Processes, Threads, Memory and Resources in Distributed Systems

Mohammed A. M. Ibrahim

Faculty of Engineering &Information Technology,
Taiz University,
Republic of Yemen.
E-mail: sabri1966@yahoo.com.

Abstract

Multiple processors and shared memory are important paradigm for building a distributed system. For building applications, threads proves to provide better performance than processes. This paper presents some of basic requirements for building a distributed system. It looks at conflict arises due to requests to shared resources in distributed systems. The solution of these conflicts in a timely manner is important. Several models to resolve conflicts exist. Some appealing ones are discussed in this paper. With several copies of data at different sites, a reference on memory is required to leave the data consistent throughout the memory blocks. Changing one copy and leaving the other is unacceptable. The memory consistency model of shared memory multi-processor solve the problem, and influence both the performance and the programmability of the system.

1. Introduction.

Keywords: processes, threads, distributing operating system, network operating system, resource allocation

This paper discusses various issues on building a distributed system. The discussion is based on the knowledge gained during the study of Distributes Operating System course, one of the prerequisite of my Ph.D. program. During the study various technics of building a distributive system, as opposed to traditional centralized system, were covered.

The outline of the paper is as follows. In section 2, the functionality of threads are described, and the question of why threads are dominantly used versus processes is answered. Section 3 discusses micro-kernel and the services it provides to the distributes system. Section 4 discusses resource allocation algorithm in Distributed Operating Systems (DOS). The consistency of data copy is presented in section 5. Section 6 analyses the difference between Network Operating System (NOS) and (DOS). The last section summarizes and concludes the paper.

2. Why Using Threads Versus Processes.

Threads, or threads of control, are min-processes. Kai Hwang [4], defines a thread as a sequence of instructions being executed. A process may simply be defined as an address space and a single thread of control. Multiple threads of control can share one address space and still run in parallel. A server with multiple threads of control, one thread could be running while the second one is slipping Such execution result in *higher throughput* and *better performance*. Higher throughput and better performance can't be achieved by creating two independent server processes because they must share a common buffer cache which require them to be in the same address space. This is the main reason as to why threads are used versus processes. A process is always owned by a single user who can create multiple threads so that they can cooperate.

Operation: Threads cooperates with each other to perform certain task, sharing the same global variables. Processes can not easily achieve cooperation. In addition threads can share the same open files, child process, and even a timer. Thread operations such as creation, termination and synchronization, are less expensive than processes because they are implemented by thread library in the same user space without entering the kernel. Processes on the other hand incurred

overheads during operation due to copying, context switching, and crossing protection boundary [4].

Management: Creating a thread is much simpler than creating a process as there is no need to create a new address space. Only the following information needs to be maintained for a specific thread. Although a process with multiple threads still have a single address space shared by all threads, each thread, logically, has its own registers, its own counter and its own stack.

3. Microkernel

Most distributed systems are equipped with an operating system which provide networking facilities. One of such systems is a micro-kernel. The user process system calls to remote machines are made by trapping the kernel, which does the routing works, and then invoke the called machine. After having the work done in a remote machine the micro-kernel return the desired result to the user process. The micro-kernel is so important in distributed systems because the services it provides are difficult or expensive to provide anywhere. With micro-kernel it is possible to have distributed systems with multiple file servers, one supporting MS-DOS and another supporting UNIX file service. New services can easily be implemented, installed, and debugged with micro-kernel.

Primarily the micro-kernel has four functions [7].

1. Manage process and threads.
2. Provide low-level memory management.
3. Support inter-process communication mechanism.
4. Handle low level input/output.

Brief discussion on each of these.

A typical example of use of a micro-kernel, is managing threads in Amoeba file server. The micro-kernel breaks the server into multiple threads, every incoming request is assigned a separate thread to work on. By splitting the server into multiple threads, each thread can be purely sequential, if it has to block waiting for I/O.

Micro-kernel enables threads to allocate and deallocate blocks of

memory, called segments, which can be used for text, data, stacks, or any other purpose desired by the process

Point- to-point and group communication in Amoeba are supported by the micro-kernel. With point-to-point communication, a client sends a message to a remote server by having its stub trapping the kernel, which then pass the message to the remote kernel. The replay also traces the opposite path, being supported by the server and client kernels. For the client to access a group, the only way is to do RPC with one of the members. RPC is supported by the micro-kernel using principal primitive system calls, *get_request*, *put_request* and *trans*. The most advantage of micro-kernel is to enable communicating parties achieve reliable communication through a sequence of events [7].

As per function number 4, for each I/O device attached to a machine, there is a device in the kernel. The driver manages all I/O for the device.

