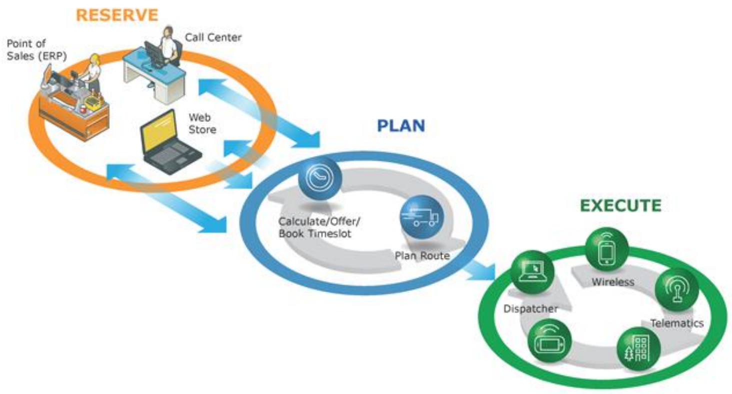
MODULE-4: Resource and Process Management

Vidyalankar Institute of Technology

Accredited A+ by NAAC



Prepared by Prof. Amit K. Nerurkar

Certificate

This is to certify that the e-book titled "RESOURCE AND PROCESS MANAGEMENT" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.

Signature Date: 20-03-2020

Prof. Amit K. Nerurkar

Assistant Professor

Department of Computer Engineering

DISCLAIMER: The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalankar Institute of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.

Module 4 Resource and Process Management

DESIRABLE FEATURES OF GLOBAL SCHEDULING ALGORITHM, TASK ASSIGNMENT APPROACH, LOAD BALANCING APPROACH, LOAD SHARING APPROACH, INTRODUCTION TO PROCESS MANAGEMENT, PROCESS MIGRATION, THREADS, VIRTUALIZATION, CLIENTS, SERVERS, CODE MIGRATION

DESIRABLE FEATURES OF GLOBAL SCHEDULING ALGORITHM

Q.1 Explain Desirable Features of a Good Global Scheduling Algorithm

(A)

- **No A Priori Knowledge about the Processes**
  A good process scheduling algorithm should operate with absolutely no a priori knowledge about the processes to be executed.
- **Dynamic in Nature**
  It is intended that 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. That is, process assignment decisions should be based on the current load of the system and not on some fixed static policy.
- **Quick Decision-Making Capability**
  A good process-scheduling algorithm must make quick decisions about the assignment of processes to processors.
- **Balanced System Performance and Scheduling Overhead**
  Several global scheduling algorithms collect global state information and use this information in making process assignment decisions.
  The general observation is that, as overhead is increased in an attempt to obtain more information regarding the global state of the system, the usefulness of that information is decreased due to both the aging of the information being gathered and the low scheduling frequency as a result of the cost of gathering and processing that information. Hence algorithms that provide near optimal system performance with a minimum of global state information gathering overhead are desirable.
- **Stability**
  It may happen that nodes 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 from 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.
- **Scalability**
  A scheduling algorithm should be capable of handling small as well as large networks. An algorithm that makes scheduling decisions by first inquiring the workload from all the nodes and then selecting the most lightly loaded node as the candidate for receive the process (es) has poor scalability factor. Such an algorithm may work fine for small networks but gets crippled when applied to large networks. This is because the inquirer receives a very large number of replies almost

simultaneously, and the time required to process the reply messages for making a host selection is normally too long.

- **Fault Tolerance**
  A good scheduling algorithm should not be disabled by the crash of one or more nodes of the system. At any instance of time, it should continue functioning for nodes that are up at that time.
- **Fairness of Service**
  While the average quality of service provided is clearly an important performance index, how fairly service is allocated is also a common concern. For example, two users simultaneously initiating equivalent processes expect to receive about the same quality of service.

VIDEO

TASK ASSIGNMENT APPROACH

Q.2 Discuss Task Assignment Approach.

(A) The Basic Idea

i) A process has already been split into pieces called tasks.
ii) The amount of computation required by each task and the speed of each processor are known.
iii) The cost of processing each task on every node of the system is known.
iv) The interprocess communication (IPC) costs between every pair of tasks is known.
v) 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.
vi) Reassignment of the tasks is generally not possible.

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
- A high degree of parallelism
- Efficient utilization of system resources in general

These goals often conflict with each other. For example, while minimizing IPC tends to assign all the tasks of a process to a single node, efficient utilization of system resources.

To illustrate with an example, let us consider the assignment problem of Figure 1. It involves only two task assignment parameters - the task execution cost and the intertask communication cost. This system is made up of six tasks {t_1, t_2, t_3, t_4, t_5, t_6} and two nodes {n_1, n_2}. The intertask communication costs (C_{ij}) and the execution costs (x_{ab}) of the tasks are given in tabular form in Figure 1 (a) and (b) respectively. An infinite cost for a particular task against a particular node in Figure 1 (b) indicates 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. Thus, task t_2 cannot be executed on node n_2 and task t_6 cannot be executed on node n_1. In this model of a distributed computing system, there is no parallelism or multitasking of task execution within a program. Thus the total cost of process execution consists of the total execution cost of the tasks on their assigned nodes plus the intertask communication costs between tasks assigned to different nodes.

Figure 1 (c) shows a serial assignment of the tasks to the two nodes in which the first three tasks are assigned to node n_1 and the remaining three are assigned to node n_2. Observe that this assignment is aimed at minimizing the total execution cost. But if both the execution costs and the communication costs are taken into account, the total cost for this assignment comes out to be 58. Figure 1 (d) shows an optimal assignment of the tasks to the two nodes that minimizes total execution and communication costs. In this case, although the execution cost is more than that of the previous assignment, the total assignment cost is only 38.

Finding an Optimal Assignment

In this approach, an optimal assignment is found by creating a static assignment graph, as shown in Figure 2 In this graph, nodes n_1 and n_2 represent the two nodes (processors) of the distributed system and nodes t_1 through t_6 represent the tasks of the process. The weights of the edges joining pairs of task nodes represent intertask communication costs. The weight on the edge joining a task node to node n_1 represents the execution cost of that task on node n_2 and vice versa.

A cutest in this graph is defined to be a set of edges such that when these edges are removed, the nodes of the graph are partitioned into two disjoint subsets such that the nodes in one subset are reachable from n_1 and the nodes in the other are reachable from n_2. Each task node is reachable from either $n_1$or n_2. No proper subset of a cutest is also a cutest, that is, a cutest is a minimal set. Each cutest corresponds in a one-to-one manner to a task assignment.

Intertask communication 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

(a)

Execution costs		
Tasks	Nodes	
	n_1	n_2
t_1	5	10
t_2	2	∞
t_3	4	4
t_4	6	3
t_5	5	2
t_6	∞	4

(b)

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

(c)

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

(d)

Fig. 1: A task assignment problem example. (a) intertake communication costs ; (b) execution cost of the tasks on the two nodes; (c) serial assignment; (d) optimal assignment.

Serial assignment execution cost (x) = $x_{11} + x_{21} + x_{31} + x_{42} + x_{52} + x_{52}$
$= 5 + 2 + 4 + 3 + 2 + 4 = 20$

Serial assignment communication cost (c) = $c_{14} + c_{15} + c_{16} + c_{24} + c_{25} + c_{26} + c_{34} + c_{35} + c_{36}$
$= 0 + 0 + 12 + 12 + 3 + 0 + 0 + 11 + 0 = 38$

Serial assignment total cost = x + c = 20 + 38 = 58

Optimal assignment execution cost (x) = $x_{11} + x_{21} + x_{31} + x_{41} + x_{51} + x_{62}$
$= 5 + 2 + 4 + 6 + 5 + 4 = 26$

Optimal assignment communication cost (c) = $c_{16} + c_{26} + c_{36} + c_{46} + c_{56}$
$= 12 + 0 + 0 + 0 + 0 = 12$

Optimal assignment total cost = x + c = 26 + 12 = 38

The weight of a cutest is the sum of the weights of the edges in the cutest. It represents the cost of the corresponding task assignment since the weight of cutest sums up the execution and communication costs for that assignment. An optimal assignment may be obtained by finding a minimum-weight cutest. This may be done by the use of network flow algorithms, which are among the class of algorithms with relatively low computational complexity. The bold line in Figure 2 indicates a minimum–weight cutest that corresponds to the optimal assignment of Figure 1 (d). Note that if task $t_1$is assigned to node n_2, then the edge to node n_2 is cut, but this edge carries the cost of executing task t_1 on node n_1. Similarly, other edges cut between task t_1 and other nodes of the graph represent actual communication costs incurred by this assignment for communication between task t_1 and the other corresponding nodes.

