



**NATIONAL OPEN UNIVERSITY OF NIGERIA**

**SCHOOL OF SCIENCE AND TECHNOLOGY**

**COURSE CODE: CIT752**

**COURSE TITLE: OPERATING SYSTEM (OS) CONCEPTS**

**COURSE CODE**

CIT752

**COURSE TITLE**

Operating System (OS) Concepts

**COURSE DEVELOPER**

Dr. C. K. Ayo

Department of Computer & Information Sciences

Covenant University

Ota – Nigeria.

**COURSE EDITOR**

**PROGRAMME LEADER**

**COURSE COORDINATOR**

<b>CONTENTS</b>	<b>PAGE</b>
Introduction.....	1
What You will Learn in this Course.....	1
Course Aims.....	1
Course Objectives.....	1
Working through this Course.....	2
The Course Material.....	2
Study Units.....	2
Presentation Schedule.....	3
Assessment.....	3
Tutor-Marked Assignment.....	4
Final Examination and Grading.....	4
Course Marking Scheme.....	4
Facilitators/Tutors and Tutorials.....	5
Summary.....	7

## **Introduction**

The Operating System (OS) Concept is a two (2) Credit Unit, Second Semester Course for Postgraduate Students in Information Technology (IT).

The Course gives a general overview of an Operating System as a monitor, Supervisor and Resource Manager. Therefore, the course teaches the OS design, its usefulness in resource scheduling and management. A working knowledge of the operating system is very important to all would-be IT practitioners: Programmers and Engineers alike.

## **What You will Learn in this Course**

The course is broken down into units and a course guide. The course guide tells you briefly what the course is about, what course materials you will be using and how you can work with these materials. In addition, it suggests some general guidelines for the amount of time you are likely to spend to each unit of the course in order to complete it successfully.

It gives you guidance in respect of your Tutor-Marked Assignment which will be made available in the assignment file. There will be regular tutorial classes that are related to the course. It is advisable for you to attend these tutorial sessions. The course will prepare you for the challenges you will meet in the field as a software developer and or Systems Engineer.

## **Course Aim**

The aim of the course is to introduce the students to the concepts of operating Systems (OS) as a guide towards understanding their design and implementation as well as their roles in resource management.

## **Course Objectives**

Each unit of this course has a set of stated objectives design to serve as a guide for the students on what to expect in each of them. Consequently, the student is advised to go through the objectives before and after each unit to ascertain that all expectations are met. However, in line with the stated aim of this course, below are some specific objectives which the student must achieve at the end of the course. They include among other things to:

- expose the students to the different types and components of operating system.
- introduce the students Operating System as a resource manager (process, processor, memory and I/O management).
- expose the students to the different types scheduling algorithms and policies.
- expose the students to the different challenges of concurrent and cooperation processes (such as Race condition, Deadlock etc.) and their solutions
- expose the students to the different debugging methods.
- expose the students to the different memory management schemes.
- expose the students to the different file attributes, protection and security mechanisms.

## **Working through this Course**

To complete this course you are required to read each study unit, read the textbooks and read other materials which may be provided by the National Open University of Nigeria.

Each unit contains self-assessment exercises and at certain points in the course you would be required to submit assignments for assessment purposes. At the end of the course there is a final examination. The course should take you about a total of 17 weeks to complete. Below you will find listed all the components of the course, what you have to do and how you should allocate your time to each unit in order to complete the course on time and successfully.

This course entails that you spend a lot of time to read. I would advice that you avail yourself the opportunity of attending the tutorial sessions where you have the opportunity to rub mind with your colleagues.

