DISTRIBUTED SYSTEM MANAGEMENT

RESOURCE MANAGEMENT

RESOURCE MANAGEMENT

- Every distributed system consists of a number of resources (physical or logical) interconnected by a network and there are multitude of processes competing to use these resources
- To ensure that there are no long waits by a user process in accessing a particular resource, a Distributed System in general has replication of resources connected to different nodes of the system
- In order to improve performance of the Distributed System as whole, there are two ways this can be achieved
  - i.e., either move the resource to point near to the process or move the process to the node having the resource
  - Besides providing communication facilities, the network facilitates resource sharing by migrating a local processes and executing it at a remote node of the network, in order to improve the performance

RESOURCE MANAGEMENT (CONT'D)

- In practice resource manager function can be either centralized or distributed in such a manner that each manage certain group of nodes and a set of resources
- The resource manager performs its function in a manner that ensures optimization of the
  - usage of resources,
  - response time,
  - network traffic,
  - scheduling overhead
  - Process migration overhead and
  - overall performance of the Distributed System

RESOURCE MANAGEMENT

- These scheduling techniques can be broadly classified into three types:
  - Task assignment approach: in which each process submitted by a user is viewed as a collection of related tasks and these tasks are scheduled to suitable nodes so as to improve performance
  - Load balancing approach: in which all the processes submitted by the users are distributed among the nodes of the system so as to equalize the work load among the nodes
  - Load sharing approach: which simply attempts to conserve the ability of system to perform work by assuring that no node is idle while processes wait for being processed at other nodes
- The task assignment approach has limited applicability in practical situations because it works on the assumption that the characteristics of all the processes to be scheduled are known in advance

DESIRABLE FEATURES OF SCHEDULING ALGORITHM

1. No a priori knowledge about the processes
2. Dynamic in nature
3. Quick decision making capability
4. Balanced system performance and scheduling overhead
5. Stability
6. Scalability
7. Fault tolerance
8. Fairness of service

TASK ASSIGNMENT APPROACH

TASK ASSIGNMENT APPROACH

- In this approach, a process is considered to be composed of multiple tasks and goal is to find an optimal assignment for tasks of an individual process
- Typical assumptions found in task assignment work are as follows
  - Process has already been split into pieces called tasks
  - Amount of computation required by each task & speed of each processor are known
  - The cost of processing each task on every node is known
  - Inter process communication (IPC) costs between every pair of tasks is known
  - The IPC cost is considered to be zero (negligible) for tasks assigned to the same node
  - The IPC cost are usually estimated by an analysis of the static program of a process

TASK ASSIGNMENT APPROACH (CONT'D)

- For example if two tasks communicate *n* times and if the average time for each intertask communication is *t*, the intertask communication cost for the two tasks is *n* x *t*
- Other constraints such as resource requirements of the tasks and the available resources at each node, precedence relationships among tasks and so on, are also known
- Reassignment of tasks is generally not possible

TASK ASSIGNMENT APPROACH (CONT'D)

- With these assumptions, the task assignment algorithms seek to assign the tasks of a process to the nodes of the distributed system in such a manner so as to achieve goals such as the following
  - Minimization of IPC costs
  - Quick turnaround time for the complete process
  - High degree of parallelism
  - Efficient utilization of system resources in general
- These goals often conflict with each other, e.g., while minimizing IPC tends to assign all the tasks of a process to a single node, efficient utilization of system resources tries to distribute the tasks evenly among the nodes

AN EXAMPLE

- Similarly, while quick turnaround time and a high degree of parallelism encourage parallel execution of the tasks, the precedence relationship among tasks limits their parallel execution
- All the required resources may not be available at all the nodes of the system
- Let us consider an example
  - Involves only two assignment parameters – the task execution cost and inter-task communication cost
  - The system consists of six tasks (t_1, t_2, t_3, t_4, t_5, t_6) and two nodes (n_1, n_2)

a) **Inter-task communications cost**

	t_1	t_2	t_3	t_4	t_5	t_6
t_1	0	6	4	0	0	12
t_2	6	0	8	12	3	0
t_3	4	8	0	0	11	0
t_4	0	12	0	0	5	0
t_5	0	3	11	5	0	0
t_6	12	0	0	0	0	0

b) **Execution costs**

Tasks	Nodes	
	n_1	n_2
t_1	5	10
t_2	2	∞
t3	4	4
t_4	6	3
t_5	5	2
t_6	∞	4

AN EXAMPLE

- Note that assignment is aimed at minimizing the total execution cost

c) **Serial assignment**

Task	node
t_1	n_1
t_2	n_1
t_3	n_1
t_4	n_2
t_5	n_2
t_6	n_2

d) **Optimal assignment**

Task	node
t_1	n_1
t_2	n_1
t_3	n_1
t_4	n_1
t_5	n_1
t_6	n_2

AN EXAMPLE

- Serial assignment execution cost

  $t_{11} + t_{21} + t_{31} + t_{42} + t_{52} + t_{62} = 5+2+4+3+2+4 = 20$
- Serial assignment communication cost

  $c_{14} + c_{15} + c_{16} + c_{24} + c_{25} + c_{26} + c_{34} + c_{35} + c_{36}$
  $=0+0+12+12+3+0+0+11+0 = 38$
- Serial assignment total cost = 20+38 = 58
- Optimal assignment execution cost

  $t_{11} + t_{21} + t_{31} + t_{41} + t_{51} + t_{62} = 5+2+4+6+5+4 = 26$
- Optimal assignment communication cost

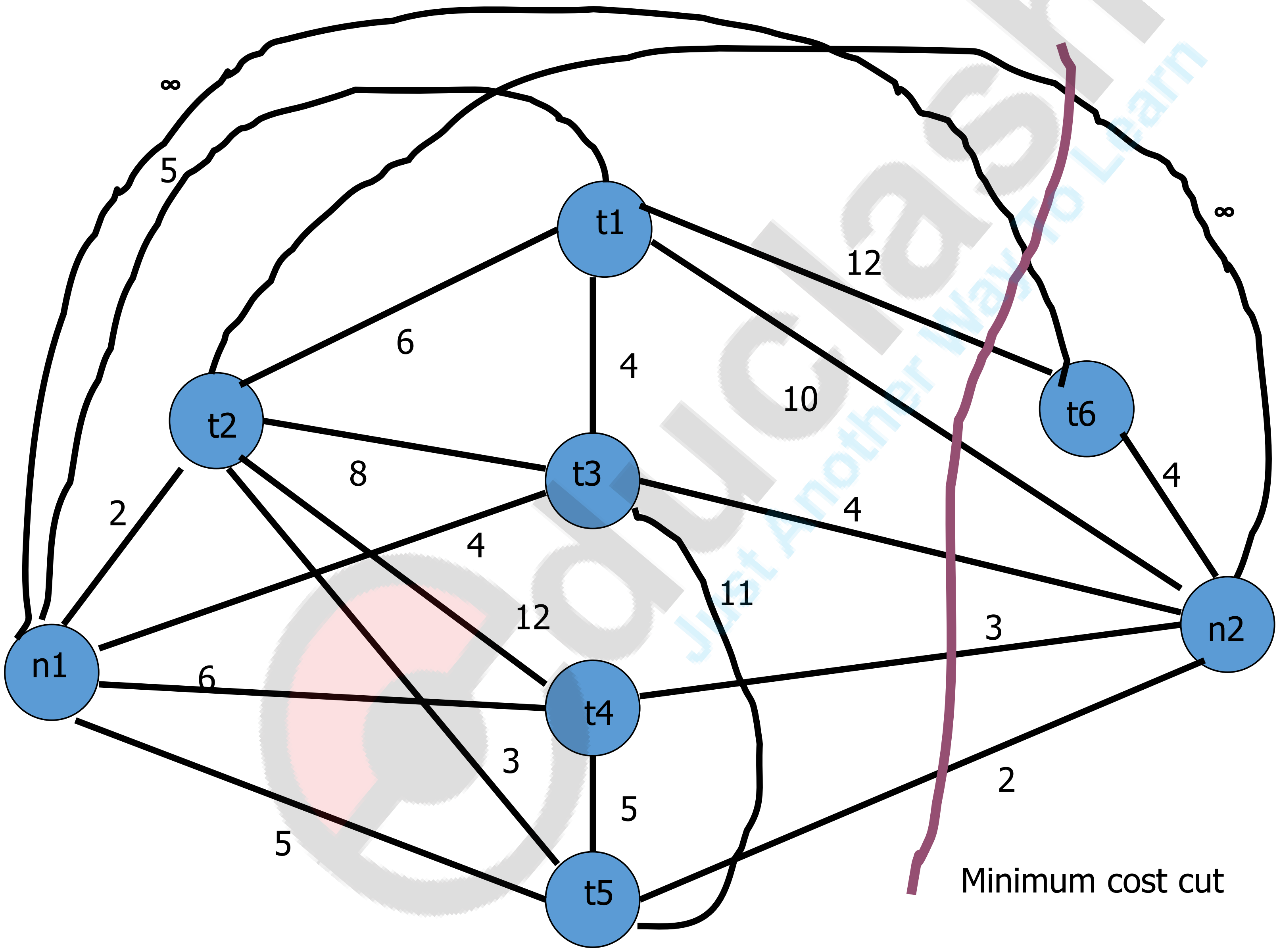
  $c_{16} + c_{26} + c_{36} + c_{46} + c_{56} = 12 +0+0+0+0 = 12$
- Optimal assignment total cost = 26+12 = 38

TASK ASSIGNMENT APPROACH (CONT'D)

- If we take the execution and communication cost comes out be 58
- Fig d) shows an optimal assignment of tasks to two nodes that minimizes the total execution and communication cost, though execution cost is more the total cost is only 38

