Techno International Batanagar

Term Paper

Process Management

Name: Subhradeep Bhadra

Stream: CSE

Semester: 5th

Roll: 33200118010

Subject: Operating System

Sub Code: PCC-CS 502

Process Management

A process is a program in execution. An integral part of any modern-day operating system (OS). The OS must allocate resources to processes, enable processes to share and exchange information, protect the resources of each process from other processes and enable synchronization among processes. To meet these requirements, the OS must maintain a data structure for each process, which describes the state and resource ownership of that process, and which enables the OS to exert control over each process.

Multiprogramming

In any modern operating system there can be more than one instance of a program loaded in memory at the same time. For example, more than one user could be executing the same program, each user having separate copies of the program loaded into memory. With some programs, it is possible to have one copy loaded into memory, while several users have shared access to it so that they each can execute the same program-code. Such a program is said to be re-entrant. The processor at any instant can only be executing one instruction from one program but several processes can be sustained over a period of time by assigning each process to the processor at intervals while the remainder become temporarily inactive. A number of processes being executed over a period of time instead of at the same time is called concurrent execution.

A multiprogramming or multitasking OS is a system executing many processes concurrently. Multiprogramming requires that the processor be allocated to each process for a period of time and deallocated at an appropriate moment. If the processor is de-allocated during the execution of a process, it must be done in such a way that it can be restarted later as easily as possible.

There are two possible ways for an OS to regain control of the processor during a program's execution in order for the OS to perform de-allocation or allocation:

1. The process issues a system call (sometimes called a software interrupt); for example, an I/O request occurs requesting to access a file on hard disk.
2. A hardware interrupt occurs; for example, a key was pressed on the keyboard, or a timer runs out (used in pre-emptive multitasking).

The stopping of one process and starting (or restarting) of another process is called a context switch or context change. In many modern operating systems, processes can consist of many sub-processes. This introduces the concept of a thread. A thread may be viewed as a sub-process; that is, a separate, independent sequence of execution within the code of one process. Threads are becoming increasingly important in the design of distributed and client–server systems and in software run on multi-processor systems.

How Multiprogramming Increases Efficiency

A common trait observed among processes associated with most computer programs, is that they alternate between CPU cycles and I/O cycles. For the portion of the time required for CPU cycles, the process is being executed; i.e. is occupying the CPU. During the time required for I/O cycles, the process is not using the processor. Instead, it is either waiting to perform Input/Output, or is actually performing Input/Output. An example of this is the reading from or writing to a file on disk. Prior to the advent of multiprogramming, computers operated as single-user systems. Users of such systems quickly became aware that for much of the time that a computer was allocated to a single user, the processor was idle; when the user was entering information or debugging programs for example. Computer scientists observed that overall performance of the machine could be improved by letting a different process use the processor whenever one process was waiting for input/output. In a uni-programming system, if N users were to execute programs with individual execution times of t_1 , t_2, ..., t_N, then the total time, t_{uni} , to service the N processes (consecutively) of all N users would be:

$$t_{uni} = t_1 + t_2 + ... + t_N$$

However, because each process consumes both CPU cycles and I/O cycles, the time which each process actually uses the CPU is a very small fraction of the total execution time for the process. So, for process i:

$$t_{i\ (processor)} \ll t_{i\ (execution)}$$

where

$t_{i\ (processor)}$ is the time process *i* spends using the CPU and $t_{i\ (execution)}$ is the total execution time for the process; i.e. the time for CPU cycles plus I/O cycles to be carried out (executed) until completion of the process. In fact, usually the sum of all the processor time, used by N processes, rarely exceeds a small fraction of the time to execute any one of the processes;

$$\sum_{j=1}^{N} t_{j\ (\mathrm{processor})} < t_{i\ (\mathrm{execution})}$$

Therefore, in uni-programming systems, the processor lay idle for a considerable proportion of the time. To overcome this inefficiency, multiprogramming is now implemented in modern operating systems such as Linux, UNIX and Microsoft

Windows. This enables the processor to switch from one process, X, to another, Y, whenever X is involved in the I/O phase of its execution. Since the processing time is much less than a single job's runtime, the total time to service all N users with a multiprogramming system can be reduced to approximately:

$$t_{multi} = \max(t_1 , t_2 , ..., t_N)$$

Process Creation

Operating systems need some ways to create processes. In a very simple system designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is needed to create and terminate processes as needed during operation. There are four principal events that cause a process to be created:

- System initialization.
- Execution of process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

When an operating system is booted, typically several processes are created. Some of these are foreground processes, that interact with a (human) user and perform work for them. Others are background processes, which are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming e-mails, sleeping most of the day but suddenly springing to life when an incoming e-mail arrives. Another background process may be designed to accept an incoming request for web pages hosted on the machine, waking up when a request arrives to service that request.

