**Lecture 8**

**RESOURCE MANAGEMENT**

## **Structure:**

8.1 Desirable Features of a Good Global Schedules Algorithm

8.2 Task Assignment Approach

8.3 Process Migration

8.4 Desirable Features of a Good Process Migration

8.5 Load Sharing Approach

**Learning Activity 8.0**

1. Explain Process Migration

2. Discuss Load Sharing Approach

**8.1 DESIRABLE FEATURES OF A GOOD GLOBAL SCHEDULES ALGORITHM**

**8.1.1 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. Scheduling algorithms that operate based on the information about the characteristics and resource requirements of the processes normally pose an extra burden upon the users who must specify this information while submitting their processes for execution.

**8.1.2 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. For this, sometimes it is also recommended that the scheduling algorithm should possess the flexibility to migrate a process more than once because the initial decision of placing a process on a particular node may have to be changed after some time to adapt to the new system load. This feature may also require that the system support preemptive process migration facility in which a process can be migrated from one node to another during the course of its execution.

**82**

**8.1.3 Quick Decision Making Capability:**

A good process scheduling algorithm must make quick decisions about the assignment of processes to processors. This is an extremely important aspect of the algorithms and makes many potential solutions unsuitable. For example, an algorithm that models the system by a mathematical program and solves it on line is unsuitable because it does not meet this requirement. Heuristic methods requiring less computational effort while providing near optimal results are therefore normally preferable to exhaustive (optimal) solution methods.

**8.1.4 Balanced System Performance and Scheduling Overhead**

Several global scheduling algorithms collect global state information and use this information in making process assignment decisions. A common intuition is that greater amounts of information describing global system state allow more intelligent process assignment decisions to be made that have a positive affect on the system as a whole. In a distributed environment, however, information regarding the state of the system is typically gathered at a higher cost than in a centralized system. 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.

**8.1.5 Stability:**

A scheduling algorithm is said to be unstable if it can enter a state in which all the nodes of the system are spending all of their time migrating processes without accomplishing any useful work in an attempt to properly schedule the processes for better performance. This form of fruitless migration of processes is known as processor thrashing. Processor thrashing can occur in situations where each node of the system has the power of scheduling its own processes and scheduling decisions either are made independently of decisions made by other processors or are based on relatively old data de to transmission delay between nodes. For example, it may happen that node n1 and n2 both observe that node n3 is idle and then both offload a portion of their work to node n3 without being aware of the offloading decision made by the other. Now if node n3 becomes overloaded due to the processes received from both nodes n1 and n2 , 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.

**8.2 TASK ASSIGNMENT APPROACH**

**8.2.1 The Basic Idea:**

In this approach, a process is considered to be composed of multiple tasks and the goal is to find an optimal assignment policy for the tasks of an individual process. Typical assumptions found in task assignment work are as follows:

A process has already been split into pieces called tasks. This split occurs along natural boundaries, so that each task will have integrity in itself and data transfers among the tasks will be minimized.

The amount of computation required by each task and the speed of each processor are known.

The cost of processing each task on every node of the system is known. This cost is usually derived based on the information about the speed of each processor and the amount of computation required by each task.

The Interprocesses Communication (IPC) costs between every pair of tasks is known. The IPC cost of considered zero (negligible) for tasks assigned to the same node. They are usually estimated by an analysis of the static program of a process. For example during the execution of the process, if two tasks communicate n times and average time for each intertask communication is t, the intertask communication cost for the two tasks is n t .

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.

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.

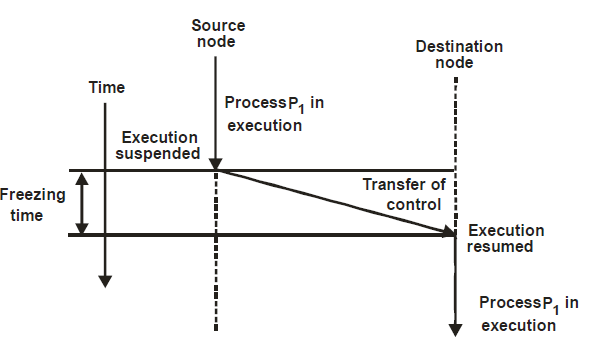
Minimization of IPC costs

Quick turnaround time for the complete process A high degree of parallelism

Efficient utilization of system resources in general

**8.3 PROCESS MIGRATION**

Process migration is the relocation of a process from its current location (the source node) to another node (the destination node). The flow of execution of a migrating process is illustrated in Figure 8.1.



