *binding-by-identifier* (BI) – precisely the referenced resource, and nothing else, has to be migrated.

- *Binding-by-value* (BV) – weaker than BI, but only the value of the resource need be migrated.

- **Weakest**: *binding-by-type* (BT) – nothing is migrated, but a resource of a specific type needs to be available after migration (eg, a printer).

More Resource Classification

- Resources are further distinguished as one of:
  1. **Unattached**: a resource that can be moved easily from machine to machine.
  2. **Fastened**: migration is possible, but at a high cost.
  3. **Fixed**: a resource is bound to a specific machine or environment, and cannot be migrated.

- Refer to diagram 3-14 in the textbook for a good summary of resource-to-binding characteristics (to find out *what to do with which resource when*).

Migration and Local Resources

Process-to-resource binding	Resource-to-machine binding: Unattached	Fastened	Fixed
By identifier	MV (or GR)	GR (or MV)	GR
By value	CP (or MV,GR)	GR (or CP)	GR
By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR	Establish a global systemwide reference
MV	Move the resource
CP	Copy the value of the resource
RB	Rebind process to locally-available resource

- Actions to be taken with respect to the references to local resources when migrating code to another machine.