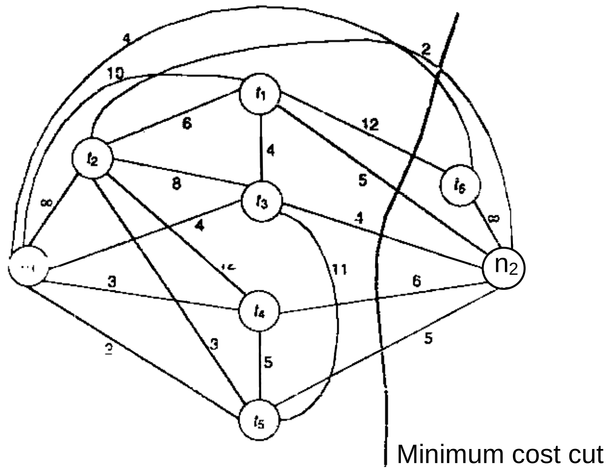


Fig. 2 : Assignment graph for the assignment problem of Fig. 1 with minimum cost cut

LOAD-BALANCING APPROACH

Q.3 What load-balancing approach & how to implement it?

(A) The scheduling algorithms using this approach are known as load-balancing algorithms or load-leveling alogorithms. These algorithms are based on the intuition that, for better resource utilization, it is desirable for the load in a distributed system to be balanced evenly. Thus, a load-balancing algorithm tries to balance the total system load by transparently transferring the workload from heavily loaded nodes to lightly loaded nodes in an attempt to ensure good overall performance relative to some specific metric of system performance. When considering performance from the user point of view, the metric involved is often the response time of the processes. However, when performance is considered from the resource point of view, the metric involved is the total system throughput. In contrast to response time, throughput is concerned with seeing that all users are treated fairly and that all are making progress. Notice that the resource view of maximizing resource utilization is compatible with the desire to maximize system throughput. Thus the basic goal of almost all the load-balancing algorithms is to maximize the total system throughput.

A Taxonomy of Load-Balancing Algorithms

The taxonomy presented here is a hierarchy of the features of load-balancing algorithms. The structure of the taxonomy is shown in figure.

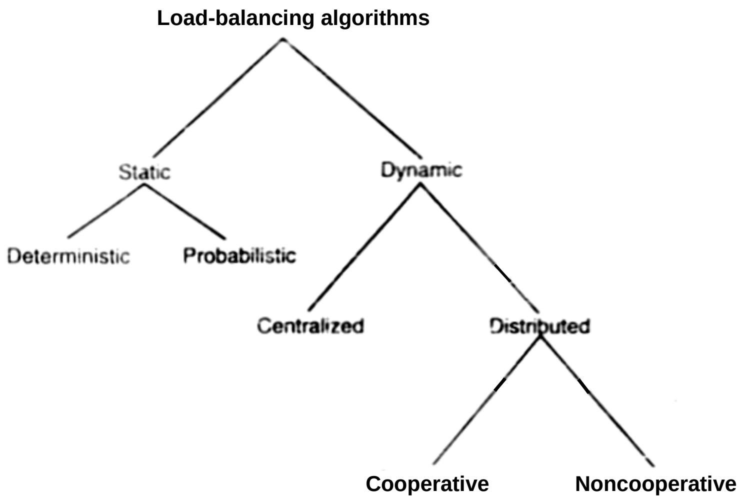


Fig. : A taxonomy of load-balancing algorithms

- **Static versus Dynamic**
  At the highest level, we may distinguish between static and dynamic load-balancing algorithms. Static algorithms use only information about the average behaviour of the system, ignoring the current state of the system. On the other hand, dynamic algorithms react to the system state that changes dynamically.

- **Deterministic versus Probabilistic**

  Static load-balancing algorithms may be either *deterministic* or *probabilistic*. Deterministic algorithms use the information about the properties of the nodes and the characteristics of the processes to be scheduled to deterministically allocate processes to nodes. Notice that the task assignment algorithms basically belong to the category of deterministic static load-balancing algorithms.

  A probabilistic load-balancing algorithm uses information regarding static attributes of the system such as number of nodes, the processing capability of each node, the network topology, and so on, to formulate simple process placement rules. For example, suppose a system has two processors p_1 and p_2 and four terminals t_1, t_2, t_3 and t_4. Then a simple process placement rule can be, assign all processes originating at terminals t_1 and t_2 to processor p_1 and processes originating at terminals t_3 and t_4 to processor p_2. Obviously, such static load-balancing algorithms have limited potential to avoid those states that have unnecessarily poor performance. For example, suppose at the time a particular process originates at terminal t_1 processor p_1 is very heavily loaded and processor p_2 is idle. Certainly, processor p_2 is a better choice for the process in such a situation.

  In general, the deterministic approach is difficult to optimize and costs more to implement. The probabilistic approach is easier to implement but often suffers from having poor performance.

- **Centralized versus Distributed**

  Dynamic scheduling algorithms may be centralized or distributed. In a centralized dynamic scheduling algorithm, the responsibility of scheduling physically resides on a single node. On the other hand, in a distributed dynamic scheduling algorithm, the work involved in making process assignment decisions is physically distributed among the various nodes of the system. In the centralized approach, the system state information is collected at a single node at which all scheduling decisions are made. This node is called the centralized server node.

  A Problem associated with the centralized mechanism is that of reliability. If the centralized server fails, all scheduling in the system would cease. A typical approach to overcome this problem would be to replicate the server on k + 1 nodes if it is to survive k faults (node failures). In this approach, the overhead involved in keeping consistent all the k + 1 replicas of the server may be considerably high.

  In contrast to the centralized scheme, a distributed scheme does not limit the scheduling intelligence to one node. It avoids the bottlenecks of collecting state information at a single node and allows the scheduler to react quickly to dynamic changes in the system state. A distributed dynamic scheduling algorithm is composed is composed of k physically distributed entities e_1, e_2, …, e_k. Each entity is considered a local controller. Each local controller runs asynchronously and concurrently with the others, and each is responsible for making scheduling decisions for the processes of a predetermined set of nodes.

- **Cooperative versus Noncooperative**
  Distributed dynamic scheduling algorithms may be categorized as cooperative and noncooperative. In noncooperative algorithms, individual entities act as autonomous entities and make scheduling decisions independently of the actions of other entities. On the other hand, in cooperative algorithms, the distributed entities cooperate with each other to make scheduling decisions. Hence, cooperative algorithms are more complex and involve larger overhead than noncooperative ones. However, the stability of a cooperative algorithm is better that of a noncooperative algorithm.

Q.4 Discuss Issues in designing load balancing algorithms.

(A) ➤ **Load Estimation Policies**

The first issue in any load-balancing algorithm is to decide on the method to be used to estimate the workload of a particular node. Estimation of the workload of a particular node is a difficult problem for which no completely satisfactory solution exists. A node's workload can be estimated based on some measurable parameters. These parameters could include time-dependent and node-dependent factors such as the following:

- Total number of processes on the node at the time of load estimation.
- Resource demands of these processes.
- Instruction mixes of these processes.
- Architecture and speed of the node's processor.