- Drawbacks
  - Characteristics of all processes should be known in advance
  - Does not take care of dynamically changing state
  - A priori estimation of characteristics of the processes are based on static conditions and may be on different hardware

OPTIMAL ASSIGNMENT USING MINIMAL CUTSET

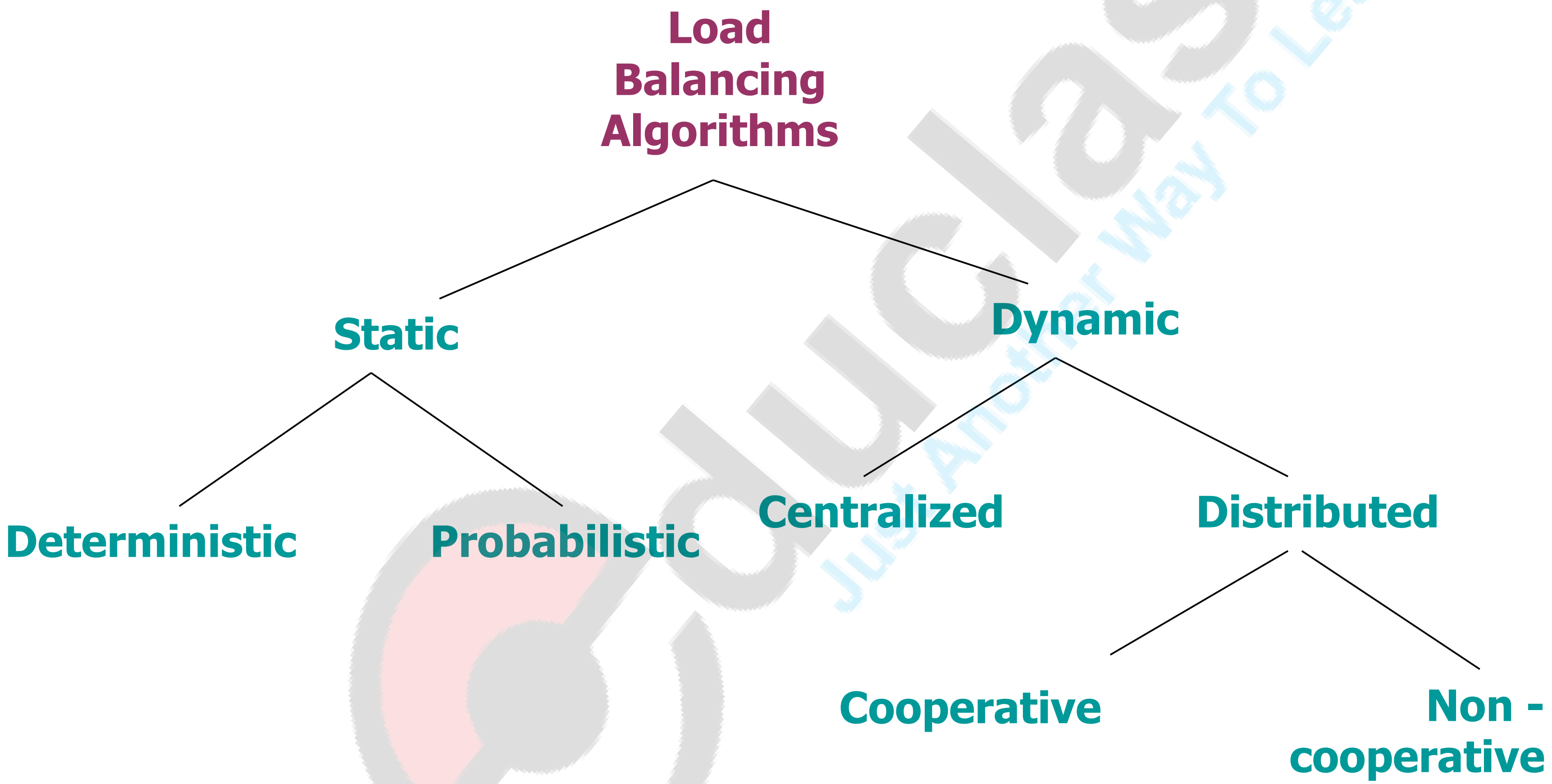


LOAD BALANCING

LOAD BALANCING APPROACH

- A load balancing algorithm tries to balance the total system load by transparently transferring the workload from heavily loaded nodes to lightly loaded nodes to maximize the total system throughput
- While considering the performance from the user point of view the metric involved is often the response time of the processes
- The basic goal of almost all the load balancing algorithms is to maximize total system throughput

A TAXONOMY OF LOAD BALANCING ALGORITHMS



STATIC VS DYNAMIC

- At the highest level we distinguish between Static and Dynamic load balancing algorithms
- Static
  - Use only average behavior of system, ignoring current state of the system
  - System Decisions are hard-coded into an algorithm with a priori knowledge of system
  - These algorithms are simpler because, there is no need to maintain and process system state information
  - However, the potential of static algorithms is limited as the algorithms do not respond to current system state

STATIC VS DYNAMIC

- Dynamic
  - Advantage of dynamic systems is that they react to the system state that changes dynamically and so are able to avoid those states with unnecessarily poor performance
  - Have greater performance benefits than static policies
  - Since they need to collect and react to system state information, they are necessarily more complex than static algorithms

STATIC ALGORITHMS

- Static load-balancing algorithms may be either deterministic or probabilistic
- Deterministic
  - Deterministic algorithms use information about the properties of the nodes and the characteristics of the processes to be scheduled to deterministically allocate processes to nodes. e.g. Task assignment approach belong to this category of deterministic static algorithm
- Probabilistic
  - Probabilistic algorithm uses information regarding static attributes of the system such as number of nodes, processing capability of each node, network topology etc to allocate nodes to processes
- In general the deterministic approach is difficult to optimize and costs more to implement

DYNAMIC ALGORITHMS

- Dynamic scheduling algorithms may be centralized or distributed
- Centralized
  - The single node, known as centralized server node collects system state information and is responsible for all scheduling decisions
  - This approach is efficient in process assignment decisions as it knows both load at each node and number of processes requiring service
  - Other nodes periodically send status update information to the central node and they are used to maintain the state information up to date
  - One of the problems with centralized mechanism is that of reliability(single-point-of-failure)
  - A typical solution to overcome this problem would be to replicate the server on K+1 nodes

DYNAMIC ALGORITHMS (CONT'D)

- In this model cost of maintaining k+1 replicas of server consistent can be considerably high
  - Another approach is, instead of maintaining k+1 server replicas, a single server is maintained and there are k entities monitoring the server to detect its failure
  - When failure is detected, a new instance of the server is brought up, which reconstructs its state information by sending a multicast messages requesting immediate state information

DYNAMIC ALGORITHMS (CONT'D)

- Distributed
  - The distributed scheme does not limit the scheduling intelligence to one node
  - In dynamic distributed scheduling algorithm, the work involved in making process assignment decisions is physically distributed among the various nodes of the system
  - Uses k physically distributed entities that work as local controller, where each is responsible for making scheduling decisions for processes of a predetermined set of nodes
  - Each local controller makes decisions based on system wide objective function, rather than on a local one
  - In a fully distributed system each node acts as local controller making scheduling decisions for processes of its own node, which includes both transfer of local processes and acceptance of remote processes

DISTRIBUTED ALGORITHMS (CONT'D)

- Non-cooperative
  - Individual entities act as autonomous entities and make scheduling decision independently of the actions of other entities
  - Less stable
- Cooperative
  - Distributed entities cooperate with each other to make scheduling decision
  - More complex and involve larger overhead
  - But more stable when compared to non-cooperative algorithms

ISSUES IN DESIGNING LOAD BALANCING ALGORITHM

- All processes are distributed among nodes of the system to equalize workload among nodes but designing a distributed dynamic load balancing algorithm is a difficult task
- Design Issues
  - Load estimation policy: determines the work load of a particular node of the system
  - Process transfer policy: determines whether to execute locally or remotely
  - State information exchange policy: determines how to exchange the system load information among the nodes
  - Location policy: determines to which node a process selected for transfer should be sent

1. LOAD ESTIMATION POLICY

- Priority assignment policy: determines the priority of execution of local and remote processes at a particular node
- Migration limitation policy: determines the total number of times a process can migrate from one node to another

- Load Estimation Policy
  - The main goal of load-balancing algorithms is to balance the workload on all the nodes of the system
  - How to estimate the workload of a particular node in the system?

LOAD ESTIMATION POLICY (CONT'D)

- Several load balancing algorithms use total no of processes present on the node as a measure of the node's workload
- Above method is not suitable because of existence of daemon processes
- Other way is, estimate the remaining CPU service time of the processes
- CPU utilization measured by observing CPU state
- A better estimate is the CPU utilization of the nodes: it is defined as the number of CPU cycles actually executed per unit of real time

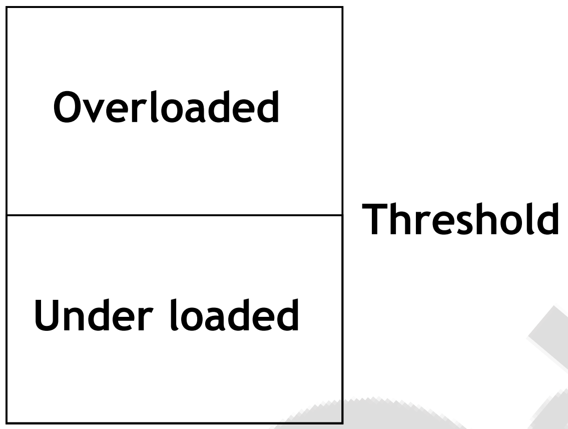
PROCESS TRANSFER POLICY

- The strategy of load balancing is based on the concept of moving some of the processes from a heavily loaded nodes to lightly loaded nodes
- Most of the algorithms use Threshold policy: and its value is limiting value of a nodes workload whether lightly or heavily loaded
- It can be determined by one of following methods
  - Static policy
    - Each node has a predefined threshold value depending on its processing capability and does not change dynamically
    - The main advantage of the method is that no exchange of state information among the nodes is required for deciding the threshold value.