### **The Course Materials**

The main components of the course include:

1. The Course Guide
2. Study Units
3. Reference/Further Readings
4. Assignments
5. Presentation Schedule

### **Study Unit**

#### **MODULE 1: BASIC CONCEPTS IN OPERATING SYSTEMS**

- Unit 1: Operating Systems Functions and Services
- Unit 2: Process Concepts
- Unit 3: Process Creation

#### **MODULE 2: SCHEDULING, THREADS AND SYNCHRONIZATION**

- Unit 1: Dispatching
- Unit 2: Threads
- Unit 3: Synchronization

#### **MODULE 3: COMMUNICATION, DEADLOCKS AND INTERRUPTS**

- Unit 1: Messages
- Unit 2: Deadlocks
- Unit 3: Interrupts

#### **MODULE 4: DEGUGGING AND MEMORY MANAGEMENT**

- Unit 1: OS Debugging Strategies
- Unit 2: Memory Allocation Techniques
- Unit 3: Virtual Memory

#### **MODULE 5: DEVICE AND FILE MANAGEMENT**

- Unit 1: Input/Output Device Management
- Unit 2: File Concepts
- Unit 3: Protection

Each unit consists of one or two weeks' work and include an introduction, objectives, reading materials, exercises, conclusion, summary Tutor Marked Assignments (TMAs), references and other resources. The unit directs you to work on exercises related to the required reading. In general, these exercises test you on the materials you have just covered or require you to apply it in some way and thereby assist you to evaluate your progress and to reinforce your comprehension of the material. Together with TMAs, these exercises will help you in achieving the stated learning objectives of the individual units and of the course as a whole.

### **Presentation Schedule**

Your course materials have important dates for the early and timely completion and submission of your TMAs and attending tutorials. You should remember that you are required to submit all your assignments by the stipulated time and date. You should guard against falling behind in your work.

### **Assessment**

There are three aspects to the assessment of the course. The first is made up of self-assessment exercises, the second consists of the tutor-marked assignments and the third is the written examination/end of course examination.

You are advised to do the exercises. In tackling the assignments, you are expected to apply information, knowledge and techniques you gathered during the course. The assignments must be submitted to your facilitator for formal assessment in accordance with the deadlines stated in the presentation schedule and the assignment file. The work you submit to your tutor for assessment will account for 30% of your total course work. At the end of the course you will need to sit for a final or end of course examination of about three hour duration. This examination will count for 70% of your total course mark.

### **Tutor-Marked Assignment**

The TMA is a continuous assessment component of the course. It accounts for 30% of the total score. You will be given four (4) TMAs to answer. Three of these must be answered before you are allowed to sit for the end of course examination. The TMAs would be given to you by your facilitator and returned after you have done the assignment. Assignment questions for the units in this course are contained in the assignment file. You will be able to complete your assignment from the information and material contained in your reading, references and study units. However, it is desirable in all degree level of education to demonstrate that you have read and researched more into your references, which will give you a wider view point and may provide you with a deeper understanding of the subject.

Make sure that each assignment reaches your facilitator on or before the deadline given in the presentation schedule and assignment file. If for any reason you cannot complete your work on time, contact your facilitator before the assignment is due to discuss the possibility of an extension. Extension will not be granted after the due date unless there are exceptional circumstances.

### **Final Examination and Grading**

The end of course examination for Operating System Concept will be for about 2 hours and it has a value of 70% of the total course work. The examination will consist of questions, which will reflect the type of self-testing, practice exercise and tutor-marked assignment problems you have previously encountered. All areas of the course will be assessed.

Use the time between finishing the last unit and sitting for the examination to revise the whole course. You might find it useful to review your self-test, TMAs and comments on them before the examination. The end of course examination covers information from all parts of the course.

### **Course Marking Scheme**

<b>Assignment</b>	<b>Marks</b>
Assignments 1 - 4	Four assignments, best three marks of the four count at 10% each – 30% of course marks.
End of course examination	70% of overall course marks.

Total	100% of course materials.
-------	---------------------------

### **Facilitators/Tutors and Tutorials**

There are 16 hours of tutorials provided in support of this course. You will be notified of the dates, times and locations of these tutorials as well as the name and phone number of your facilitator, as soon as you are allocated a tutorial group.

Your facilitator will mark and comment on your assignments, keep a close watch on your progress and any difficulties you might face and provide assistance to you during the course. You are expected to mail your Tutor Marked Assignment to your facilitator before the schedule date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not delay to contact your facilitator by telephone or e-mail if you need assistance.

