Operating Systems for Parallel Processing

Assistant Alecu Felician
Economic Informatics Department
Academy of Economic Studies, Bucharest

Abstract

Applications from our days have growing computational needs. This is why the concurrency becomes a fundamental requirement for algorithms and programs. The base entity in computer programming is the process or task. The parallelism can be achieved by executing multiple processes on different processors. A thread is a lighter process and it is used to reduce the process switch overhead. The operating system creates a process every time the RUN command is used on an executable file. Distributed systems are managed by distributed operating systems that represent the extension for multiprocessor architectures of multitasking and multiprogramming operating systems.

1. Introduction

Few years ago, parallel computers could be found only in research laboratories and they were used mainly for computation intensive applications like 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.

Applications from our days have growing computational needs. As computers become ever faster, we can expect that one-day computers will become fast enough. However, new application will arise and these applications will demand faster computers. In our days, commercial applications are the 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. These applications include graphics, virtual reality, decision support, parallel databases, medicine diagnosis and so on. We can say with no doubt that commercial applications will define future parallel computers architecture but scientific applications will remain important users of parallel computing technology.

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.

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.

According to Tanenbaum, a distributed system is a set of independent computers that appear to the user like a single one. So, the computers have to be independent and the

software has to hide individual computers to the users. MIMD computers and workstations connected through LAN and are examples of 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 process (or task) is the base entity in computers programming. The concept is taken from uniprocessor systems sequential programming. A process is a dynamic entity and represents an execution instance of a code segment. A process 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).

When using a RUN command on an executable file, a process is created by the operating system. The process will receive a process ID – 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 addresses space contains program text segment, data segment and stack segment. The text segment consists of program executable text and is shared by multiple processes. A process can read from text segment but is not allowed to write or to change the content of this segment. The next segment is data segment. It stores static and dynamic allocated data. This segment is not shared and cannot be accessed by other processes. The stack segment contains the process stack and is not shared.

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 dormant state.
- *Ready to run* – all resources needed are allocated to the process except processor. The operating system scheduler 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 operating system suspends current process.

- *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 information about the process. When a new process is created, the operating system creates 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.

A process control block consists of:

- Process ID (PID);
- Process priority;
- Process state (ready to run, suspended, execution);
- Hardware status (processor registers and flags);
- Scheduling information;
- Information about resources used by the process (files, input, output)

The operating system can switch between two processes. The operation is named process switch and is requested by the operating system only. 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 that affects the system performance. A thread is 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.

Another execution entity used by operating system derived from UNIX is the lightweight process. A lightweight process has a minimal state (hardware status and stack) and shares all parent process resources except processor. Lightweight processes are implemented in IRIX operating system for Silicon Graphics multiprocessor workstations and in DYNIX operating system running on Sequent Symmetry multiprocessors.

2.3. Processes and Threads Creation

A process can be created dynamically by another running process using the *fork* operating system function. A new child process is created when invoking fork function.

The fork operation is used to divide a program in two sequences that can be run in parallel. The child process will execute a program segment and its parent will execute the

other segment. Child and parent processes are executed from the same text segment. The child process, at creation time, receives a copy of parent data segment.

Using the *exec* operating function, it is possible to change the text segment of the child process. The parent and child processes will have different text and data segments.

The parent process can be forced to wait for child process execution end using the *join* function.

2.4. Processes and Threads Implementation in UNIX

A process is identified in UNIX by a process identifier (PID). The process identifier is a unique number in the system. Two processes will have different values for process identifier.

Fork function creates a new process in UNIX operating system. The child process receives a copy of parent memory and shared access to all parent opened files. So, the child process will have its own data and stack segments but will be executed from parent text segment.

Exec function can be used in order to change the text segment for the child process. In such a way, parent and child processes will be executed from different text segments. Exec function arguments are the file that will be executed by the child process and command line arguments of this file.

Wait function forces the parent to wait for child processes execution end. In some UNIX systems, fork and exec functions are grouped together into a single function – spawn.