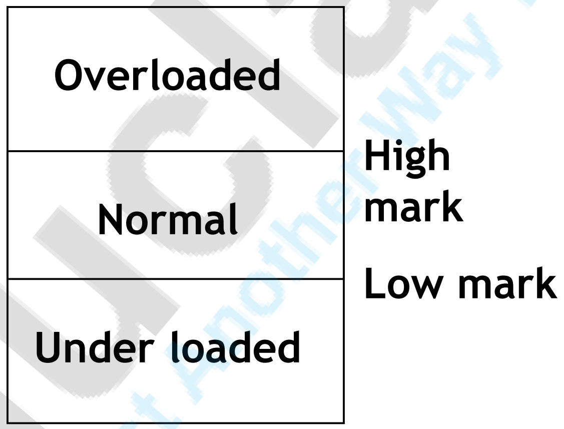
PROCESS TRANSFER POLICY (CONT'D)

- Dynamic policy
  - Threshold value calculated as a product of average workload of all nodes.
  - Only 1 or 2 processes must be transmitted.
  - Incoming remote processes must not effect local processes.
- To reduce the instability of single threshold double threshold policy was proposed called high-low policy
  - It uses two threshold values the high mark and low mark based on which the possible load states are divided into three states as
    - Overloaded : above the high mark
    - Normal: between the high and low mark
    - Underloaded: below both marks

PROCESS TRANSFER POLICY (CONT'D)



Single-threshold policy



Double-threshold policy

- Depending on the current load status of the node, the decision to transfer a local process or accept a remote process is based on the following policies
  - When the load of the node is in overload region, new local procedures are sent to be run remotely and requests to accept remote processes are rejected
  - When the load of node is normal region, new local procedure run locally and request to accept remote processes are rejected
  - When the load of the node is in the underloaded region, new local processes are run locally and requests to accept remote processes are accepted

LOCATION POLICY

- Once the decision is taken to transfer a process from a node, the next is to select the destination node
- Random
  - Destination nodes selected randomly to check whether node is able to receive the process
  - If yes then transfer the process, else another node is selected randomly
  - This continues until a static probe limit L_p is reached, else process is executed at originating node

LOCATION POLICY (CONT'D)

- Shortest
  - L_p distinct nodes are chosen at random & polled to determine its load
  - Process is transferred to node having minimum load unless its workload value prohibits to accept the process
  - If none of polled nodes can accept process, it is executed at originating node
  - Once a destination node is decided, and the process is transferred, it must execute the process regardless of its state at the time the process actually arrives
  - A simple improvement to the shortest policy is to discontinue probing whenever a node with zero load is encountered
  - It was observed that shortest policy uses more state information and hence more complex, but does not show much performance improvement over threshold policy

LOCATION POLICY (CONT'D)

- Bidding
  - In this method the system is turned into a distributed computational economy with buyers and sellers of service
  - Each node can act as manager (send process) and contractor (receive process)
  - Note that a single node takes on both these roles and no node is strictly managers or contractors alone under different conditions
  - To select a node for its process managers broadcast request for bid, contractors respond with bids (prices based on capacity of the contractor node) and manager selects the best offer which may the cheapest, fastest or best price-performance, based on the application
  - Once the best bid is determined, winning contractor is notified and asked if it accepts bid or not

LOCATION POLICY (CONT'D)

- The contractor may have bid for many requests and thus become overloaded and hence reject the acceptance message
- If bid is rejected, bidding is started again

- Pairing
  - The method of pairing policy is to reduce the variance of loads only between pairs of nodes of the system
  - Two nodes that differ greatly in load are temporarily paired with each other and the load balancing is carried out between the two nodes by migrating one or more processes from the more heavily loaded node to the other
  - After the formation of the pair, one or more processes are migrated from heavily loaded node of the two nodes to the other node to balance the load between these two nodes
  - The processes to be migrated are selected by comparing their expected time to complete on their current node with the expected time to complete on its partner and migration delay is included in the estimate
  - The pair is broken as soon as the migration is over

STATE INFORMATION EXCHANGE POLICY

- We have seen that dynamic policies require frequent exchange of state information among the nodes of the system
- How to exchange load information among nodes?
- Periodic broadcast
  - Each node broadcasts its state information after the elapse of every T units of time
  - This method is not good, as it generates heavy network traffic and unwanted messages from nodes whose state has not changed in the last T time units
  - Poor scalability

STATE INFORMATION EXCHANGE POLICY

- Broadcast when state changes
  - Avoids fruitless messages by broadcasting state information only on arrival or departure of a process or change of state
  - A further improvement in this method can be obtained by observing that it is not necessary to report all minor change in the state of a node to all other nodes, because it can participate in the load balancing process, only when either underloaded or overloaded called on-demand exchange.

STATE INFORMATION EXCHANGE POLICY

- On-demand exchange
  - In this method a node broadcasts a *StateInformationRequest* message when its state changes to either underloaded or overloaded state
  - On receiving the *StateInformationRequest* message, other nodes send their current state to the requesting node
  - This can be further refined as only those nodes need to reply that can co-operate with it in load balancing process
  - i.e., if the requesting node is underloaded, only overloaded nodes can cooperate with it in the load balancing process and vice versa

STATE INFORMATION EXCHANGE POLICY

- Exchange by polling
  - All the above methods use broadcasting due to which their scalability is poor
  - The polling mechanism is based on the idea that there is no need for a node to exchange its state information with all other nodes in the system
  - Hence state information is exchanged only between the polling node and the polled nodes
  - The polling process stops on either finding a suitable partner or a predefined poll limit is reached

PRIORITY ASSIGNMENT POLICY

- When process migration is supported by a distributed operating system, it becomes necessary to devise a priority assignment rule for scheduling both local and remote processes on a particular node
- One of the following priority assignment rules may be used
- Selfish
  - Local processes are given higher priority than remote processes
- Altruistic
  - Remote processes are given higher priority than local processes
- Intermediate
  - When number of local processes is greater or equal to number of remote processes, local processes are given higher priority, otherwise remote processes are given higher priority

MIGRATION LIMITING POLICIES

- Migration limiting Policies
- Another important policy to be used in Distributed Operating System that support process migration, is to decide about the total no. of times a process should be allowed to migrate
  - Uncontrolled: A process may migrate any no of times
  - This policy has the property of causing instability
  - Controlled: To overcome the instability problem of the uncontrolled policy, most systems treat remote processes different from local processes and use a migration count to fix a limit on the number of times a process can migrate

LOAD SHARING APPROACH

LOAD SHARING APPROACH

- It is necessary and sufficient to prevent nodes from being idle while other nodes have multiple processes running
- This modification is called dynamic load sharing instead of dynamic load balancing

LOAD SHARING APPROACH (CONT'D)

- Issues in Designing Load Sharing Algorithms
  - Similar to load balancing algorithms, the design of a load sharing algorithm also require that proper decisions be made regarding load estimation policy, process transfer policy, state information exchange policy, location policy, priority assignment policy, and migration limiting policy
  - However, when compared to load balancing, it is simpler as the policies of load sharing does not attempt to balance the workload on all the nodes of the system, like load balancing algorithms try to do
  - They only attempt to ensure that no node is idle when other nodes are heavily loaded

LOAD SHARING APPROACH (CONT'D)

- We will discuss different policies in detail for load sharing approach

1. Load estimation policy
   - Load sharing algorithms normally employ the simplest load estimation policy of counting the total number of processes on a node
   - Simple count of total number of processes on a node is not a good estimate as there are several idle daemon processes in a modern day distributed system
   - Hence a measure of CPU utilization should be used as a method of load estimation

LOAD SHARING APPROACH (CONT'D)

2. Process transfer policy
   - As load-sharing algorithms are normally interested only in busy or idle states of node, most of them use All-or-nothing strategy
   - The strategy uses single process policy with a single threshold value for all nodes fixed at 1
   - Nodes can receive process when it has no process, and send process when it has more than 1 process
   - The all-or-nothing strategy is not good in the sense that a node that becomes idle is unable to immediately acquire a new process even though processes wait for service at other nodes leading to loss of processing power in the Distributed System
   - To address this anticipatory transfers to that are not idle, but are expected to become idle soon is necessary

- Location Policy
- In load-sharing algorithms, the location policy decides the sender node or the receiver node of a process that is to be moved within the system for load sharing
- Based on the type of node that takes the initiative to globally search for a suitable node for the process, the location policies are adopted
- Sender initiated location policy
  - The sender node decides where to send the process
  - When node becomes overloaded, it either broadcasts or randomly probes other nodes one by one to find a lightly loaded node that can accept one or more of its processes

LOCATION POLICY (CONT'D)

- Receiver initiated location policy
  - When node becomes under-loaded (below threshold), it either broadcasts or randomly probes other nodes indicating its willingness to receive remote processes
  - A node is a viable candidate for sending one of its processes for executing only if it does not reduce its load below the threshold limit
  - Receiver initiated polices require preemptive process migration facility while sender initiated policies can work with out the support of preemptive process migration facility
  - A preemptive migration facility allows the transfer of an executing process from one node to another
  - Preemptive process migration is costlier, since the process state, which must accompany the process to its new node, is much more complex after execution begins