**Fig 8.1 : Flow of execution of a migrating process**

A process may be migrated either before it starts executing on its source node on during the course of its execution. The former is known as non-preemptive process migration, and the latter is known as preemptive process migration. Preemptive process migration is costlier than non-preemptive process migration since the process environment must also accompany the process to its new node for an already executing process.

Process migration involves the following major steps:

1. Selection of a process that should be migrated.
2. Selection of the destination node to which the selected process should be migrated
3. Actual transfer of the selected process to the destination node.

The first two steps are taken care of by the process migration policy and the third step is taken care of by the process migration mechanism. The policies for the selection of a source node, a destination node, and the process to be migrated on resource management.

**85**

**8.4 DESIRABLE FEATURES OF A GOOD PROCESS MIGRATION**

A good migration mechanism must process transparency, minimal interferences, minimal residue dependencies, efficiency, robustness, and communication between co-processes.

**8.4.1 Transparency:**

Transparency is an important requirement for a system that supports process migration. The following levels of transparency can be identified:

1. **Object access level:** Transparency at the object access level isthe minimum requirement for a system to support non – preemptive process migration facility. If a system supports transparency at the object access level, access to objects such as files and devices can be done in a location independent manner. Thus, the object access level transparency allows free initiation of programs at an arbitrary node. Of course, to support transparency at object access level, the system must provide a mechanism for transparent object naming and locating.

System call and interprocess communication level. So that a migrated process does not continue to depend upon its originating node after being migrated. It is necessary that all system calls, including interprocess communication, are location independent. Thus, transparency at this level must be provided in a system that is to support preemptive process migration facility. However, system calls to request the physical properties of a node need not be location independent.

Transparency of interprocess communication is also the transparent redirection of messages during the transient state of process that recently migrated. That is, once a message sent, it should reach its receiver process without the need for resending a from the sender node is sure the receiver process moves to another node before the message is received.

**8.4.2 Minimal Interference:**

Migration of a process should cause minimal interference to the progress of the process involved the system as a whole. One method to achieve this is by minimizing the freezing time of the process being migrated. Freezing time is defined as the time period for which the execution of the process is stopped for transferring its information to the destination node.

**8.4.3 Minimal Residual Dependencies:**

No residual dependency should be left on the previous node. That is, a migrated process should not in any way continue to depend on its previous node once it has started executing on its new node since, otherwise, the following will occur.

It may be noted that in the model described above, the tasks of a process were assigned to the various nodes of the system. This model may be generalized to the general task assignment problem in which several processes are to be assigned. In this case, each process is treated to be a task of the process force and the inter-process communication costs are assumed to be known.

Several extensions to the basic task assignment model described above have been proposed in the literature. In addition to the task assignment cost and the inter-task communication cost parameters of the basic task assignment model, the extended models take into account to her parameters such as memory size requirements of the task and memory size constraint of the processors, precedence relationship among the tasks, and so on. However, we will not discuss this topic any further because of the limited applicability of the task assignment approach in practical situations.

**8.4.4 Load Balancing Approach:**

The scheduling algorithms using this approach are known as load balancing algorithms or load leveling algorithms. 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 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 resources point of view, the metric involved is the total systems 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.

**8.4.5 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. To describe a specific load balancing algorithm, a taxonomy user traces paths through the hierarchy. A description of this taxonomy is given below.

**8.4.6 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.

**8.4.8 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.

**Uncontrolled:** In this case, remote process arriving at a node istreated 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.