2.5. Processes and Threads Implementation in Windows

Windows operating systems are thread oriented. Windows NT and Windows 2000 are multiprocessing operating systems because they support multiprocessor shared memory architecture.

The elementary allocation entity in Windows is the thread. Every process contains at least one thread (named main thread) and can create new threads that will share parent address space and resources (files, synchronization objects).

Programmers can access operating system functions through API interface (Application Programming Interface). A process can create a new child process using *CreateProcess* function. The result of using CreateProcess function is equivalent with using fork and exec combination in UNIX: a new process with a new text segment is created. The executable file that will be executed by the child process is passed to the function as parameter with command line arguments, if any.

If the child process successfully created, the function returns boolean value TRUE, otherwise the returned value will be FALSE.

A new thread can be created using *CreateThread* function. Before using the function we have to define the function to be executed by the new thread. This function will be passed as a parameter to *CreateThread*.

3. Distributed operating systems

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

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

Multiprogramming operating systems supports multitasking and have the ability to protect memory and to control concurrent processes access to shared resources. A multiprogramming operating system is also a multitasking operating system.

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 system is managed by a distributed operating system. 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. Every computer has its own memory, hard disk. There are some shared resources such files and printers. If the interconnection network broke down, individual computers could be used but without some features like printing to a non-local printer.

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

Combining loosely coupled software 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 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. Every user has its own workstation in exclusive use. A user can execute a login command in order to connect to

another station and also can access a set of shared files maintained by a workstation named file server.

There are only a few processing requirements at system level. Processes executed over the network do not need to synchronize.

The network operating system has to manage individual workstations and file servers and has to assure the communication between them. For this reason, workstations do not need to use the same operating system.

3.2. Real Distributed Operating Systems

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

Because the interconnection network is transparent for the users, the set of computers appears like a single multitasking system and not like a set of independent computers.

The computers appear to the users like a virtual uniprocessor or like a single system image. This is why the users don't have to be concerned by the number of computers from the network. No system available on the market today entirely fulfills this requirement.

All computers from the network use the same operating system. The same operating system kernel will run on every individual computer. The local kernel 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 different one. This is why the operating system has to use the same system calls routines for all computers. The operating system has also to use the same file system on all workstations without file name length limitations. Also, the operating system has to implement files security and protection policies.

3.3. Multiprocessing Operating Systems

Multiprocessing operating systems represent the tightly coupled software and tightly coupled hardware case.

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

The main characteristic of multiprocessing operating systems is a single list of ready to run processes used. When a new process is ready to run it is added to the list located in the shared memory area. The list can be accessed by any process. When a processor is 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 from 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 as a virtual uniprocessor for users and the same operating system is executed in every processing unit.

3.4. Examples of Distributed Operating Systems

A few distributed operating systems used on parallel computers are:

- IRIX operating system; is the implementation of UNIX System V, Release 3 for Silicon Graphics multiprocessor workstations;
- DYNIX operating system running on Sequent Symmetry multiprocessor computers;
- AIX operating system for IBM RS/6000 computers;
- Solaris operating system for SUN multiprocessor workstations;
- Mach/OS – a multithreading and multitasking UNIX compatible operating system;
- OSF/1 operating system developed by Open Foundation Software: UNIX compatible

Bibliography

1. G. Dodescu, *Operating Systems*, A.S.E. Publishing House, Bucharest, 1997.
2. Gh. Dodescu, B. Oancea, M. Raceanu, *Parallel Processing*, Economic Publishing House, Bucharest, 2002.
3. J. S. Gray, *Interprocess Comunications in UNIX*, Prentice Hall, 1997.
4. D. Gross, C. M. Harris, *Fundamentals of Queuing Theory*, Wiley, New York, 2003
5. R. H. Perrot, *Parellel Programming*, Addison-Wesley, Reading, MA, 1997.
6. M. Quinn, *Parallel Computing*, McGrow-Hill, 1994.
7. A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995.
8. A. S. Tanenbaum, *Computer Networks*, Computer Press AGORA, 1998.