However, several designers believe that this is an unsuitable measure for such an estimate since the true load could vary widely depending on the remaining service times for those processes.

However, in this case another issue that must be resolved is how to estimate the remaining service time of the processes. Bryant and Finkel have proposed the use of one of the following methods for this purpose:

i) **Memoryless method.** This method assumes that all processes have the same expected remaining service time, independent of the time used so far. The use of this method for remaining service time estimation of a process basically reduces the load estimation method to that of total number of processes.
ii) **Pastrepeats.** This method assumes that the remaining service time of a process is equal to the time used so far by it.
iii) **Distribution method.** If the distribution of service times is known, the associated process's remaining service time is the expected remaining time conditioned by the time already used.

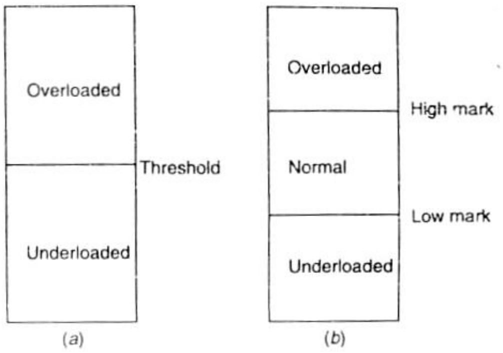
➤ **Process Transfer Policies**

The strategy of load-balancing algorithms is based of transferring some processes from the heavily loaded nodes to the lightly loaded nodes for processing. However, to facilitate this, it is necessary to devise a policy to decide whether a node is lightly or heavily loaded. Most of the load-balancing algorithms use the threshold policy to make this decision. The threshold value of a node is the limiting value of its workload and is used to decide whether a node is lightly or heavily loaded. Thus a new process at a node is accepted locally for processing if the workload of the node is below its threshold value at that time. Otherwise, an attempt is made to transfer the process to a lightly loaded node. The threshold value of a node may be determined by any of the following methods:

i) **Static policy.** In this method each node has a predefined threshold value depending on its processing capability. This threshold value does not vary with the dynamic changes in workload at local or remote nodes.
ii) **Dynamic policy.** In this method, the threshold value of a node (n_1) is calculated as a product of the average workload of all the nodes and a predefined constant (C_i). For each node n_i, the value of c_i depends on the processing capability of node n_i relative to the processing capability of all other nodes.

Most load-balancing algorithms use a single threshold and thus only have overloaded and underloaded regions Figure 1(a). In this single-threshold policy, a node accepts new processes (either local or remote) if its load is below the threshold value and attempts to transfer local processes and rejects remote execution requests if its load is above the threshold value.

Fig. 1 : The load regions of (a) single-threshold policy and (b) double-threshold policy.



A double-threshold policy called the high-low policy. As shown in Figure 4 (b), the higher-low policy uses two threshold values called high mark and low mark, which divide the space of possible load states of a node into the following three regions:

- Overloaded–above the high-mark and low-mark values.
- Normal–above the low-mark value and below the high-mark value.
- Underloaded–below both values.

A node's load state switches dynamically from one region to another, as shown in Figure 2. Now depending on the current load status of a node, the decision to transfer a local process or to accept a remote process is based on the following policies.

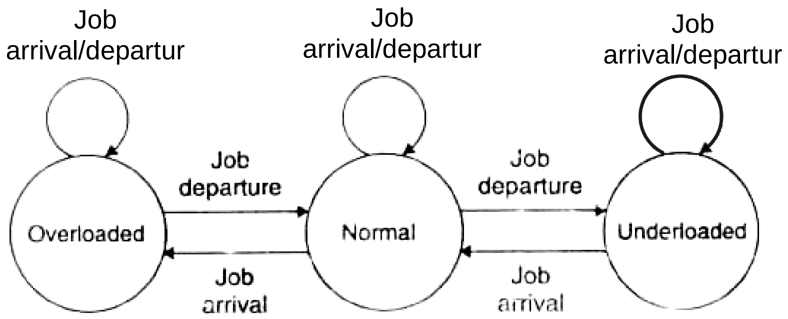


Fig. 2 : State transition diagram of the load of a node in case of double-threshold policy.

- When the load of the node is in the overloaded region, new local processes are sent to be run remotely and requests to accept remote processes are rejected.
- When the load of the node is in the normal region, new local processes run locally and requests to accept remote processes are rejected.
- When the load of the node is in the underloaded region, new local processes run locally and requests to accept remote processes are accepted.

➤ **Location Policies**

Once a decision has been made through the transfer policy to transfer a process from a node, the next step is to select the destination node for that process's execution. This selection is made by the location policy of a scheduling algorithm. The main location policies proposed in the literature are described below.

i) Threshold. In this method, a destination node is selected at random and a check is made to determine whether the transfer of the process to that node would place it in a state that prohibits the node to accept remote processes. If not, the process is transferred to the selected node, which must execute the process regardless of its state when the process actually arrives. On the other hand, if the check indicates that the selected node is in a state that prohibits it to accept remote processes, another node is selected at random and probed in the same manner. This continues until either a suitable destination node is found or the number of probes exceeds a static probe limit L_P.

ii) Shortest. In this method, L_P distinct nodes are chosen at random, and each is polled in turn to determine its load. The process is transferred to the node having the minimum load value, unless that node's load is such that it prohibits the node to accept remote processes. If none of the polled nodes can accept the process, it is executed at its originating node. If a destination node is found and the process is transferred there, the destination node must execute the process regardless of its state at the time the process actually arrives. A simple improvement to the basic shortest policy is to discontinue probing whenever a node with zero load is encountered, since that node is guaranteed to be an acceptable destination.

iii) **Bidding.** In this method, the system is turned into a distributed computational economy with buyers and sellers of services. Each node in the network is responsible for two roles with respect to the bidding process: manager and contractor. The manager represents a node having a process in need of a location to execute, and the contractor represents a node that is able to accept remote processes. Note that a single node takes on both these roles and no nodes are strictly managers or contractors alone. To select a node for its process, the manager broadcasts a request-for-bids message to all other nodes in the system. Upon receiving this message, the contractor nodes return bids to the manager node. The bids contain the quoted prices, which vary based on the processing capability, memory size, resource availability, and so on, of the contractor nodes. Of the bids received from the contractor nodes, the manager node chooses the best bid. The best bid for a manager's request may mean the cheapest, fastest or best price-performance, depending on the application for which the request was made. Once the best bid is determined, the process is transferred from the manager node to the winning contractor node. But it is possible that a contractor node may simultaneously win many bids from many other manager nodes and thus become overloaded. To prevent this situation when the best bid is selected, a message is sent to the owner of that bid. At that point the bidder may choose to accept or reject that process. A message is sent back to the concerned manager node informing it as to whether the process has been accepted or rejected. A contractor node may reject a winning bid because of changes in its state between the time the bid was made and the time it was notified that it won the bid. If the bid is rejected, the bidding procedure is started all over again.

iv) **Pairing.** In this method, two nodes that differ greatly in load are temporally paired with each other, and load-balancing operation is carried out between the nodes belonging to the same pair by migrating one or more processes from the more heavily loaded node to the other node. Several node pairs may exist simultaneously in the system. A node only tries to find a partner if it has at least two processes; otherwise migration from this node is never reasonable. However, every node is willing to respond favourably to a pairing request.

➤ **State Information Exchange Policies**

The proposed load-balancing algorithms use one of the following policies for this purpose.

i) **Periodic Broadcast.** In this method each node broadcasts its state information after the elapse of every t units of time. Obviously this method is not good because it generates heavy network traffic.

ii) **Broadcast When State Changes.** A node broadcasts its state information only when the state of the node changes.