4. The Resource Allocation Algorithm in Distributed Operating System (DOS).

Simultaneous requests to shared resources in distributed systems, may result into conflicts. For the system efficiency resolution of these concurrent requests is necessary. Patterns of access to conflict can be categorized on the basis of two attributes; (1) the number of resources requested by a process (or thread), and (2) whether a request in *single-step* or *incremental* [2]. In incremental request, a process holds onto previously granted resource, while making further request. In a single step case, a process request its resources all at once.

Synchronized algorithms are need to decide which process should run on which machine. The decision to be made depends on may factors such as the knowledge about a process behavior, whether processors are centralized or decentralized etc. The best allocation always aims at *maximizing CPU utilization* and *minimizing response time.*

Transfer polices are also considered, decision is to be made whether a process is to run on the local machine or to be transferred elsewhere. For workstation model the question may be when to run a

process locally and when to look for an idle workstation. For the process pool model a decision is usually made for every new process created, whether to run to completion in one particular place, *(omnigatory)* or to move it from one place to another as it runs *(migratory)*. All processors know their own load, and can tell others about their state; whether overloaded, or underloaded. An example depicted from [7], can be seen on figure 1-(a). Here an overloaded machine sends out request for help to other machines. In contrast fig 1-(b), a machine that is idle or underloaded announces to other machine that is prepared to take more load.

Centralized as well as decentralized resource allocation algorithms have been proposed. The popular ones includes [7]:

- Graph theoretic deterministic algorithm.
- A Centralized algorithm.

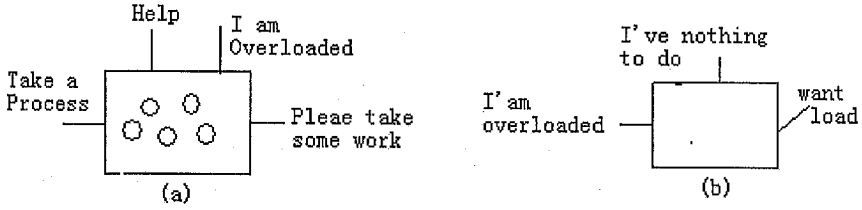

Figure 1 (a) Overloaded machine Seeking help (b) Underloaded machine seeking for some work

- Hierarchical algorithm.
- A sender-initiated distributed algorithm.
- A receiver-initiated heuristic algorithm and
- Bidding algorithm.

Two of these are briefly discussed.

The graph theory algorithm assumes that the number of CPUs and memory requirement are known in advance. In case the number of CPUs k is smaller than the number of processes, several processes are assigned to each CPU. The assignment is done such that the network traffic is minimized as shown in [7] figure 4-17 pp.204.

Centralized algorithm ensures that each workstation gets a fair share

of computing power [6],[7]. Allocation here is based its decision on a *usage table*, that keeps watch the processor status. The usage table coordinator allocate the process to a particular process upon a parent machine request .

Hierarchical algorithm processors are organized as a group of workers in hierarchy, with deans, department head, etc. These bosses does the work of assigning work to processors.

If a process has multiple processes running on it, scheduling become an issue. When a group of related and heavily interacting processors are all running on different processors, scheduling should support inter-process communication. Several scheduling algorithm which takes intercommunication into account exist. *Co-scheduling* suggested by Ousterhout (1982) [6],[7], is one of them. It ensures that all members of the group run at the same time with each CPU using round robin scheduling algorithm.

5. The Consistency of Data Copy

When Lu and Hudack, (1986), proposed a scheme known as *Distributed Shared Memory* (DSM), that is both easy to build and to program, its implementation exhibit poor performance and latecy, as pages were sent back and forth the network [7].

One possible optimization was to replicate the shared variables on multiple machines. Allowing multiple copies seem to improve performance problem, but introduces another problem of how to keep all the copies consistent. How to keep several copies of data on different site, largely depends on the system architecture; i.e. the connection between the memory and the CPU.

For multi-processors based on bus architecture, the problem might be two or more CPUs trying to access the memory at the same time. This problem is solved by having a CPU request a permission to use the bus. Only after granted permission it can use the bus and acquire memory access. Centralized and decentralized methods ways of granting permission are available.

To reduce the bus load, *snooping cache* are used [1],[7]. When a CPU first read a word from memory, that word is stored in the cache of

the CPU making a request. If that word is needed again later, the CPU jus take it from its own cache, thus minimizing memory reference and reducing bus traffic. Each CPU does its caching independently. The problem arises when a CPU wants to write a word that is held in other CPUs cache. How to make sure that different cache do not contain different values of the same memory location becomes an issue.

One technique used to solve this problem is w*rite through protoco*l [7]. It means that; "If the word is currently in the cache, update the cache entry and then update the memory. Let other caches holding the word *invalidate* their entries to make the memory up-to-date".

An alternative to invalidating the cache entries is to *update* all others. However, it is not the best approach, as it demands supplying cache entry, which makes it slower and causing more network circles.