4. STATE INFORMATION EXCHANGE POLICY

- Commonly used policies for this are as follows:
- Broadcast when state changes
  - In sender-initiated/ receiver-initiated location policy a node broadcasts *StateInformationRequest* when it becomes overloaded / underloaded respectively
  - In a sender-initiated policy, a node broadcasts this message only when it becomes overloaded and in receiver-initiated policy, this message is broadcast only when it becomes underloaded

STATE INFORMATION EXCHANGE POLICY

- Poll when state changes
  - Since broadcast is not suitable for large networks, the polling mechanism is normally used in such systems
  - When a nodes state changes, it does not exchange state information with all other nodes but randomly polls other nodes one by one and exchanges state information with the polled nodes
  - The state exchange process stops either when suitable node of sharing load is found or has reached the probe limit

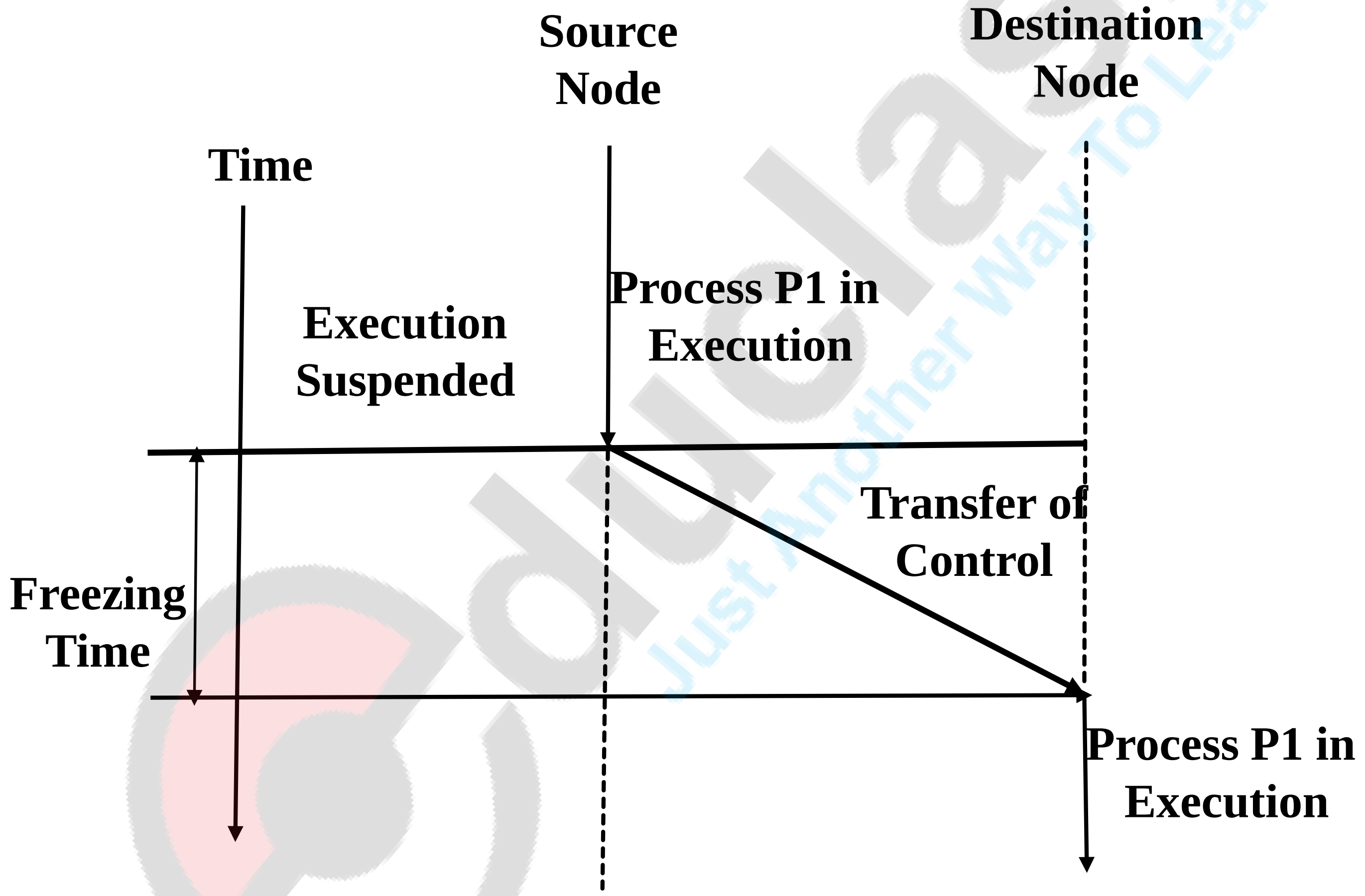
PROCESS MIGRATION

PROCESS MANAGEMENT

- In a DOS, the main goal of the process management is to make the best possible use of the processing resources of the entire system by sharing them among all processes
- Process allocation: Which process should be assigned to which processor; already discussed in resource management
- Process Migration: Movement of process from its current location to new processor
- Threads : Fine grain parallelism for better utilization of the processing capability of the system

PROCESS MIGRATION

- Process migration is the relocation of a process from its current location (source node) to another node (destination node)
- The flow of execution of a migrating process is shown in fig. in the next slide
- Process migration mechanism deals with the actual transfer of the process
- A process may be migrated either before it starts executing called as non-preemptive process migration or
- During the course of its execution called as preemptive process migration


Source Node
Destination Node
Time
Process P1 in Execution
Execution Suspended
Transfer of Control
Freezing Time
Process P1 in Execution

PROCESS MIGRATION (CONT'D)

- Process migration involves following major steps
  - Selection of a process that should be migrated
  - Selecting the destination node to which the selected process should be migrated
  - Actual transfer of the selected process to the destination node

DESIRABLE FEATURES OF GOOD PROCESS MIGRATION MECHANISM

- A good process migration mechanism must possess
  - Transparency – *object access and IPC*
  - Minimal Interference
  - Minimal Residual Dependencies
  - Efficiency
    - Minimize freezing time, cost of locating migrated process
    - Cost of supporting remote execution
  - Robustness
  - Communication between Co-processes of a Job

PROCESS MIGRATION MECHANISM

- Migration of a process is a complex activity that involves proper handling of several sub-activities in order to meet the requirements of a good process migration explained earlier
- The four major sub-activities involved process migration are:
  - freezing the process on its source node and restarting at destination node
  - Moving the process's address space from source node to destination node
  - Forwarding messages meant for the migrant process
  - Handling communication between cooperating processes that are separated (placed on different nodes) as a result of process migration

MECHANISM FOR FREEZING AND RESTARTING A PROCESS

- In Preemptive process migration, the usual process is to take a snapshot of process's state on its source node & reinstate it on the destination node
- For this, at some point during migration, the process is frozen on its source node, its state information is transferred to the destination node, and the process is restarted on the destination node using this state information
- By freezing process, we mean that the execution of process is suspended and all external interactions with the process are deferred
- Though freezing and restarting varies from system to system, some of the general issues are discussed here

IMMEDIATE AND DELAYED BLOCKING

- Before freezing a process, its execution has to be blocked
- The blocking may be immediate or the blocking may have to be delayed until the process reaches a state when it can be blocked
- Some of the typical situations are as follows
  - If a process is not executing a system call it can be immediately blocked
  - If the process is executing a system call, but is sleeping at an interruptible priority (at which any received signal would awaken the process) waiting for a kernel event to occur, it can be immediately blocked from further execution
  - If the process is executing a system call and is sleeping at a non-interruptible priority waiting for kernel event to occur, it can not be blocked immediately

FAST AND SLOW I/O OPERATIONS

- In this situation, a flag is set, telling the process that when the system call is complete, it should block itself from further execution

Fast and slow I/O operations

- In general, after the process has been blocked, the next step in freezing the process is to wait for the completion of all fast I/O operations (e.g., disk I/O associated with the process)
- The process is frozen after the completion of all fast I/O operations
- However, it is not feasible to wait for slow I/O operations to complete, such as those on a pipe or terminal, because the process must be frozen in a timely manner for the effectiveness of process migration
- Resume slow I/O performed at destination

INFORMATION ABOUT OPEN FILES

- Information About Open Files
  - A process state information also contain the information pertaining to files currently open by the process
  - This includes such information as the names or identifier of the files, their access modes, current positions of their file pointers
  - One of the two following approaches are used for this
    - In the first approach, a link is created to the file and pathname of the link is used as an access point to the file after the process migrates
    - In the second approach, an open file's complete pathname is reconstructed when required
    - For this necessary modifications are done in the kernel

REINSTATING THE PROCESS ON ITS DESTINATION NODE

- On the destination node, an empty process state is created that is similar to that allocated during the process creation
- Once all the state information of the migrating process has been transferred from the source node to the destination node and copied into the empty process state, the new process is unfrozen and the old copy is deleted
- Thus the process is restarted on the destination node in what ever state it was before being migrated