**Controlled:** To overcome the instability problem of theuncontrolled 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. Several system designers feel that process migration is an expensive operation and hence a process should not be allowed to migrate too frequently. Hence this group of designers normally favors an irrevocable migration policy. That is, the upper limit of the value of migration count is fixed to t, and hence a process cannot be migrated more than once under this policy. However, some system designers feel that multiple process migrations, especially for long processes, may be very useful for adapting to the dynamically changing states of the nodes. Thus this group of designers sets the upper limit of the value of migration count to some value k 1 . The value of k may be decided either statically of dynamically. Its value may also be different for processes having different characteristics. For example, a long process (a process whose execution time is large) may be allowed to migrate more times as compared to a short process.

**8.5 LOAD SHARING APPROACH**

Several researchers believe that load balancing with its implication of attempting to equalize workload on all the nodes o the system, is not an appropriate objective. This is because the overhead 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 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 for the proper utilization the resources of a distributed system, it is not required to balance 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.

**8.5.1 Issues in Designing Load Sharing Algorithms:**

Similar to the load balancing algorithms, the design of a load sharing algorithm also requires that proper decisions be made regarding load estimation policy, process transfer policy, state information exchange policy, location policy, priority assignment policy, and

with thr eads facility, a process having a single thread corresponds to a process of a traditional operating system. Threads are often referred to as lightweight processes and traditional processes are referred to as heavyweight processes.

**8.5.2 Motivations for Using Threads:**

The main motivations for using a multithreaded process instead of multiple single threaded processes for performing some computation activities are as follows:

1. The overheads involved in creating a new process are in general considerably greater than those of creating a new thread within a process.
2. Switching between threads sharing the same address space is considerably cheaper than switching between processes that have their own address space.

1. Threads allow parallelism to be combined with sequential execution and blocking system calls. Parallelism improves performance and blocking system calls make programming easier make programming easier.
2. 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.

These advantages are elaborated below:

The overheads involved in the creation of a new process and building its execution environment are liable to be much greater than creating a new thread within an existing process. This is mainly because when a new process is created its address space has to be created from scratch, although a part of it might be inherited from the process’s parent process. However, when a new thread is created, it uses the address space of its process that need not be created from scratch. For instance, in case of a kernel supported virtual memory system, a newly created process will incur page faults as date and instructions are referenced for the first time. Moreover, hardware caches will initially contain no data values for the new process, and cache entries for the process’s data will be created as the process executes. These overheads may also occur in thread creation, but they are liable to be less. This is because when the newly created thread accesses code and data that have recently been accessed by other threads within the process, it automatically takes advantage of any hardware or main memory caching that has taken place.

Threads also minimize context switching time, allowing the CPU to switch from one unit of computation to another unit of computation with minimal overhead. Due to the sharing of address space and other operating system resources among the threads of a process, the overhead involved in CPU switching among peer threads is very small as compared to CPU switching among processes having their own address spaces. This is the reason why threads are called lightweight processes.

**True file service:** It is concerned with the operation on individualfiles, such operations for accessing and modifying the data in files and for creating and deleting. To perform these primitive file operations correctly and efficiently, typical design issues of a true file service component include file accessing mechanism, file sharing semantics, file caching mechanism, file replication mechanism, concurrency control mechanism, data consistency and multiple copy update protocol, and access control mechanism. Note that the separation of the storage service from the true file service makes it easy to combine different methods of storage and different storage media in a single file system.

**Name service:** IT provides a mapping between text names for filesand references to files, that is, file IDs. Text names are required because, file IDs are awkward and difficult for human users to remember and use. Most file systems use directories to perform this mapping. Therefore, the name service is also known as a directory service. The directory service is responsible for performing directory related activities such as creation and deletion of directories, adding a new file to a directory deleting a file from a directory, changing the name of a file, moving a file from one directory to another, and so on.

The design and implementation of the storage service of a distributed file system is similar to that of the storage service of a centralized file system. Readers interested in the details of the storage service may refer to any good book on operating systems. Therefore, this chapter will mainly deal with the design and implementation issues of the true file service component of distributed file systems.

**Revision Exercise**:

1. What are the issues in designing Load –Balancing algorithms?
2. Discuss the features of a Local Scheduling algorithm?
3. Why are heuristic methods that provide near optimal results preferred over optimal solution methods in scheduling algorithms?
4. Discuss the practical applicability of the load-balancing approach as a scheduling scheme.
5. Load – balancing in a strictest sense is not achievable in distributed systems. Justify?