Operating Systems for Parallel Processing

Felician Alecu

Academy of Economic Studies Bucharest

ROMANIA

Abstract: A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. The parallelism can be achieved by executing multiple processes on different processors. A distributed operating system is a special kind of software that is used to manage the distributed system shared resources, the process scheduling activity and the implemented communication and synchronization mechanisms. Basically, it represents the extension for multiprocessor architectures of multitasking and multiprogramming operating systems. Four distributed operating systems categories can be identified by combining loosely coupled and tightly coupled hardware and software.

Keywords: Multiprocessors, multicomputers, distributed systems, parallel processing and operating systems.

Alecu Felician is an assistant lecturer at the Economic Informatics Department of the Academy of Economic Studies Bucharest. As PhD student, his current research interests include parallel systems and grid computing. He is the author of more than 10 papers concerning parallel processing published in several journals, books and proceedings of national or international conferences.

1. Introduction

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. Based on this definition, a parallel computer could be a supercomputer with hundreds or thousands of processors or could be a network of workstations.

Few years ago, parallel computers could be found only in research laboratories and they were used mainly for computation intensive applications such as numerical simulations of complex systems. Today, there are a lot of parallel computers available on the market used to execute both data intensive applications in commerce and computation intensive applications in science and engineering.

Concurrency becomes a fundamental requirement for algorithms and programs. A program has to be able to use a variable number of processors and also has to be able to run on multiple processors computers architectures.

A distributed system is a set of interconnected independent computers that appear to the user like a single one. Multiprocessors, multicomputers connected through static or dynamic interconnection networks and workstations connected through local area network are examples of such distributed systems.

The main difference between parallel systems and distributed systems is the way in which these systems are used. A parallel system uses a set of processing units to solve a single problem. A distributed system is used by many users together.

2. Processes and Threads

The base entity in computer programming is the process also called as task. The concept is taken from uniprocessor systems sequential programming. A process is a dynamic entity and represents an execution instance of a code segment. It has its own status and data. The parallelism can be achieved by executing multiple processes on different processors.

Processes can be found in all operating systems (multiprogramming, multitasking, parallel and distributed). A process is created by the operating system every time when running an executable file. The process will receive an ID that represents a unique number that will identify the process in the system.

Every process has its own virtual addresses space representing the range of addresses that can be accessed. A process cannot access other process addresses space. The process execution context is formed from process segments and process resources (opened files, synchronization objects, working folder)

A lot of operating systems are implementing a virtual memory mechanism for loading just a part of a process addresses space into the physical memory. The virtual memory mechanism can use paging, swapping or a combination of these methods.

2.1. Process States

From the operating system point of view, a process can be in one of the following states:

- *dormant* – the process is not known by the operating system because it was not created yet. Every program not executed is in a dormant state.
- *ready to run* – all resources needed, except the processor, are allocated to the process. The scheduler of the operating system selects one ready to run process at a time and executes it.
- *execution* – the process has allocated all resources needed, including processor. Only one process at a time can be in this state. The executed process can request a service from the operating system like an input/output operation or a synchronizing operation. In such a case, the process will be suspended by the operating system.
- *suspended* - a suspended process is waiting for an event, like a synchronization signal. Suspended processes are not competing for execution until the event is arising. At this moment, the operating system changes the process state to ready to run. The scheduler will select the process and will execute it.

Every process has its own control block where the operating system stores details about the process. Every time a new process is created, the operating system assigns a new control block for that process. When the process ends its execution, the control block is removed from the memory and destroyed. A dormant process has no control block because the process is not created yet.

The control block records process details like process ID, process priority and its state, processor registers and flags, scheduling information, used resources and so on.

The operating system can switch between two processes. The operation is named process switch and it could be requested only by the operating system. Process switch is a complex operation with a significant overhead that can affect system performance.

2.2. Threads

A thread is a software method used to reduce the process switch overhead. It represents a lighter process with a simplified status. In multithreading systems, a thread is the base scheduling entity.

A thread has its own stack and hardware status. The thread switch implies saving and retrieving of hardware status and stack. The thread switch between threads of the same process is fast and efficient because all other resources, except processor, are managed by the parent process. Unfortunately, the thread switch between threads of different processes involves process switch overhead.

2.3. Processes and Threads Creation

A child process can be created dynamically by another running process using the *fork* function of the operating system.

The fork operation is used to divide a program in two sequences that can be executed in parallel. The child process and its parent will execute distinct parts of the program segment. Both processes are executed from the same text segment. The child process, at creation time, receives a copy of parent data segment.

The *exec* function of the operating system can be used to change the text segment of the child process. In such a way the parent and child processes will have different text and data segments. The parent process can be forced to wait for the end execution point of the child process by using the *join* function.

Windows operating systems are thread oriented because they support multiprocessor shared memory architecture. The basic allocation entity in Windows is the thread. Every process contains at least one thread (named main thread). Threads can create new other child threads that will share parent address space and resources such as files and synchronization objects.

Windows programmers can access operating system functions through the API interface (Application Programming Interface). A process can create a new child process using *CreateProcess* function. Also, the *CreateThread* function can be used threads creation.