ADDRESS SPACE TRANSFER MECHANISMS

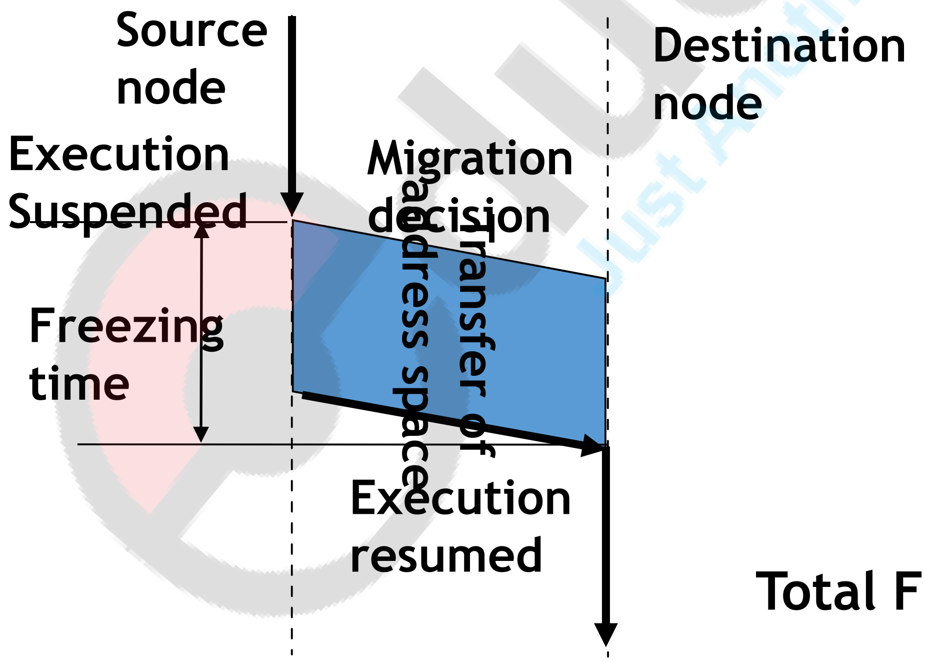
- A process consists of a program being executed, along with the programs data, stack and state
- Hence the migration of a process involves the transfer of the following data
  - Process's state which consist of the execution status (contents of registers), program counter, scheduling information, main memory being used by the process (memory table), I/O states (I/O queue, contents of the I/O buffers, interrupt signals etc.), list of objects to which the process has the right to access(capability list) process's identifier, process's user and group identifiers, information about the files opened by the process (mode, current position of the pointer) etc.
  - Process's address space (code, data & stack of the program) – which is usually more than process's state information.

ADDRESS SPACE TRANSFER MECHANISMS

- Though for transferring of state information the process has to be stopped completely, the address space can be transferred without stopping the execution
- Due to the flexibility in transferring the process's address space at any time after migration decision is made, the existing distributed systems use one of the following address space transfer mechanisms:
  - Total freezing
  - Pretransferring
  - Transfer on reference

TOTAL FREEZING

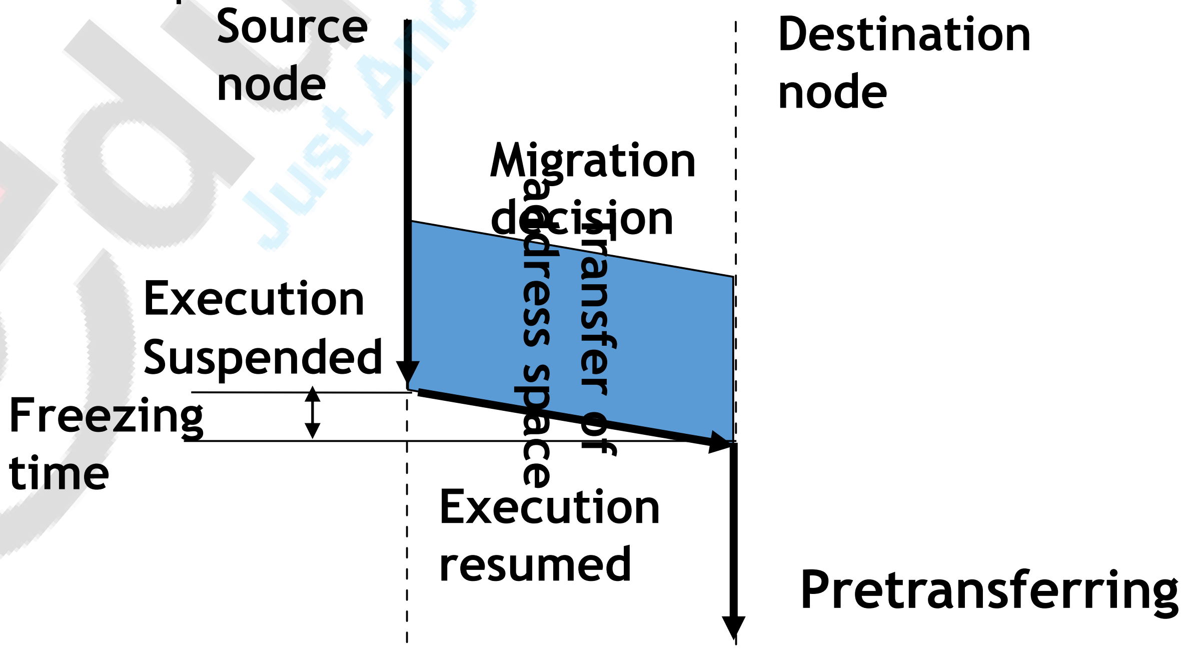
- Process's execution is stopped while transferring the address space
- Simplest and easy to implement but slowest
- Can not be used with interactive processes as it will be noticed by the user



Total Freezing

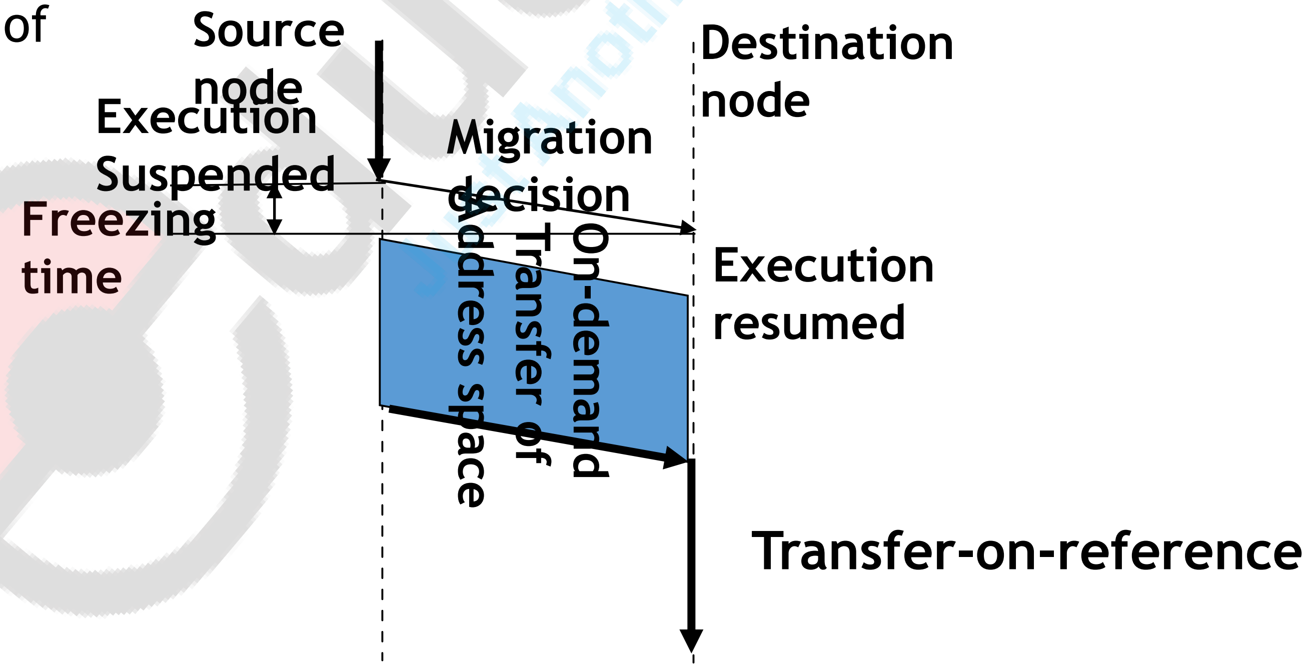
PRETRANSFERRING (PRECOPYING)

- Address space is transferred while the process is still running on the source node
- After the transfer, the modified pages are retransferred
- Freezing time is reduced
- Migration time may increase due to possibility of redundant transfer of same pages
- Operation is executed at higher priority than all other programs in the source node to facilitate interrupt free address space transfer



TRANSFER ON REFERENCE

- This method is based on the principle of spatial locality (processes tend to use small part of address space while executing
- Process starts executing at destination before the address space is migrated
- Pages are fetched from the source node as required demand-driven, copy-on-reference approach
- Process continues to impose load on source node
- Freezing time very less
- Failure of source node results in failure of process



MESSAGE FORWARDING MECHANISM

- In moving a process, it must be ensured that all pending, en-route and future messages arrive at the process's new location
- These messages can be classified into three types of messages:
  - Type1: Received when the process execution is stopped on the source node and has not restarted on the destination node
  - Type2: Received on the source node after the execution started on destination node
  - Type3: Sent to the migrant process after it started execution on destination node
- The different mechanisms used for message forwarding are:
- Mechanism of resending the message
  - Can handle all three type messages

MESSAGE FORWARDING MECHANISM

- Message type 1 and 2 are returned to sender or simply dropped, with the assurance that sender of the message is storing a copy of data and prepared to retransmit it
- Sender retries after locating the new node using a locate operation to find the new whereabouts of the process
- Type 3 message directly sent to new node
- Main drawback is that message forwarding mechanism not transparent to the processes interacting with the migrant process

- Origin site mechanism
  - The process identifier of these systems has the process's origin site (or home node) embedded in it
  - Each site keeps information about current locations of all processes created on it

MESSAGE FORWARDING MECHANISM (CONT'D)

- All messages are sent to origin site
- Origin site forwards messages to process's current location
- If origin site fails, forwarding mechanism fails – reliability issue
- Continuous load on the origin site even after migration