With large systems such as Dash Switched multiprocessor in which a system is built as a hierarchy of clusters and super-clustors, [7] Fig 6-7 pg. 304, the CPU doing the write ensures that it is the owner of the own copy of the cache block in the system.

Write can only proceed only if its block in the cache in DIRTY. If it has a CLEAN block, all other copies in the home cluster are invalidated, i.e. declared INVALID.

NUMA multiprocessor is another design with single virtual address space visible to all CPUs. It ensures that, when any CPU writes a value to a location A, a subsequent read of A will return the value just written.

In an attempt to influence both performance and the programmability of the system, various memory consistency models (or *memory model*) have been invented. The most intuitive model, defined by Lamport (1979) , is *sequential consistency* model. It says that all processes see all memory reference in the same order. *Causal consistence, PRAM consistency* and *processor consistency* models also exist, but these weaken the concept that processes see all memory reference in the same order. For higher performance, several alternative models have been proposed. However, many of these are hardware-centric in nature and difficult to program as they place strong restrictions on software.

6. The Difference between DOS and NOS

Network operating system is a software for machines with multiple CPUs, that allow users at independent workstations to communicate via a shared file system while leaving each user as a master of his own work station. It is loosely-coupled software on a loosely-coupled hardware [7]. A typical example of NOS is a network of workstation in company or a university connected by a LAN. Each user has a personal wok-station with its own operating system. It may also have a hard disk.

All commands are normally run locally right on the workstation. It is also possible for a user to log into another workstation remotely using a command such us.

rlogin machine

The user has to log out first if he want to switch into a different machine. Files are copied from one machine to another by using copy command.

rpc machine1:file1 machine2:file2

Communication and information sharing are provided by global file system, supported by *file server*s. The file server accept request from user programs running on the *clients* to read and write files.

On the other hand Distributed Operating system (DOS), combine the entire collection of hardware and software into a single integrated system, acting like a virtual uniprocessor [7]. It is tightly coupled software on a loosely coupled hardware. Communication is achieved through message passing. The goal of DOS is to create a *single-system image* i.e. to create an illusion in the users minds that the entire network is a single time-sharing system rather than a collection of distinct machine. Figure 2 gives a summary of differences between NOS and DOS

Item	**Network Operating system**	**Distributed Operating System**
Look like Virtual Uniprocessor	NO	YES
All machines run the same operating system	NO	YES
Communication is achieved by	Shared files	Message passing
Files have well defined semantics	Usually no	YES

Figure 2. Difference between NOS and DOS.

7. Summary And Conclusion.

This paper has discussed some ideas of building a distributed system. It has indicated that even though all distributed systems have many CPUs, different ways in connecting them exist. Bus based multiprocessor and switched multiprocessors are among commonly used architectures.

On software point of view, thread are preferred versus processes as they are easy to build, can cooperate to perform certain task, and thus provide higher throughput and better performance. On the other hand process work independently, and are less efficient.

The micro-kernel plays an important role in a distributed system. Among other things, it manages processes and threads, and supports communication. A typical usage of micro-kernel is in Amoeba and Mach operating systems.

Resource allocation in distributed systems is a critical issue in improving performance. Concurrent requests of resources by processes may result into conflicts. Resource allocation algorithms have been devised, by researches, to solve the problem. Graph theory, A centralized algorithm and many others allocate resources aiming at maximizing CPU utilization and throughput.

Memory consistency models ensures that, a memory reference leave data consistence throughout the memory blocks. A standard against all models, and more practical is *sequential consistence.*

References.

[1] Agarwal, A., and Cherian, M., Adaptive Backoff Synchronization Techniques", "Proceeding. 16th Ann. International Symposium on Computer Architecture ACM, pp.396-406, 1989.

[2] Choi Manhoi, and Singh Ambuj, K. "Dynamic Resource Allocation using Local Views. IEEE 14th International Conference on Distributed Computing Systems", 1994.

[3] Johson, K.L., Kaashoek, M. F., and Wallach, D. A., CRL: "High performance All-Software Distributed Shared Memory". In 15th symposium on operating systems Principals, 1995.

[4] Kai Hwang, and Zhiwe Xu, "Scalable Parallel computing Technology, Archicture, Programming", McGraw-Hill Inc., 1998.

[5] Mutka, M.W. and Livny, M., "Scheduling Remote Processor Capacity in a Workstation-Processor Bank Network," In IEEE 7th International Conference On Distributed Computing Systems, pp. 2-9, 1987.

[6] Ousterhout,J.K., "Scheduling Techniques for concurrent systems", In IEEE 3rd International Conference On Distributed Computing systems, pp. 22-30, 1982.

[7] Tanenbaum Andrew S., "Distributed Operating system", Prentice-Hall inc. 1995