A further improvement in this method can be obtained by observing that it is not necessary to report every small change in the state of a node to all other nodes because a node can participate in the load-balancing process only when it is either underloaded or overloaded. Therefore in the refined method, a node broadcasts its state information only when its state switches from the normal load region to either the underloaded region or the overloaded region.

iii) **On-Demand Exchange.** A node needs to know about the state of other nodes only when it is either underloaded or overloaded. The method of on-demand exchange of state information is based on this observation. In this method a node broadcast a StateInformationRequest message when its state switches from the normal load region to either the underloaded region or the overloaded region. On receiving this message, other nodes send their current state to the requesting node.

iv) **Exchange by Polling.** All the methods described above the use the method of broadcasting due to which their scalability is poor. The polling mechanism overcomes this limitation by avoiding the use of broadcast protocol. This method 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. Rather, when a node needs the coopertion of some other node for load balancing, it can search for a suitable partner by randomly polling the other nodes one by one. Therefore state information is exchanged only between the polling node and the polled nodes. The polling process stops either when a suitable partner is found or a predefined poll limit is reached.

v) **Priority Assignment Policies**

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 at a particular node. One of the following priority assignment rules may be used for this purpose:

- Selfish. Local processes are given higher priority than remote processes.
- Altruistic. Remote processes are given higher priority than local processes.
- Intermediate. The priority of processes depends on the number of local processes and the number of remote processes at the concerned node. If the number of local process is greater than or equal to the number of remote processes. Local processes are given higher priority than remote processes. Otherwise, remote processes are given higher priority than local processes.

➤ **Migration-Limiting Policies**

Another important policy to be used by a distributed operating system that supports process migration is to decide about the total number of times a process should be allowed to migrate. One of the following two policies may be used for this purpose.

i) **Uncontrolled.** In this case, a remote process arriving at a node is treated just as a process originating at the node. Therefore, under this policy, a process may be migrated any number of times. This policy has the unfortunate property of causing instability.

ii) **Controlled.** To overcome the instability problem of the uncontrolled policy, most systems treat remote processes different from local processes and use a migration count parameter to fix a limit on the number of times that a process may migrate.

Video

LOAD-SHARING APPROACH

Q.5 What is Load-Sharing Approach & how to implement it ?

(A) Several researchers believe that load balancing, with its implication of attempting to equalize workload on all the nodes of the system, is not an appropriate objective. This is because the overload involved in gathering state information to achieve this objective is normally very large, especially in distributed systems having a large number of nodes. Moreover, load balancing in the strictest sense is not achievable because the number of processes in a node is always fluctuating and the temporal unbalance among the nodes exists at every moment, even if the static (average) load is perfectly balanced. In fact, for the proper utilization of the resources of a distributed system, it is not required to balance to the load on all the nodes. Rather, it is necessary and sufficient to prevent the nodes from being idle while some other nodes have more than two processes. Therefore this rectification is often called dynamic load sharing instead of dynamic load balancing.

1. **Load Estimation Policies :** Since load-sharing algorithms simply attempt to ensure that no node is idle while processes wait for service at some other node, it is sufficient to know whether a node is busy or idle. Thus load-sharing algorithms normally employ the simplest load estimation policy of routing the total number of processes on a node.

2. **Process Transfer Policies :** Since load sharing algorithms are normally interested only in the busy or idle states of a node most of them employ the all-or-nothing strategy. This strategy uses the single-threshold policy with the threshold value of all the nodes fixed at 1. That is, a node becomes a candidate for accepting a remote process only when it has no process, and a node becomes a candidate for transferring a process as soon as it has more than one process.

3. **Location Policies :** The location policies are of the following types:
   i) **Sender-Initiated Location Policy.** In the sender-initiated location policy, heavily loaded nodes search for lightly loaded nodes to which work may be transferred. That is in this method, when a node's load becomes more than the threshold value, it either broadcasts a message or randomly probes the other nodes one by one to find a lightly loaded node that can accept one or more of its processes. A node is a viable candidate for receiving a process from the sender node only if the transfer of the process to that node only if the transfer of the process to that node would not increase the receiver node's load above its threshold value. In the broadcast method, the presence or absence of a suitable receiver node is known as soon as the sender node receives reply messages from the other nodes.

   ii) **Receiver-Initiated Location Policy.** In the receiver-initiated location policy, lightly loaded nodes search for heavily loaded nodes from which work may be transferred. That is, in this method, when a node's load falls below the threshold value, it either broadcasts a message indicating its willingness to receive processes for executing or randomly probes the other nodes one by one to find a heavily loaded node that can send one or more of its processes. A node is a viable candidate for sending one of its processes only if the transfer of the process from that node would not reduce its load below the threshold value. In the broadcast method, a suitable node is found as soon as the receiver node receives reply messages from the other nodes.

iii) State Information Exchange Policies

In load-sharing algorithms, a node normally exchanges state information with other nodes only when its state changes. The two commonly used policies for this purpose are described below.

- **Broadcast When State Changes.** In this method, a node broadcasts a StateInformationRequest message when it becomes either underloaded or overloaded. Obviously, in the sender-initiated policy, a node broadcasts this message only when it becomes overloaded, and in the receiver-initiated policy, a node broadcasts this message only when it becomes overloaded, and in the receiver-initiated policy, this message is broadcast by a node when it becomes underloaded. In receiverr-initiated policies that use a fixed threshold value of 1, this method of state information exchange is called broadcast-when-idle policy.
- **Poll When State Changes.** Since a mechanism that uses broadcast protocol is unsuitable for large networks, the polling mechanism is normally used in such systems. In this method, when a node's state changes, it does not exchange state information with all other nodes but randomly polls the other nodes one by one and exchanges state information with the polled nodes. The state exchange process stops either when a suitable node for sharing load with the probing node has been found or the number of probes has reached the probe limit.

PROCESS MIGRATION

Q.6 What are Desirable Features of a Good Process Migration ?

(A) A good process migration mechanism must possess transparency

- transparency
- minimal residue dependencies
- robustness
- minimal interferences
- efficiency
- communication between coprocesses

Process Migration Mechanisms:

Migration of a process is a complex activity. The four major subactivities involved in process migration are as follows:

1. Freezing the process on its source node and restarting it on its destination node.
2. Transferring the process's address space from its source node to its destination node.
3. Forwarding messages meant for the migrant process.
4. Handling communication between cooperating processes that have been separated as a result of process migration.

1. **Mechanisms for freezing and restarting a process:**
   At some point during migration, the process is frozen on its source node, its state information is transferred to its destination node, and the process is restarted on its destination node using this state information.
   Some general issues involved in these operations are
   1. Immediate and Delayed Blocking of the process.
   2. Fast and Slow I/O Operations.
   3. Information about open files.
   4. Reinstating the process on its Destination Node.
2. **Address Space Transfer Mechanisms:**
   A process consists of the program being executed, along with the programs data, stock and state.
   Thus, the migration of a process involves the transfer of the following types of information from the source node to the destination node.

Process's State:

Which consists of the execution status (contents of regs),

- Scheduling information.
- Information about main member being used by the process
- I / O Status
- Process identifier
- Process's user and group identifiers
- Information about the files opened by the process and so on.

Process's address space

- code
- data
- and stack of the program.

In all the systems, the migrant process's execution is stopped while transferring its, state information.

However, due to flexibility in transforming the process's address space at any time after the migration decision is made, the existing distributed systems use one of the following address space transfer mechanisms.

- total freezing
- pretransferring
- or transfer on reference

Total freezing: In this method, a process's execution is stopped while its address space is being transferred

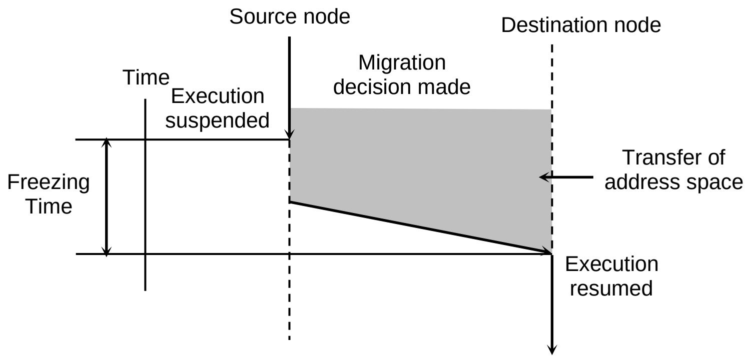


Fig. 1 : Total freezing mechanism.