- Link traversal mechanism
  - Message queue created at origin for type 1 & sent to destination as a part of the migration procedure
  - On migration, *link* of destination node is left on source node
  - Thus to forward type 2 and 3 messages, a migrated process is located by traversing a series of links (starting from the node, where the process was originally created) forming chain ultimately leading the process final destination

MESSAGE FORWARDING MECHANISM (CONT'D)

- Its main drawbacks are poor efficiency and reliability as several links may have to be traversed to locate a process
- Process can't be located if any node in chain of links fail

- Link update mechanism
  - During transfer phase the source node sends link update messages to the kernels controlling all of the migrant process's communication partners.
  - This task is not expensive as its performed in parallel
  - Type 1 and 2 messages are sent via source node, type 3 sent directly

CO-PROCESSES HANDLING

- Need to provide efficient communication between a parent process and its sub-processes (children), which might have migrated and placed on different nodes.
- Two different mechanisms to address this problem are:
  - Disallow separation of co-processes: the simplest method is to disallow their separation which can be achieved by
    - Disallow migration of processes that wait for their children to complete
    - Ensure that when parent process migrates all its child processes also migrates along with it
  - Home node origin
    - Communication between parent process & its children processes take place via home node increasing the message traffic and communication cost considerably

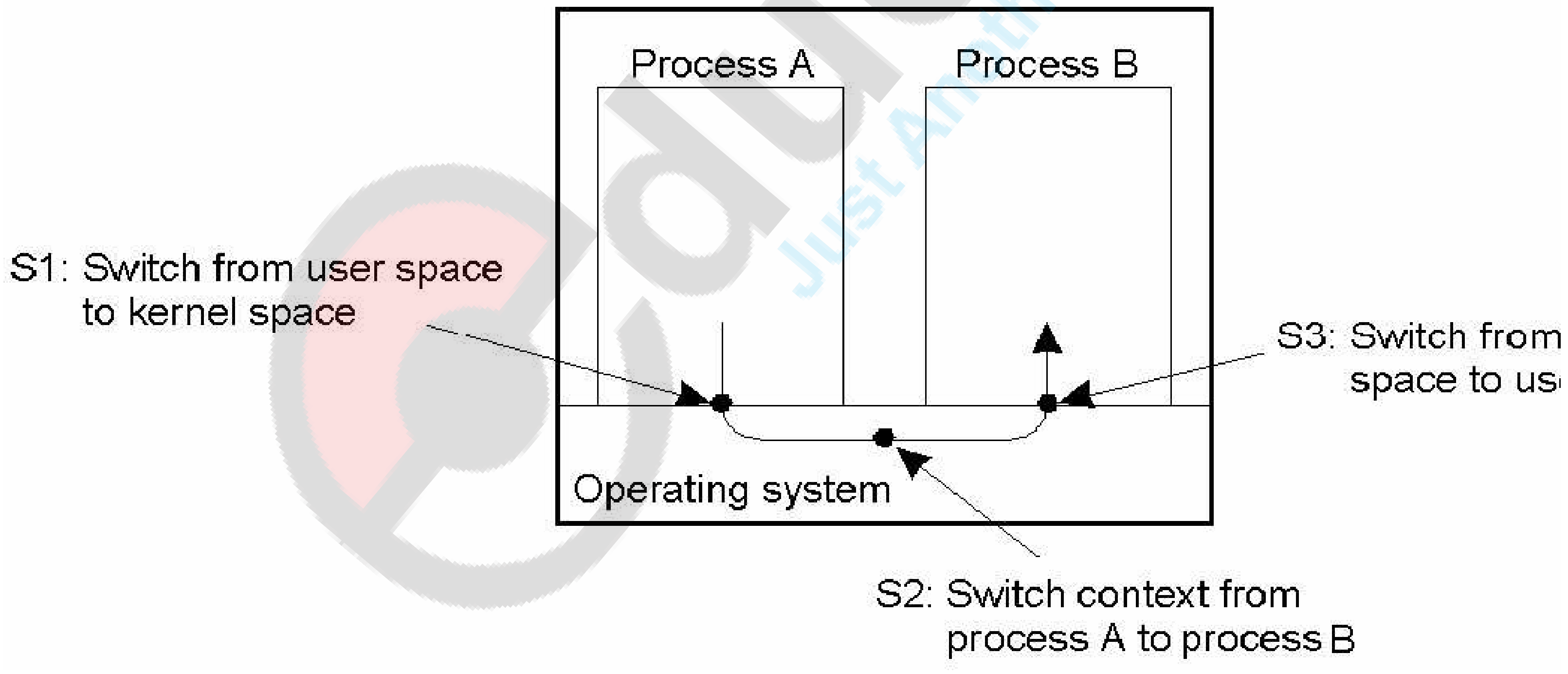
ADVANTAGES OF PROCESS MIGRATION

- Reducing average response time of processes by balancing workload
- Higher throughput
- Speeding up individual jobs by concurrent execution
- Utilizing resources effectively
- Reducing network traffic by migrating process closer to resource
- Improving system reliability by migrating critical processes to more reliable node
- Improving system security by migrating sensitive processes to more secure node

THREADS

PROCESSES

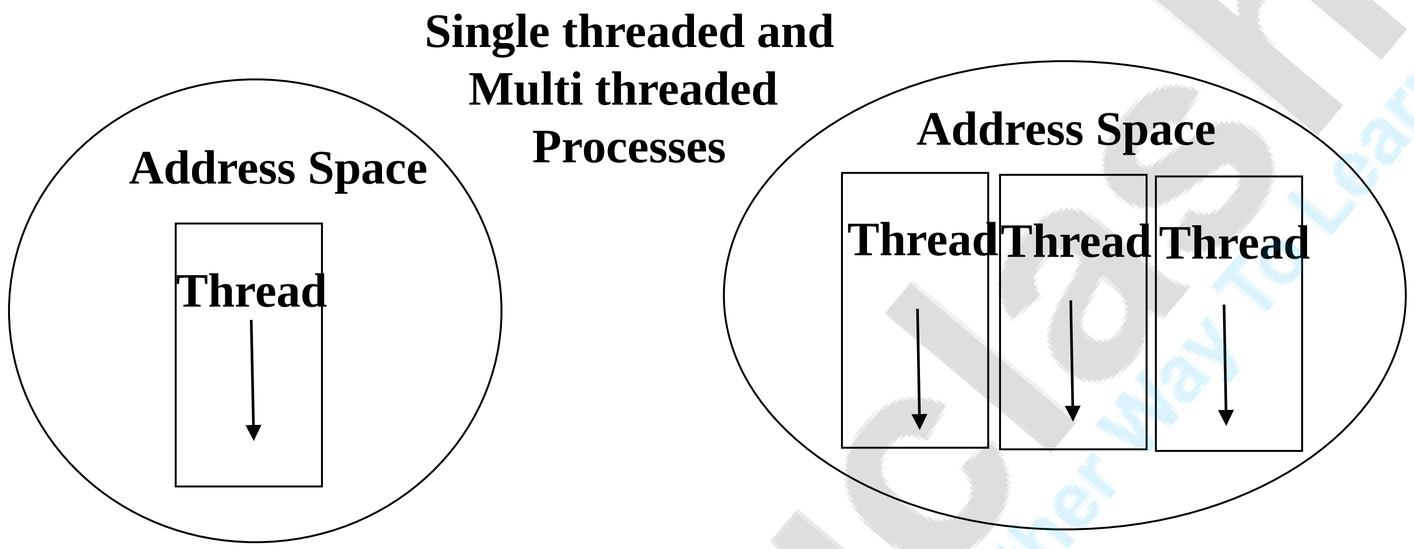
- Processes are independent execution units that contain their own state information, use their own address spaces, and only interact with each other via IPC
- Inter-process communication is expensive
- Context Switch expensive but Secure: one process cannot corrupt another process


Process A
Process B
Operating system
S1: Switch from user space to kernel space
S3: Switch from space to us
S2: Switch context from process A to process B

THREADS

- Threads are a popular way to improve application performance through parallelism
- In traditional operating systems the basic unit of CPU utilization is a process
- On the other hand, in OS that support threads the basic unit of CPU utilization is a thread
- In these operating systems, a process has an address space and one or more threads of control as shown in the figure in the next slide
- Each thread of a process has its own program counter, its own register states, and its own stack
- But all the threads of a process share the same address space
- Hence they also share the same global variabless

THREADS (CONT'D)