The following might be circumstances in which you would find assistance necessary, hence you would have to contact your facilitator if:

- You do not understand any part of the study or the assigned readings.
- You have difficulty with the self-tests
- You have a question or problem with an assignment or with the grading of an assignment.

You should endeavour to attend the tutorials. This is the only chance to have face-to-face contact with your course facilitator and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study.

To gain much benefit from course tutorials prepare a question list before attending them. You will learn a lot by participating actively in discussions.

### **Summary**

The Course OS Concepts is primarily intended to provide students with the requisite knowledge of operation systems, their functions, and types. The OS being the primary tool for all Programmers and Computer Engineers alike, the course introduces the students to resource management and scheduling issues as well as the associated challenges and solutions among other things. At the end of the course the student will be able to answer the following questions:

- What is an OS?
- Explain three types of system calls.
- Explain the process life cycle.
- Explain the various scheduling algorithms and policies.
- Explain the term Race Condition and Deadlock. What are the solutions to the problems?
- Explain the four classes of Interrupt.
- Explain the various Memory Management schemes
- What is segmentation? How can this problem be solved?
- Explain the disk allocation and scheduling methods.
- Explain the various device protection mechanisms.

Etc.

Best of Luck.



**COURSE CODE**

CIT752

**COURSE TITLE**

Operating System (OS) Concepts

**COURSE DEVELOPER**

Dr. C. K. Ayo

Department of Computer & Information Sciences

Covenant University

Ota – Nigeria.

**COURSE EDITOR**

**PROGRAMME LEADER**

**COURSE COORDINATOR**

# **CIT752: OPERATING SYSTEM CONCEPTS**

## **MODULE 1: BASIC CONCEPTS IN OPERATING SYSTEMS**

### **UNIT 1: OPERATING SYSTEMS FUNCTIONS AND SERVICES**

	<b>Page</b>
1.0	Introduction
2.0	Objectives
3.0	Meaning of Operating Systems
3.1	Operating System functions
3.2	Operating System services
3.3	System calls
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### **1.0 Introduction**

The operating system is the software that controls the allocation and usage of hardware resources such as memory, CPU time, disk space, and input and output devices. The unit examines the meaning, function and services provided by the operating system.

#### **2.0 Objectives**

At the end of this unit, the reader should be able to:

- (a) define the operating system
- (b) know the functions of the operating system
- (c) know the services provided by the operating system
- (d) describe system calls

#### **3.0 Meaning of Operating systems**

An operating system (commonly abbreviated OS or O/S) is the software component of a computer system that is responsible for the management and coordination of activities and the sharing of the resources of the computer. The operating system acts as a host for applications that are run on the machine. As a host, one of the purposes of an operating system is to handle the details of the operation of the hardware. This relieves application programs from having to manage these details and this makes it easier to write applications. Almost all computers, including handheld computers, desktop computers, supercomputers, and even video game consoles, use an operating system of some type. Some of the oldest models may however use an embedded operating system, that may be contained on a compact disk or other data storage device.

Operating systems offer a number of services to application programs and users. Applications access these services through application programming interfaces (APIs) or system calls. By invoking these interfaces, the application can request a service from the operating system, pass parameters, and receive the results of the operation. Users may also interact with the operating system with some kind of software user interface (UI) like typing commands using command line interface (CLI) or using a graphical user interface (GUI, commonly pronounced “gooey”). For hand-held and desktop computers, the user interface is generally considered part of the operating system. On large multi-user systems like Unix and Unix-like systems, the user interface is generally implemented as an application program that runs outside the operating system. (Whether the user interface should be included as part of the operating system is a point of contention).

Common contemporary operating systems include Microsoft Windows, Mac OS, Linux and Solaris. Microsoft Windows has a significant majority of the market share in the desktop and notebook computer markets, while servers generally run on Linux or other Unix-like systems. Embedded device markets are split amongst several operating systems.

### **Types of Operating Systems**

Operating systems are classified based on the number of users into single user, multi-user and network operating systems.