3. Distributed Operating systems

A distributed operating system is the extension for multiprocessor architectures of multitasking operating systems.

Multitasking operating systems are able to concurrently execute multiple processes on single processor computers using resources sharing. A running process is not allowed to access or to destroy another process data.

A distributed system is a set of computers that communicate and collaborate each other using software and hardware interconnecting components. Multiprocessors (MIMD computers using shared memory architecture), multicomputers connected through static or dynamic interconnection networks (MIMD computers using message passing architecture) and workstations connected through local area network are examples of such distributed systems.

A distributed operating system manages the system shared resources used by multiple processes, the process scheduling activity (how processes are allocating on available processors), the communication and synchronization between running processes and so on.

Multiprocessors are known as tightly coupled systems and multicomputers as loosely coupled systems.

The software for parallel computers could be also tightly coupled or loosely coupled. The loosely coupled software allows computers and users of a distributed system to be independent each other but having a limited possibility to cooperate. An example of such a system is a group of computers connected through a local network and sharing some resources. If the interconnection network breaks down, the individual computers could be used but without some features involving the common resources.

At the opposite side is the tightly coupled software. If we use a multiprocessor system in order to solve an intensive computational problem, every processor will operate with a data set and the final result will be obtained by combining the partial results. In such a case, an interconnection network malfunction may result in the incapacity of the computer to solve the problem.

Combining loosely and tightly coupled hardware and software we can identify four distributed operating systems categories. The loosely coupled software and tightly coupled hardware case is not met in practice and therefore it will not be covered below.

3.1. Network Operating Systems

Network operating systems represent the loosely coupled hardware and loosely coupled software case. A typical example of such a system is a set of workstations connected together through a local area network (LAN).

Every workstation has its own operating system. A user can execute a login command in order to connect to another station and also he can access a set of shared files maintained by the file server. There are only a few processing requirements at system level. Processes executed over the network do not need to synchronize. This is why there is no need to use the same operating system in every node.

3.2. Real Distributed Operating Systems

A real distributed operating system is the case of tightly coupled software used on a loosely coupled hardware.

The interconnection network is transparent for the users. This is why the computers appear to the users like a single multitasking system (virtual uniprocessor) and not like a set of independent computers. The users do not have to be concerned by the number of computers from the network.

No system available today on the market entirely fulfills this requirement.

The computers from the network will use the same operating system that will manage local resources like virtual memory and process scheduling.

A process has to be able to communicate with any other process, no matter if the second process is running on the same computer or on a different one. This is why the operating system has to use the same system calls routines and file systems for all computers.

3.3. Multiprocessing Operating Systems

Multiprocessing operating systems represent the case of tightly coupled software for tightly coupled hardware.

A multiprocessing operating system is acting like a multitasking UNIX system but having multiple processors in the system.

There is a single list of ready to run processes used by the operating system. When a new process is ready to run, it is added to the execution queue located in the shared memory area. Every time a processor becomes free, it extracts a process from the list and executes it.

The operating system has to implement mutual exclusion mechanisms (semaphores, monitors, locks or events using busy-wait or sleep-wait protocols) in order to protect the concurrent accesses to the list located in the shared memory. So, once a process has exclusive access to the ready to run processes list, it extracts the first process from the list, releases the list and executes the process.

The system also appears for the users as a virtual uniprocessor and the same operating system is executed in every processing unit.

4. Conclusions

Nowadays, commercial applications are most used on parallel computers. A computer that runs such an application has to be able to process large amount of data in sophisticated ways. We can say with no doubt that commercial applications will define future parallel computers architecture, but scientific applications will still remain important users of parallel computing technology. Trends in commercial and scientific applications are merging as commercial applications perform more sophisticated computations and scientific applications become more data intensive.

Today, a lot of parallel programming languages and compilers, based on dependencies detected in source code, are able to automatically split a program into multiple processes and/or threads to be executed concurrently on the available processors from a parallel system. The operating system of a parallel system has to make possible the communication between the processes and threads using shared memory or message passing mechanisms. Also, it has to support the applications in detecting, analyzing and managing the dependencies in complex programs. The mutual exclusion techniques have to be used in order to serialize the concurrent access to the shared resources of the distributed system.

REFERENCES

1. G. DODESCU, **Operating Systems**, ASE, 1997
2. G. DODESCU, B. OANCEA, M. RACEANU, **Parallel Processing**, Economic Publishing House, Bucharest, 2002
3. W. FOKKINK, W. BRAUER, **Introduction to Process Algebra**, Springer 2000
4. C. HUGHES, T. HUGHES, **Parallel and Distributed Programming Using C++**, Addison-Wesley, 2003
5. H. F. JORDAN, H. E. JORDAN, **Fundamentals of Parallel Computing**, Prentice Hall, 2002
6. J. JOSEPH, C. FELLENSTEIN, **Grid Computing**, Prentice Hall, 2003
7. A. S. TANENBAUM, **Distributed Operating Systems**, Prentice Hall, 1995
8. A. S. TANENBAUM, **Computer Networks**, Prentice Hall, 1996.