Pretransferring: In this method, the address space is transferred while the process is still running on the source node.

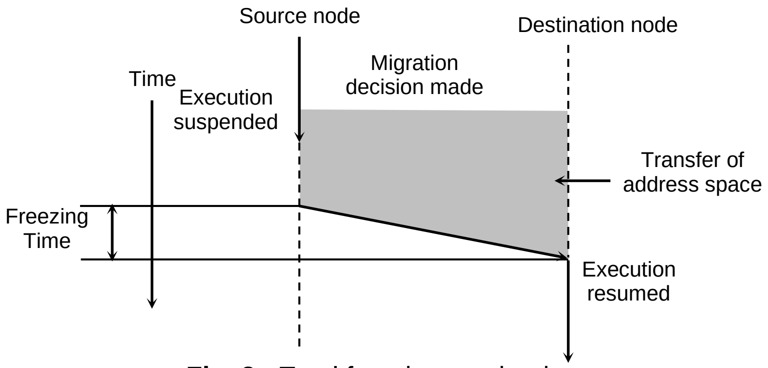


Fig. 2 : Total freezing mechanism.

Transfer on reference: This method is based on the assumption that processes tend to use only a relatively small part of their address spaces while executing.

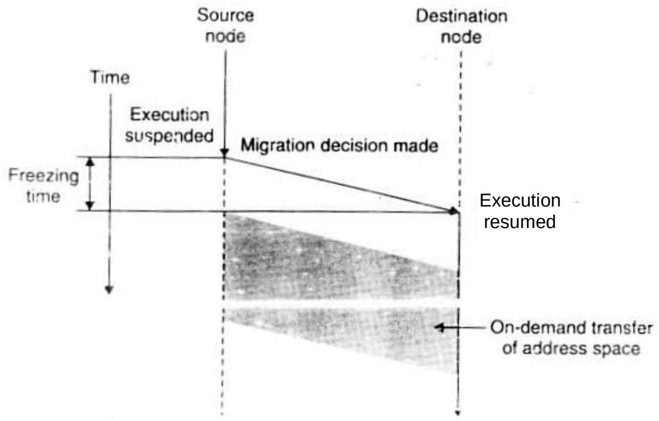


Fig. 3 : Transfer on–reference mechanism.

3. **Message–Forwarding Mechanisms:** The different mechanisms used for message forwarding in existing distributed systems are described below.
   i) Mechanism of Resending the Message
   ii) Origin site Mechanism
   iii) Link Traversal Mechanism
   iv) Link update Mechanism.

   i) Mechanism of Resending the Message: This mechanism is used in the V-system and Amoeba to handle messages of all types.
   ii) Origin site Mechanism: This method is used in Alx's TCF (Transparent Computing Facility) and Sprite.
   iii) Link Traversal Mechanism: In DEMOS / MP to redirect the message, a message queue for the migrant process is created on its source node. All the messages are placed in this message queue. All messages in the queue are sent to the destination node as a part of the migration procedure.
   iv) Link update Mechanism: In this processes communicate via location-independent links.
4. **Mechanisms for Handling Coprocesses:** Two different mechanisms are used.
   i) Disallowing Separation of Coprocesses.
   ii) Home Node or origin site concept.

   i) Disallowing Separation of Coprocesses: The easiest method of handling communication between coprocesses is to disallow their separation.
   ii) Home Node or origin site concept: Sprite uses its home node concept for communication between a process and its subprocess when the two are running on different nodes.

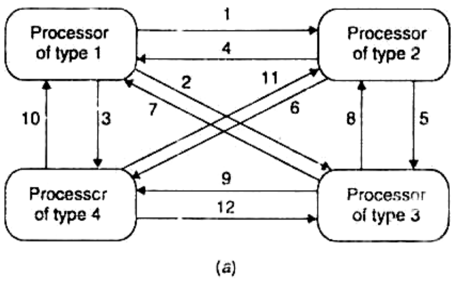
Advantages of Process Migration

1. Reducing average response time of processes.
2. Speeding up individual jobs.
3. Gaining higher throughput.
4. Utilizing resources effectively.
5. Reducing network traffic.
6. Improving system reliability.
7. Improving system security.

Video

Q.7 Discuss Process Migration in Heterogeneous Systems.

(A) When a process is migrated in a homogeneous environment, the interpretation of data is consistent on both the source and the destination nodes. Therefore, the question of data translation does not arise. However, when a process is migrated in a heterogeneous environment, all the concerned data must translated from the source CPU format to the destination CPU format before it can be executed on the destination node. If the system consists of two CPU types, each processor must be able to convert the data from the foreign processor type into its own format. If a third CPU is added, each processor must be able to translate between its own representation and that of the other two processors. Hence, in general, a heterogeneous system having n CPU types must have n(n–1) pieces of translation software in order to support the facility of migrating a process from any node to any other node.



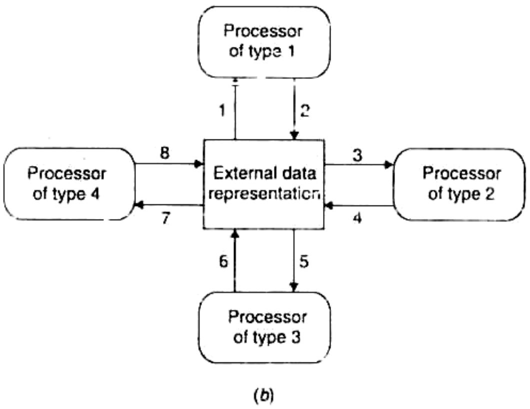


Fig. : (a) Example illustrating the need for 12 pieces of translation software required in a heterogeneous system having 4 types of processors. (b) Example illustrating the need for only 8 pieces of translation software in a heterogeneous system having 4 types of processors when the external data representation mechanism is used.

An example for four processor types is shown in Figure (a).This is undesirable, as adding a new CPU type becomes a more difficult task over time.

Maguire and Smith proposed the use of the external data-representation mechanism for reducing the software complexity of this translation process. In this mechanism, a standard representation is used for the transport of data and each processor needs only to be able to convert data to and from the standard form. This bounds the complexity of the translation software. An example for four processor types is shown in Figure (b).The process of converting from a particular machine representation to external data representation format is called serializing, and the reverse process is called deserializing.

The standard data representation format is called *external data representation*, and its designer must successfully handle the problem of different representations for data such as characters, integers, and floating-point numbers. Of these, the handling of floating-point numbers needs special precautions.

THREADS

Q.8 Explain Models for Organizing Threads.

(A)

1. Dispatcher-workers model. In this model, the process consists of a single dispatcher thread and multiple worker threads. The dispatcher thread accepts requests from clients and, after examining the request, dispatches the request to one of the free worker threads for further processing of the request. Each worker thread works on a different client request. Therefore multiple client requests can be processed in parallel. An example of this model is shown in Figure (a).

2. Team model. In this mode, all threads behave as equal in the sense that there is no dispatcher-worker relationship for processing client's requests. Each thread gets and processes client's requests on its own. This mode is often used for implementing specialized threads within a process. That is, each thread of the process is specialized in servicing a specific a specific type of request. Therefore, multiple types of requests can be simultaneously handled by the process. An example of this model is shown in Figure (b).

3. Pipeline model. This model is useful for applications based on the producer-consumer model, in which the output data generated by one part of the application is used as input for another part of the application. In this model, the threads of a process are organized as a pipeline so that the output data generated by the 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 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. An example of this model is shown in Figure (c).