#### **a. Single-user operating systems**

Single-user operating systems are designed to run on stand-alone microcomputers or PCs. Their primary function is to manage the resources of that PC and control its activities in its entirety. Examples are: MS-DOS, PC -DOS, CP/M, Windows 3.1 e.t.c

#### **b. Multi-User Operating System**

The multi-user operating systems were prevalent during the era of centralized processing of jobs on a host (CPU) to which other terminals (Dummy terminals) are connected. These operating systems run on minicomputer systems. Examples are: PC-MOS, and UNIX.

#### **c. Network Operating System**

Network operating systems are primarily concerned with communication among the various systems and peripherals on the network; as well as the flow of data across the network and sharing of resources on the network.

The various examples are:

- i. Windows 3.11 (workgroup), 95,98, NT, 2000 pro and server, XP, 2003, etc.
- ii. Novell Netware, UNIX and Linux

In computer science, a virtual machine is software that creates a virtualized environment between the computer platform and its operating system, so that the end user can operate software on an abstract machine. Figure 1 shows a layered structure of OS.



Figure 1.1: A layered structure showing where operating system is located

Source: Wikipedia, the free encyclopedia, [http://en.wikipedia.org/wiki/Operating\\_system](http://en.wikipedia.org/wiki/Operating_system)

From the layered structured, the OS is an intermediary between the user and the hardware. In essence, it simplifies the use of the hardware for the user.

### 3.1 Operating System functions

An operating system must be able to perform the following functions, amongst others:

- Job scheduling
- Job control language interpretation
- Error handling
- Input/Output (I/O) handling
- Interrupt handling
- Scheduling
- Memory management
- Resource control
- Protection
- Inter process communication

#### The role of the operating system Shell and Kernel

The kernel is the core of the OS (see Figure 1.2). It provides most of the functionality that the rest of the system needs in order to function. The user needs to be able to “talk” to the kernel in order to send commands to the OS. However, the kernel does not speak a language that is anything like any human language.

Attempting to communicate with the kernel in its own language would be extremely complicated and frustrating. This is where the shell comes in. The shell is basically an interpreter that

understands commands in something resembling common English and translates those commands into a language the kernel understands. The primary role of the shell is to provide an interface through which the user can interact with the kernel. The kernel also accepts messages from the shell and displays them in a language the user can understand.

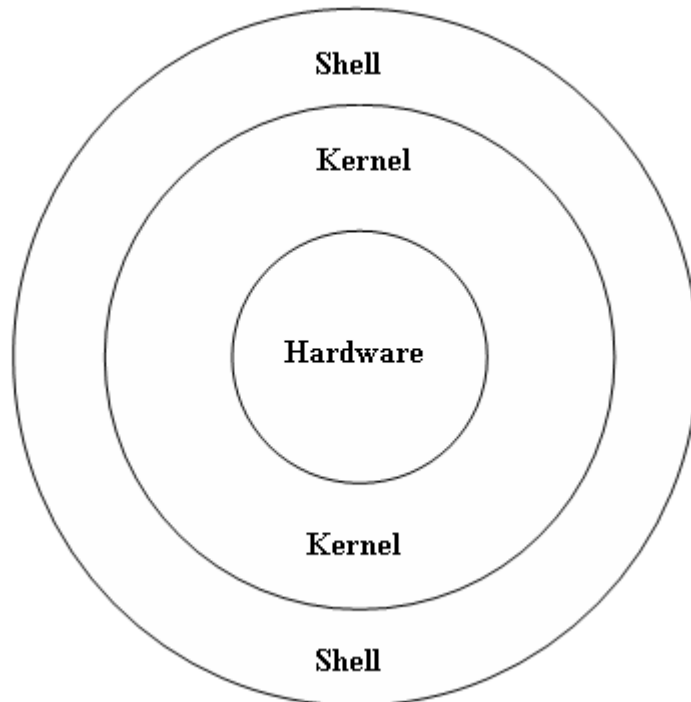


Figure 1.2: The Shell, Kernel and System Hardware using the seed metaphor  
Source: SAMS Teach Yourself FreeBSD in 24 hours by Michael Urban and Brian Tiemann, 2003

The hierarchical view of figure 1.2 is shown in figure 1.3 below.