Process creation in UNIX and Linux are done through fork() or clone() system calls. There are several steps involved in process creation. The first step is the validation of whether the parent process has sufficient authorization to create a process. Upon successful validation, the parent process is copied almost entirely, with changes only to the unique process id, parent process, and user-space. Each new process gets its own user space.

Process Termination

There are many reasons for process termination:

- Batch job issues halt instruction
- User logs off
- Process executes a service request to terminate

- Error and fault conditions
- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation; for example: attempted access of (non-existent) 11th element of a 10-element array
- Protection error; for example: attempted write to read-only file
- Arithmetic error; for example: attempted division by zero
- Time overrun; for example: process waited longer than a specified maximum for an event
- I/O failure
- Invalid instruction; for example: when a process tries to execute data (text)
- Privileged instruction
- Data misuse
- Operating system intervention; for example: to resolve a deadlock
- Parent terminates so child processes terminate (cascading termination)
- Parent request

Two-State Process Management Model

The operating system's principal responsibility is in controlling the execution of processes. This includes determining the interleaving pattern for execution and allocation of resources to processes. One part of designing an OS is to describe the behaviour that we would like each process to exhibit. The simplest model is based on the fact that a process is either being executed by a processor or it is not. Thus, a process may be considered to be in one of two states, RUNNING or NOT RUNNING. When the operating system creates a new process, that process is initially labelled as NOT RUNNING, and is placed into a queue in the system in the NOT RUNNING state. The process (or some portion of it) then exists in main memory, and it waits in the queue for an opportunity to be executed. After some period of time, the currently RUNNING process will be interrupted, and moved from the RUNNING state to the NOT RUNNING state, making the processor available for a different process. The dispatch portion of the OS will then select, from the queue of NOT RUNNING processes, one of the waiting processes to transfer to the processor. The chosen process is then relabelled from a NOT RUNNING state to a RUNNING state, and its execution is either begun if it is a new process, or is resumed if it is a process which was interrupted at an earlier time.

From this model we can identify some design elements of the OS:

- The need to represent, and keep track of each process.

- The state of a process.
- The queuing of NON-RUNNING processes

Three-state process management model

Although the two-state process management model is a perfectly valid design for an operating system, the absence of a BLOCKED state means that the processor lies idle when the active process changes from CPU cycles to I/O cycles. This design does not make efficient use of the processor. The three-state process management model is designed to overcome this problem, by introducing a new state called the BLOCKED state. This state describes any process which is waiting for an I/O event to take place. In this case, an I/O event can mean the use of some device or a signal from another process. The three states in this model are:

- RUNNING: The process that is currently being executed.
- READY: A process that is queuing and prepared to execute when given the opportunity.
- BLOCKED: A process that cannot execute until some event occurs, such as the completion of an I/O operation.

At any instant, a process is in one and only one of the three states. For a single processor computer, only one process can be in the RUNNING state at any one instant. There can be many processes in the READY and BLOCKED states, and each of these states will have an associated queue for processes.

Processes entering the system must go initially into the READY state, processes can only enter the RUNNING state via the READY state. Processes normally leave the system from the RUNNING state. For each of the three states, the process occupies space in main memory. While the reason for most transitions from one state to another might be obvious, some may not be so clear.

- RUNNING → READY The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution; i.e. time-out occurs. Other reasons can be the imposition of priority levels as determined by the scheduling policy used for the Low Level Scheduler, and the arrival of a higher priority process into the READY state.
- RUNNING → BLOCKED A process is put into the BLOCKED state if it requests something for which it must wait. A request to the OS is usually in the form of a system call, (i.e. a call from the running process to a function that is part of the OS code). For example, requesting a file from disk or a saving a section of code or data from memory to a file on disk