Video

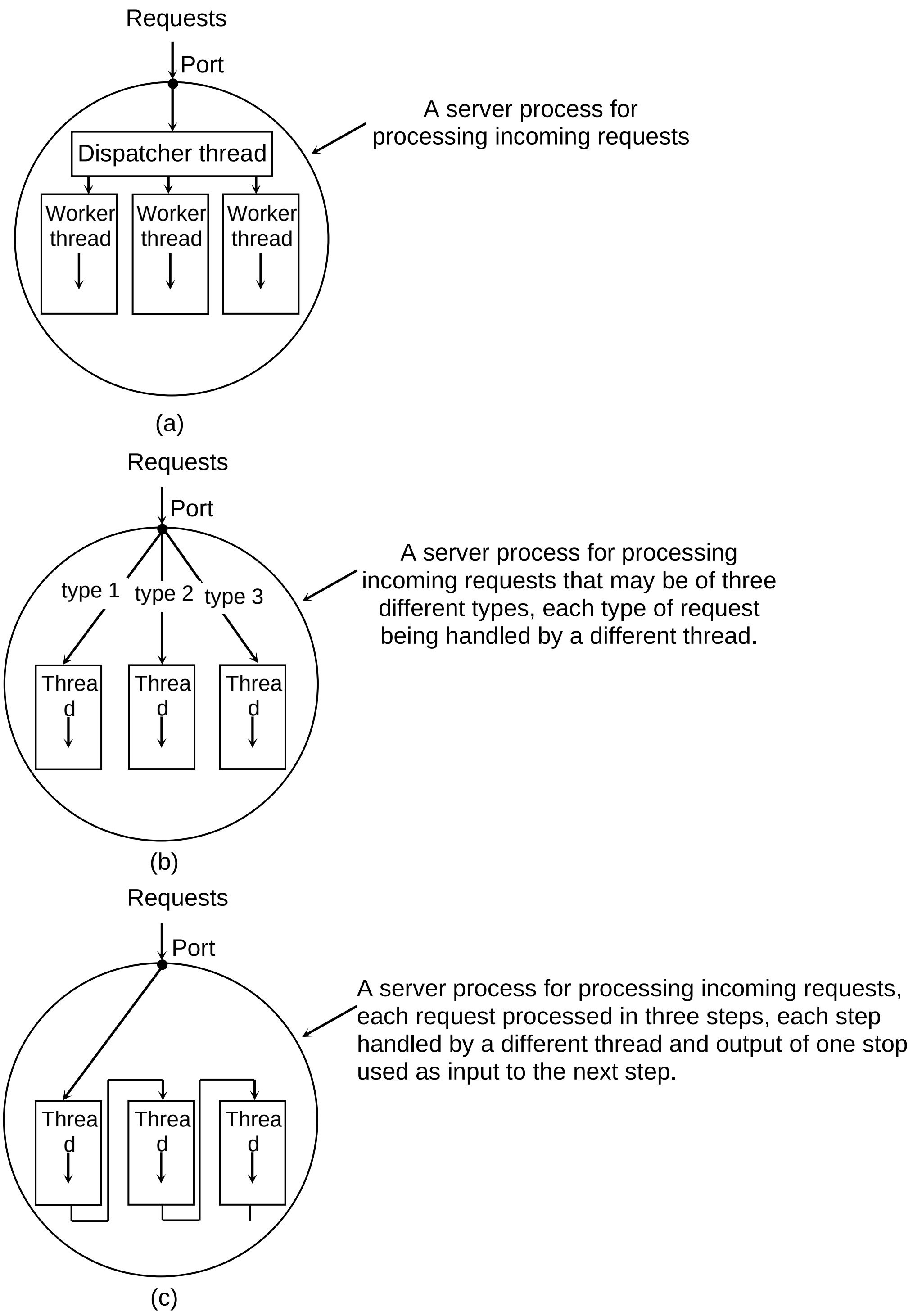

Requests
Port
Dispatcher thread
Worker thread
Worker thread
Worker thread
A server process for processing incoming requests
(a)
Requests
Port
type 1
type 2
type 3
Thread
Thread
Thread
A server process for processing incoming requests that may be of three different types, each type of request being handled by a different thread.
(b)
Requests
Port
Thread
Thread
Thread
A server process for processing incoming requests, each request processed in three steps, each step handled by a different thread and output of one stop used as input to the next step.
(c)

Fig. : Models for organizing threads: (a) dispatcher-workers model (b) team model (c) pipeline model

Q.9 Explain Issues in Designing a Threads Package

(A)

i) Threads Creation

Threads can be created either statically or dynamically. In the static approach, the number of threads of a process remains fixed for its entire lifetime, while in the dynamic approach, the number of threads of a process keeps changing dynamically.

ii) Threads Termination

Termination of threads is performed in a manner similar to the termination of conventional processes. That is, a thread may either destroy itself when it finishes its job by making an exit call or be killed from outside by using the kill command and specifying the thread identifier as its parameter. In many cases, threads are never terminated.

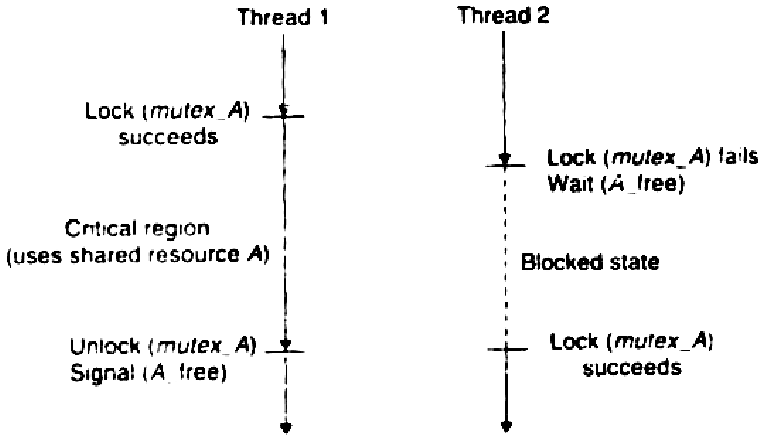
Threads Synchronization

Since all the threads of process share a common address space, some mechanism must be used to prevent multiple threads from trying to access the same data simultaneously. Each example supposes two threads of a process need to increment the same global variable within the process. For this to occur safely, each thread must ensure that it has exclusive access for this variable for some period of time. A segment of code in which a thread may be accessing some shared variable is called a critical region. To prevent multiple threads from accessing the same data simultaneously, it is sufficient to ensure that when one thread is executing in a critical region, no other thread is allowed to execute in a critical region in which the same data is accessed by the threads must be mutually exclusive in time. Two commonly used mutual exclusion techniques in a threads package are mutex variables and condition variables.

1. A mutex variable is like a binary semaphore that is always in one of two states, locked or unlocked. A thread that wants to execute in a critical region performs a lock operation on the corresponding mutex variable. If the mutex variable is in the unlocked state, the lock operation succeeds and the state of the mutex variable changes from unlocked to locked in a single atomic action. After this, the thread can execute in the critical region. However, if the mutex variable is already locked, depending on the implementation, the lock operation is handled in one of the following ways:
   i) The thread is blocked and entered in a queue of threads waiting on the mutex variable.
   ii) A status code indicating failure is returned to the thread. In this case, the thread has the flexibility to continue with some other job. However, to enter the critical region, the thread has to keep retrying to lock the mutex variable until it succeeds.

When a thread finishes executing in its critical region, it performs an unlock operation on the corresponding mutex variable. At this time, if the blocking method is used and if one or more threads are blocked waiting on the mutex variables, one of them is given the lock and its state is changed from blocked to running while others continue to wait.

Mutex variables are simple to implement. For more general synchronization requirements condition variables are used. Wait and signal are two operations normally provided for a condition variable. When a thread performs a wait operations on a condition variable, the associated mutex variable is unlocked, and the thread is blocked until a signal operation is performed by some other thread on the condition variable, indicating that the event being waited for may have occurred. When a thread performs a signal operation on the condition variable, the mutex variable is locked, and the thread that was blocked waiting on the condition variable starts executing in the critical region. Figure illustrates the use of mutex variable and condition variable for synchronizing threads.