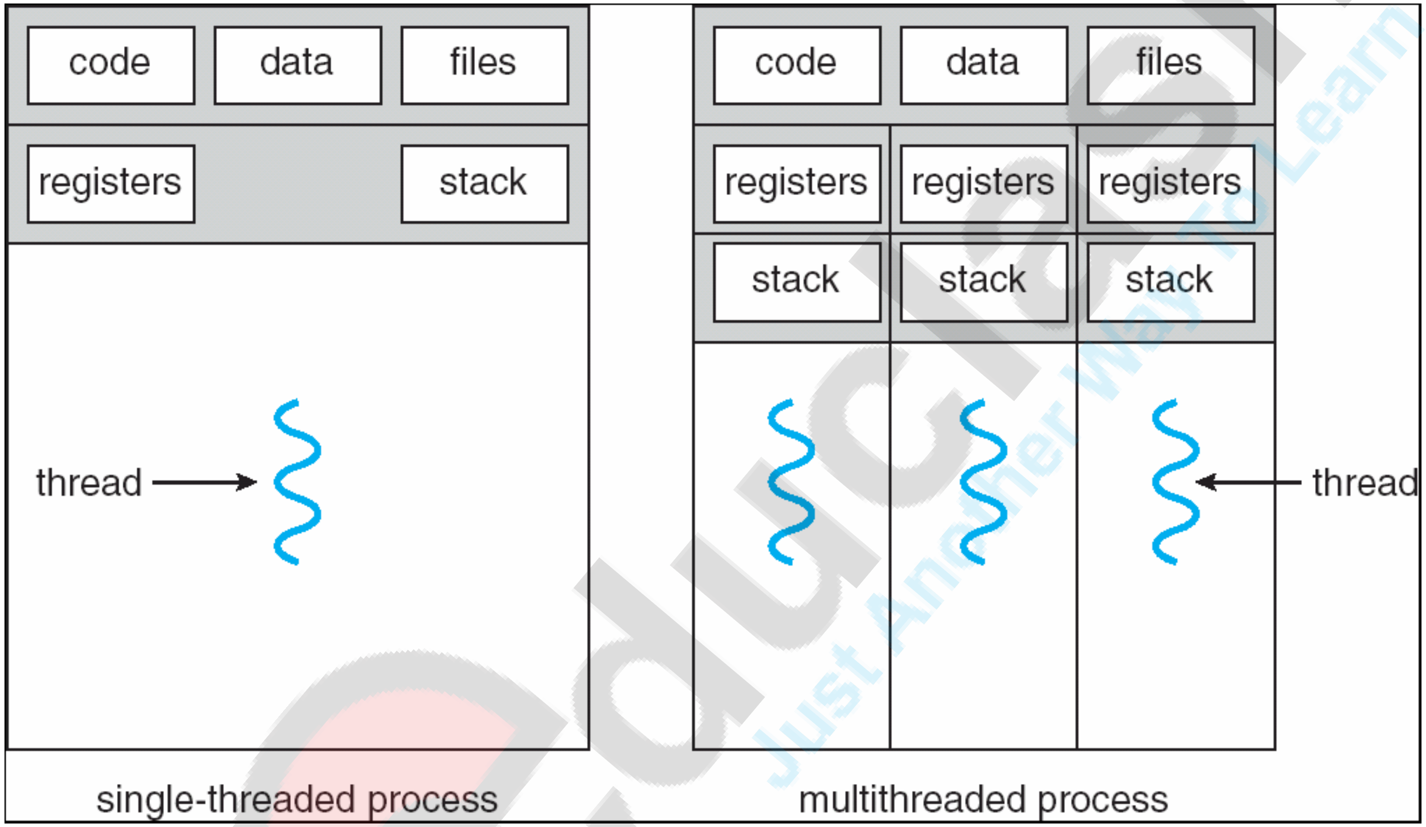


- They also share the same set OS resources, like open files, child processes, semaphores, signals, working environment (current directory, user ID), accounting information and so on
- The uniqueness of threads of a process is that all of them are owned by a single user and a single process and hence very little amount of protection needed
  - There is no protection between the threads of a process

THREADS (CONT'D)

- On a uniprocessor, threads run in quasi-parallel (time sharing), whereas on a shared memory multi-processor, as many threads can run simultaneously as there are processes
- In addition, like traditional processes, threads can create child threads, can block threads for system calls to complete and can change states during their course of execution
- At a particular instance of time a thread can be in any one of the states: running, blocked, ready, or terminated

SINGLE AND MULTITHREADED PROCESSES



- Due to these similarities, threads are often viewed as mini-processes or referred to as lightweight processes and traditional processes are referred to as heavyweight processes

MOTIVATIONS FOR USING THREADS

- In certain cases, a single application may need to run several tasks in parallel and at the same time
- Let us consider the motivation for using multithreaded process instead of multiple single threaded processes for performing some computation activities
- Create a new process for each task
  - Overheads involved in creating a new process are in general considerably greater than those of creating a new thread within a process
    - This mainly because when a new process is created, its address space has to be created from scratch, though part of it is inherited from the parent process

- Switching between threads that are sharing the same address space and other operating system resources is considerably cheaper than switching between processes with their own address space and other operating system resources
- Threads allow parallelism to be combined with sequential execution and blocking of system calls
  - Parallelism improves performance and blocking system calls make programming easier

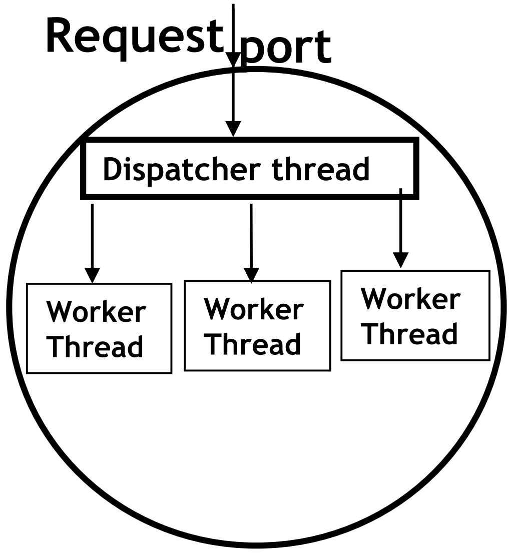
MOTIVATIONS FOR USING THREADS (CONT'D)

- Resource sharing can be achieved more efficiently and naturally between threads of a process than between processes because all threads of a process share the same address space
- From the discussion above, we saw the motivation for using threads in the design of server processes
- Use a single process with multiple threads is a better option
- For eg, when a file is to be replicated on multiple servers, a separate thread can be used to interact with each server
- Client processes that perform lots of distributed operations can also benefit from threads by using a separate thread to monitor each operation

MODELS FOR ORGANIZING THREADS

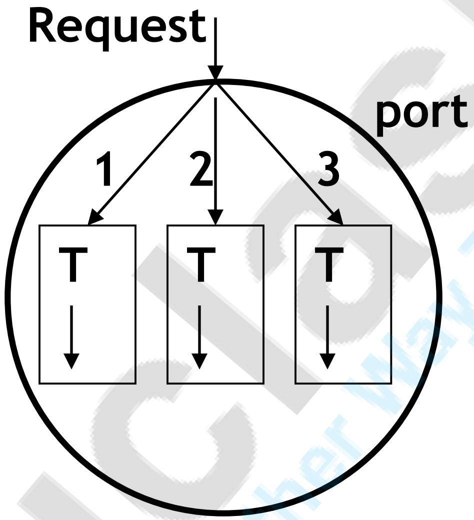
- Depending on an application needs threads of an application can be organized in different ways
- Three commonly used ways to organize the threads are
  - Dispatcher-Worker Model: We have already discussed this model in the client-server application
    - In this model the process consists of a single dispatcher thread and multiple worker threads
    - Dispatcher thread receives the request from the clients and after examining the request, and dispatches it to one the free worker threads for further processing of the request
    - Each worker thread works on different client request
    - Hence multiple client requests can be processed in parallel
    - An example of this model is shown in the next slide

MODELS FOR ORGANIZING THREADS (CONT'D)



Ex. File server

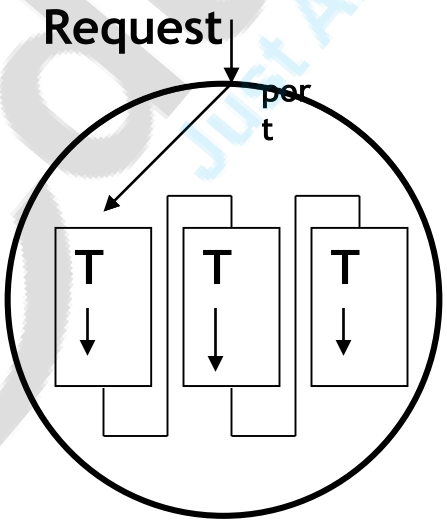
Dispatcher Worker Model



Each type of request being handled by a different thread.

Ex. Interrupt handler threads

Team Model



Output generated by first thread used for processing by second thread.

Ex Producer Consumer applications

Pipeline Model

MODELS FOR ORGANIZING THREADS (CONT'D)

- Team Model
  - In this model all threads are equals in the sense that there is no dispatcher-worker relationship for processing client requests
  - Each thread gets and processes clients' requests on its own
  - This model is often used to implement specialized threads within a process
  - i.e., each thread of the process is specialized in servicing a specific type of request
  - Hence multiple types of requests can be handled by different threads
  - An example is shown in the previous slide for interrupt handler
- Pipeline model
  - This model is useful for applications based on producer-consumer model, in which output data generated by one part of the application is used as input to another part of the application

MODELS FOR ORGANIZING THREADS (CONT'D)

- In this model, the threads of a process are organized as a pipeline so that the output data generated by first thread is used for processing by the second thread, the output of the second thread is used for processing by the third thread, and so on
- The output of the last thread in the pipeline is the final output of the process to which the threads belong
- A typical example is shown in earlier slide

THREADS PACKAGE DESIGN ISSUES

- A system that supports threads should support a set of primitives to its users for threads related operations and this set of primitives is called threads package
- Threads creation
  - Threads can be either static (created at the start of the process and fixed for entire life of the process) or dynamic (as and when required)
  - but the maximum number of threads remain fixed
- Threads termination
  - Destroy itself when it finishes its job or killed from outside by a kill command
  - Threads are terminated only when the process is terminated

THREADS PACKAGE DESIGN ISSUES

- Thread synchronization
  - Since all threads of a process share a common address space, there has to be some mechanism to prevent multiple threads from trying to access the same data simultaneously
  - For example if two threads want to increment a global variable with in a process, there has to be a mechanism to ensure that a thread has exclusive access to that variable for some time
  - Two commonly used mutual exclusion techniques in a threads package are *mutex variables* and *condition variables*