Mutex _A is a multex variable for exclusive use of shared resource A. A_free is a condition variable for resource A to become free.

Fig . : Use of mutex variable and condition variable for synchronizing threads.

Threads Scheduling

Another important issue in the design of a threads package is how to schedule the threads. Threads packages often provide calls to give the users the flexibility to specity the scheduling policy to be used for their applications. With this facility, an application programmer can use the heuristics of the problem to decide the most effective manner for scheduling. Some of the special features for threads scheduling that may be supported by a threads package are as follows:

i) Priority assignment facility. In a simple scheduling algorithm, threads are scheduled on a first-in, first-out basis or the round-robin policy is used to timeshare the CPU cycles among the threads on a quantum-by-quantum basis, with an threads treated as equals by the scheduling algorithm. However, a threads-scheduling scheme may provide the flexibility to the application programmers to assign priorities to the various threads of an application in order to ensure that important ones can be run on a higher priority basis.

ii) Flexibility to vary quantum size dynamically. A Simple round-robin scheduling scheme assigns a fixed- length quantum to timeshare the CPU cycles among the threads. However, a fixed-length quantum is not appropriate on a multiprocessor system because there may be fewer runnable threads than there are available processors. Therefore, instead of using a fixed-length quantum, a scheduling scheme may vary the size of the time quantum inversely with the total number of threads in the system.

iii) Handoff scheduling. A handoff scheduling scheme allows a thread to name its successor if it wants to. For example, 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. Therefore, this scheme provides the flexibility to bypass the queue of runnable threads and directly switch the CPU to the thread specified by the currently running thread. Handoff scheduling can enhance performance if it is wisely used.

iv) Affinity scheduling. Another scheduling policy that may be used for better performance on a multiprocessor system is affinity scheduling. In this scheme, a thread is scheduled on the CPU it last ran on in hopes that part of its address space is still in that CPU's cache.

Q.10 Discuss Implementing a Threads Package.

(A) A threads package can be implemented either in user space or in the kernel. In the description below, the two approaches are referred to as user-level and kernel-level, respectively. In the user-level approach, the user space consists of a runtime system that is collection of threads management routines. Threads run in the user space on top of the runtime system and are managed by it. The runtime system also maintains a status information table to keep track of the current status of each thread. This table has one entry per thread. An entry of this table has fields for registers values, state, priority, and other information of a thread. All calls of the threads package are implemented as calls to the runtime system procedures that performs the functions corresponding to the calls. These procedures also perform thread switching if the thread that made the calls has to be suspended during the call. That is, two-level scheduling is performed in this approach. The scheduler in the kernel allocates quanta to heavyweight processes, and the scheduler of the runtime system divides a quantum allocated to a process among the threads of that process. In this manner, the existence of threads is made totally invisible to the kernel. The kernel functions in a manner similar to an ordinary kernel that manages only single-threaded, heavyweight processes. This approach is used by the SunOS 4.1 Lightweight Processes package.

On the other hand, in the kernel-level approach, no runtime system is used and the threads are managed by the kernel. Therefore, the threads status information table is maintained within the kernel. All calls that might block a thread are implemented as system calls that trap to the kernel. When a thread blocks, the kernel selects another thread to be run. The selected thread may belong to either the same process as that of the previously running thread or a different process. Hence, the existence of threads is known to the kernel, and single-level scheduling is used in this approach.

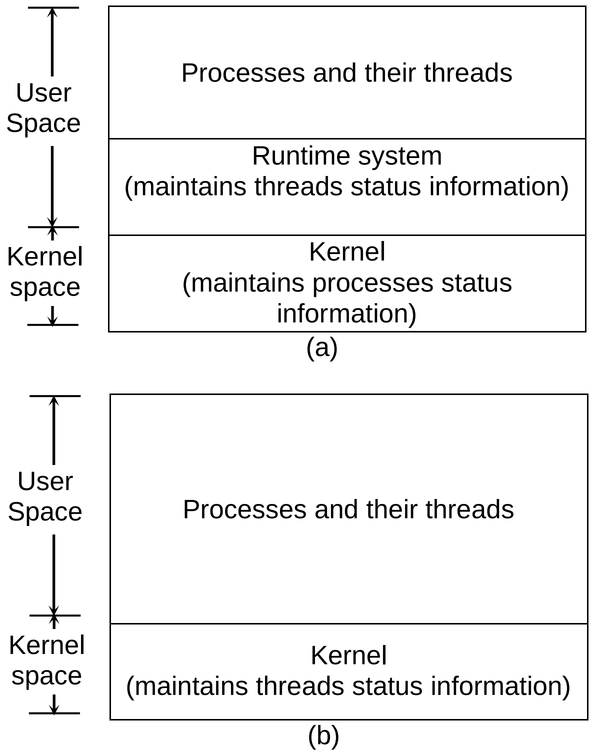

User Space
Processes and their threads
Runtime system (maintains threads status information)
Kernel space
Kernel (maintains processes status information)
(a)
User Space
Processes and their threads
Kernel space
Kernel (maintains threads status information)
(b)

Fig. : Approaches for implementing a threads package : (a) user level; (b) Kernel level.

Figure Illustrates the two approaches for implementing a threads package. The relative advantages and disadvantages of the approaches are as follows.

i) The most important advantage of the user-level approach is that a threads package can be implemented on top of an existing operating system that does not support threads. This is not possible in the kernel-level approach because in this approach the concept of threads must be incorporated in the design of the kernel of an operating system.

ii) In the user-level approach, due to the use of two-level scheduling, users have the flexibility to use their own customized algorithm to schedule the threads of a process. Therefore, depending on the needs of an application, a user can design and use the most appropriate scheduling algorithm for the application. This is not possible in the kernel-level approach because a single-level scheduler is used that is built into the kernel. Therefore, users only have the flexibility to specify through the system call parameters the priorities to be assigned to the various threads of process and to select an existing algorithm form a set of already implemented scheduling algorithms.

iii) Switching the context from one thread to another is faster in the user-level approach than in the kernel-level approach. This is because in the former approach context switching is performed by the runtime system, while in the latter approach a trap to the kernel is needed for it.

iv) In the kenel-level approach, the status information table for threads is maintained within the kernel. Due to this, the scalability of the kernel-level approach is poor as compared to the user-level approach.

v) A serious drawback associated with the user-level approach is that with this approach the use of round-robin scheduling policy to timeshare the CPU cycles among the threads on a quantum-by-quantum basis is not possible [Tanebaum 1995]. This is due to the lack of clock interrupts within a single process. Therefore, once a thread is given the CPU to run, there is no way to interrupt it, and it voluntarily gives up the CPU. This is not the case with the kernel-level approach, in which clock interrupts occur periodically, and the kernel can keep track of the amount of CPU time consumed by a thread. When a thread finishes using its allocated quantum, it can be interrupted by the kernel, and the CPU can be given to another thread.

   A crude way to solve this problem is to have the runtime system request a clock interrupt after every fixed unit of time (say every half a second) to give it control. When the runtime system gets control, the second scheduler can decide if the thread should continue running or the CPU should now be allocated to another thread.

vi) Another drawback of the user level approach is associated with the implementation of blocking system calls. In the kernel-level approach, implementation of blocking system calls is straightforward because when a thread makes such a call, it traps to the kernel, where it is suspended, and the kernel starts a new thread. However, in the user-level approach, a thread should not be allowed to make blocking system calls directly. This is because if a thread directly makes a blocking system call. All threads of its process will be stopped, and the kernel will schedule another process to run. Therefore, the basic purpose of using threads will be lost.

   A commonly used approach to overcome this problem is to use jacket routines. A jacket routine contains extra code before each blocking system call to first make a check to ensure if the call will cause a trap to the kernel. The call is made only if it is safe (will not cause a trap to the kernel); otherwise the thread is suspended and another thread is scheduled to run. Checking the safety condition and making the actual call must be done atomically.

Video

CODE MIGRATION

Q.11 Explain Code Migration.

(A) However, there are situations in which passing programs, sometimes even while they are being executed, simplifies the design of a distributed system. We start by considering different approaches to code migration, followed by a discussion on how to deal with the local resources that a migrating program uses. A particularly hard problem is migrating code in heterogeneous systems.

Approaches to Code Migration

Before taking a look at the different forms of code migration, let us first consider why it may be useful to migrate code.

Reasons for Migrating Code

Code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another. Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so. That reason has always been performance. The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines. Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

Consider, for example, a client-server system in which the server manages a huge database. If a client application needs to do many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

Support for code migration can also help improve performance by exploiting parallelism, but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speed-up compared to using just a single program instance.

However, if code can move between different machines, it becomes possible to dynamically configure distributed systems. For example, suppose a server implements a standardized interface to a file system. To allow remote clients to access the file system, the server makes use of a proprietary protocol. Normally, the client-side implementation of the file system interface, which is based on that protocol, would need to be linked with the client application. This approach requires that the software be readily available to the client at the time the client application is being developed.

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Fig. below.

This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Different solutions are discussed below and in later chapters.

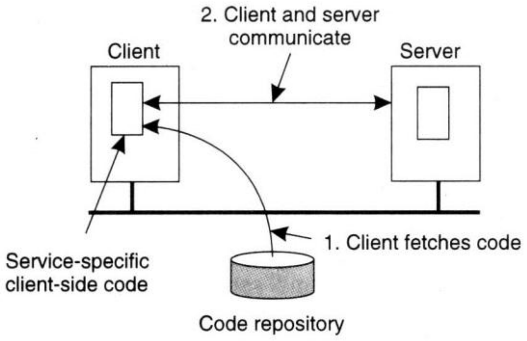

2. Client and server communicate
Client
Server
1. Client fetches code
Service-specific client-side code
Code repository

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

The important advantage of this model of dynamically downloading client-side software, is that clients need not have all the software preinstalled to talk to servers. Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed. Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like.

Models for Code Migration

Q.12 What are Models for Code Migration?

(A) Although code migration suggests that we move only code between machines, the term actually covers a much richer area. Traditionally, communication in distributed systems is concerned with exchanging data between processes. Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target. In some cases, as in process migration, the execution status of a program, pending signals, and other parts of the environment must be moved as well.

The bare minimum for code migration is to provide only **weak mobility.** In this model, it is possible to transfer only the code segment, along with perhaps some initialization data. A characteristic feature of weak mobility is that a transferred program is always started from its initial state. This is what happens, for example, with Java applets. The benefit of this approach is its simplicity. Weak mobility requires only that the target machine can execute that code, which essentially boils down to making the code portable.

In contrast to weak mobility, in systems that support **strong mobility** the execution segment can be transferred as well. The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off. Clearly, strong mobility is much more powerful than weak mobility, but also much harder to implement. An example of a system that supports strong mobility is D'Agents.

Irrespective of whether mobility is weak or strong, a further distinction can be made between sender-initiated and receiver-initiated migration. In **sender-initiated** migration, migration is initiated at the machine where the code currently resides or is being executed. Typically, sender-initiated migration is done when uploading programs to a compute server. Another example is sending a search program across the Internet to a Web database server to perform the queries at that server. In **receiver-initiated** migration, the initiative for code migration is taken by the target machine. Java applets are an example of this approach.

Receiver-initiated migration is often simpler to implement than sender-initiated migration. In many cases, code migration occurs between a client and a server, where the client takes the initiative for migration. Securely uploading code to a server, as is done in sender-initiated migration, often requires that the client has previously been registered and authenticated at that server. In other words, the server is required to know all its clients, the reason being is that the client will presumably want access to the server's resources such as its disk. Protecting such resources is essential.

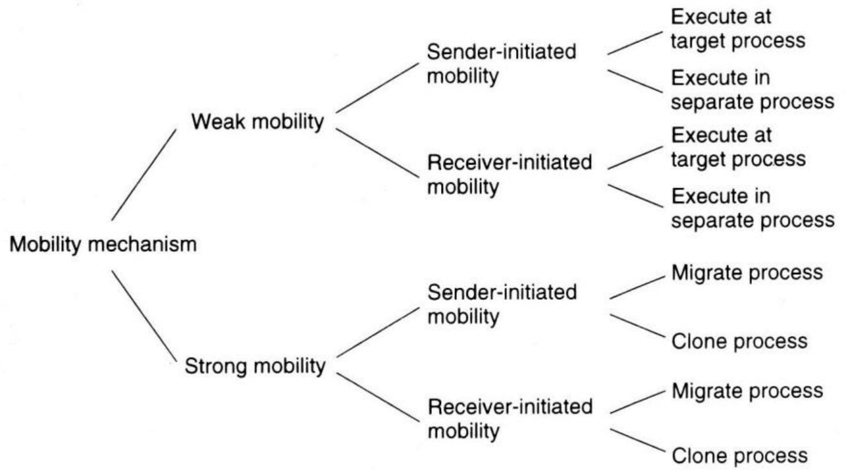

Mobility mechanism
Weak mobility
Strong mobility
Sender-initiated mobility
Receiver-initiated mobility
Sender-initiated mobility
Receiver-initiated mobility
Execute at target process
Execute in separate process
Execute at target process
Execute in separate process
Migrate process
Clone process
Migrate process
Clone process

Figure. Alternatives for code migration.

Migration in Heterogeneous Systems

Distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture. Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform, perhaps after recompiling the original source. Also, we need to ensure that the execution segment can be properly represented at each platform.

Problems can be somewhat alleviated when dealing only with weak mobility. In that case, there is basically no runtime information that needs to be transferred between machines, so that it suffices to compile the source code, but generate different code segments, one for each potential target platform.

In the case of strong mobility, the major problem that needs to be solved is the transfer of the execution segment. The problem is that this segment is highly dependent on the platform on which the process is being executed. In fact, only when the target machine has the same architecture and is running exactly the same operating system, is it possible to migrate the execution segment without having to alter it.

Video

References

1. https://www.youtube.com/watch?v=kgcUVZARftQ
2. Distributed Systems by P K Sinha
3. Distributed Operating Systems by Tanenbaum

EQ and GQ

1. What are resource and process?

2. Explain resource management.

3. What is a global scheduling algorithm?

4. List and explain the desirable towards of good global scheduling algorithm.

5. Explain
(i) Stability (ii) Scalability (iii) Fault tolerance of DFS

6. What are different techniques for scheduling processes?

7. Elaborate in brief about a scheduling technique.

8. Write short notes on :
(i) Task assignment approach
(ii) load balancing approach
(iii) load sharing approach

9. Explain the different Load estimation policies and process transfer policies used by load balancing algorithms.

10. Explain the taxonomy of load balancing algorithms.

11. Explain brief :
(i) Deterministic versus probabilistic
(ii) Centralized versus distributed
(iii) Co-operative versus non co-operative

12. Explain the issues in designing load balancing algorithms.

13. Explain process transfer policies.

14. Explain location policies.

15. State information exchange policies.

16. Explain load-sharing approach.

17. Explain process management.

18. What is process integration. Explain in detail.

19. What are threads?

20. Distinguish between thread and process.

21. Explain thread package. Explain issues in designing a thread package.

22. How to synchronize thread?

23. Write short note on thread synchronization

24. Explain thread scheduling.

25. How to implement a threads package?

26. Explain signal handling.

27. What are the common strategies sued for handling deadlocks in distributed systems.

28. Differentiate between Process and Threads using proper examples.

29. What are the issues in designing Load Balancing algorithms?

30. Describe the different models for organizing threads. Explain the working of a multi-threaded server.

31. Write short note on Process Migration in Heterogeneous systems.

32. Compare processes and threads. Explain user and kernal level threads execution and also the need of light-weight threads.

33. Write short note on Issues in Designing Load Sharing Algorithms.