THREAD SCHEDULING

- An important aspect of thread package is how to schedule the threads
- Threads package normally give the users the flexibility to specify scheduling policy to be adopted for their application
- Priority assignment facility
  - In a simple algorithm, threads as scheduled FIFO or Round Robin policy (all threads are equal)
  - It also provides users the flexibility of assigning priorities to the various threads of an application with important ones run on higher priority
  - Priority assignment facility may be Preemptive / Non Preemptive
    - In a non preemptive once CPU is assigned to a thread, it can use the CPU until it blocks, exits or uses up its quantum, even if a higher priority threads wants to start in between
    - In preemptive scheme a higher priority thread always preempts lower priority one

THREAD SCHEDULING (CONT'D)

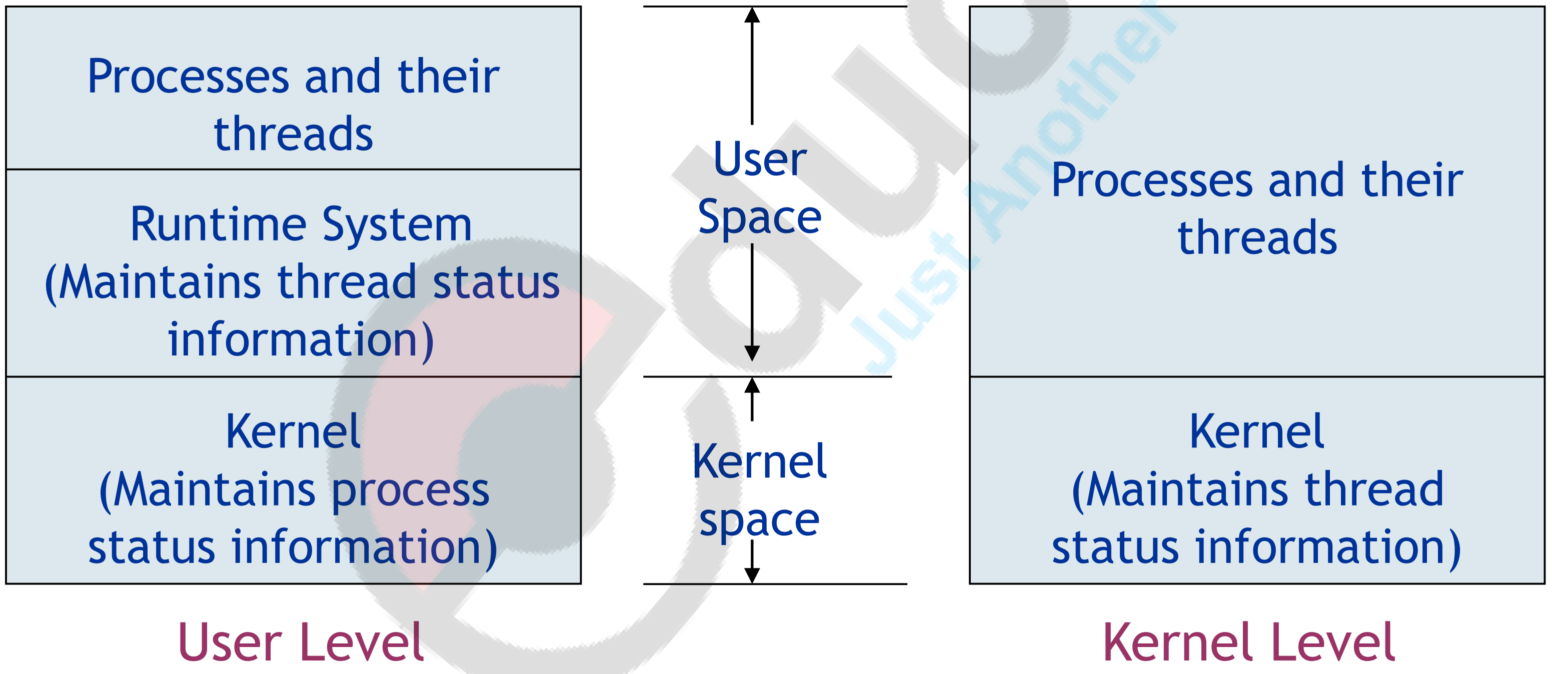
- Flexibility to vary quantum size dynamically: Instead of using fixed length time of Round Robin, vary the size of quantum inversely proportional to total no of threads in the system
  - This algorithm gives good response time to short requests, even on heavily loaded systems, provide high efficiency on lightly loaded systems
- Handoff scheduling
  - It allows a thread to name its successor
  - After sending a message, to another thread, the sending thread can give up the CPU and request that the receiving thread be allowed to run next
  - This can enhance performance if wisely used
- Affinity scheduling
  - Thread is scheduled on the CPU it last ran on in the hope that part of address space is still in CPU's cache

SIGNAL HANDLING

- Signals provide software generated interrupts and exceptions
- Interrupts are externally generated disruptions of a thread or a process, while exceptions are caused by the occurrence of unusual conditions during a thread's execution
- A signal is handled properly by creating a separate exception handler thread in each process, which is responsible for handling all exception conditions occurring in any thread of the process
- The exception handler may clear the exception, causing the victim thread to resume, or terminate the victim thread

IMPLEMENTING A THREADS PACKAGE

- A threads package can be implemented in user space or in the kernel i.e., user-level and kernel-level
  - In user-level, user space consists of a runtime system that is a collection of thread management routines
  - Threads run in the user space on top of the run time system and is managed by it


Processes and their threads
Runtime System (Maintains thread status information)
Kernel (Maintains process status information)
User Level
User Space
Kernel space
Processes and their threads
Kernel (Maintains thread status information)
Kernel Level

IMPLEMENTING A THREADS PACKAGE

- The run time system also maintains a status information table to keep track of the current status of each thread, whose entry consists of registers' value, state, priority and other information of a thread
- All calls of the threads package are implemented as calls to the runtime system procedures that perform the functions corresponding to the calls
- Existence of threads are made totally invisible to Kernel

Implemented in kernel

- In the kernel-level approach, no runtime system is used and the threads are managed by the kernel
- Status information is maintained within the kernel
- All calls that might block a thread are implemented as system calls that trap to the kernel and when a thread blocks, kernel runs another thread
- Fig. on the previous slide illustrates the two approaches for implementing thread package

USER-LEVEL VS. KERNEL-LEVEL

- Advantages of User – level implementation
  - Threads package can be implemented on top of an existing OS that does not support threads; this is not possible in kernel-level, as concept of threads must be incorporated in the design of the kernel of an operating system
  - Users can use customized scheduling algorithms using the two-level scheduling of user-level, while in kernel-level, it is built into the kernel
  - Context switching is faster in user-level as it is done by the runtime system without involving kernel, while in kernel level approach a trap to the kernel is needed for it
  - Scalability of kernel level approach is poor as the status information is maintained by the kernel

USER-LEVEL VS. KERNEL-LEVEL

- Disadvantages of User – level implementation
  - Once the thread is given a CPU to run, as there is no way to interrupt it (no clock) and it continues to run until it voluntarily gives up the CPU, while in kernel approach, clock interrupt occur periodically and kernel can keep track of amount of CPU time used by a thread
  - A means to solve this problem to have the run time system request a clock interrupt after every fixed unit of time to give it the control then the runtime can decide the thread should continue or exit
  - When a thread makes a blocking system call, all threads of its process are stopped & kernel schedules another process to run, while in kernel-level approach, implementation of blocking system calls is straight forward because it traps the kernel, where the present thread is suspended and the kernel starts a new thread

FAULT TOLERANCE

INTRODUCTION

- A fault is a malfunction caused by errors due to design, programming, manufacturing, physical damage, ageing etc.
- Failures can be mild like a file not found or catastrophic like communication crash in an air traffic control system.
- Faults can be classified as:
  - Transient faults: these faults occur suddenly, disappear and may not occur again if the operation is repeated. For eg, fault due to heavy network
  - Intermittent faults: these faults recur often, but may not be periodic in nature. For eg., loose connection to a network switch. Sometimes difficult to diagnose.

  - Permanent Faults: these faults can be easily identified and the component can be replaced. For eg., a software bug.
- Types of faults: fail-stop failure and byzantine failure.
- One of the techniques to handle fault tolerance is redundancy, which can be categorized as:
  - Information Redundancy: extra bits are added with the transmitted data to detect and correct errors. Commonly used methods are Hamming Code, parity check, CRC etc.
  - Time Redundancy: an action performed once is repeated if needed after a specific time period. Such as retransmission of messages.

- Physical redundancy: extra component added to system. For eg., extra processors

This can be further divided into active replication and primary backup methods.

Active Replication – all processors are up all the time in parallel.

Primary Backup – faulty processor replaced with backup processor.

UNIVERSITY QUESTIONS

- What are threads? How are they different from processes? Compare the implementation of a threads package at user level and system level.
- Explain the concept of total freezing in address space transfer mechanism for a process migration facility with proper diagram. Is it better than pretransferring or transfer on reference?
- Give suitable examples for each of the following, a process using multiple threads :-
  - In dispatcher worker model
  - In a pipelined process model
  - In a team model

- Explain fully the concept of preemptive process migration. What are different address space transfer mechanisms used in the process transfer?
- What are threads? How are they different from processes?
- What are the main differences between the Load Balancing and Load Sharing approaches for process scheduling in distributed systems.
- Discuss the various issues of load sharing approach used for better resource utilization.
- What are desirable features of good process migration mechanism.
- What are the different address space transfer mechanisms used in the process transfer?
- What is the need of state information exchange among nodes in distributed system? Explain the various state information exchange policies for load balancing algorithms.
- Discuss relative advantages and disadvantages of preemptive and non-preemptive process migration.