The user types commands to the shell, which communicates with the kernel, and the kernel in turn communicates with the hardware. The kernel is the heart of the operating system. The shell consists of command interpreters or programming languages.

Commands consist of a wide range of commands the user types or issues for the day-to-day running on the computer.

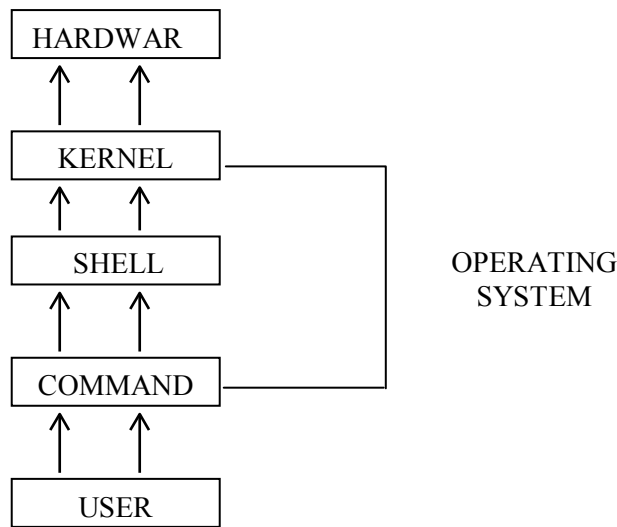


Figure 1.3: Operating System, Hardware and User Relationship

### 3.2 Operating system services

The following set of operating-system services provides functions that are helpful to the user:

- **User interface** - Almost all operating systems have a user interface (UI).
  - ▶ It varies between Command line interface (CLI), and Graphics User Interface (GUI).
- **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error).
- **Input/Output operations** - A running program may require input/output (I/O) operations, which may involve a file or an I/O device.
- **File-system manipulation** - The file system is of particular interest. Obviously, programs need to read and write files and directories (folders), create and delete them, search them, list file information and permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network.
  - ▶ Communications may be via shared memory or through message passing (packets moved by the OS).
- **Error detection** – OS needs to be constantly aware of possible errors.
  - ▶ Errors occur in the CPU and memory hardware, in I/O devices, in user program.
  - ▶ For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
  - ▶ Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system.

The following set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing:

- **Resource allocation** - When multiple users or multiple jobs are running concurrently, resources must be allocated to each of them.
  - ▶ Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
- **Accounting** - To keep track of which users use how much of a device and what kinds of computer resources.
- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want a controlled use of that information, concurrent processes should not interfere with each other.
  - ▶ Protection involves ensuring that all access to system resources is controlled.
  - ▶ Security of the system from outsiders requires user authentication, preventing external I/O devices from invalid access attempts.
  - ▶ If a system is to be protected and secured, precautions must be instituted throughout. A chain is only as strong as its weakest link.

### **User Operating System Interface - CLI**

CLI allows direct command entry

- ▶ Sometimes implemented in the kernel and sometimes by systems program.
- ▶ Sometimes multiple flavors are implemented – shells
- ▶ Primarily fetches a command from user and executes it.

### **User Operating System Interface – GUI**

- User-friendly desktop metaphor interface.
  - ▶ Usually mouse, keyboard, and monitor are used.
  - ▶ Icons are used to represent files, programs, actions, etc.
  - ▶ Placing mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)).
- Many systems now include both CLI and GUI interfaces
  - ▶ Microsoft Windows include both GUI with CLI “command” shell.
  - ▶ Apple Mac OS X as “Aqua” GUI interface with UNIX kernel underneath and shells are available.

### **3.3 System calls**

System calls provide an interface between a running program (process) and the operating system. System calls allow user-level processes to request some services from the operating system which the process itself is not allowed to do. In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on behalf of user-level process. It is because of the critical nature of operations that the

operating system itself does them every time they are needed. For example, for I/O, a process involves a system call telling the operating system to read from, or to write to a particular area and this request is satisfied by the operating system.

A system call is a request made by any program to the kernel for performing tasks, picked from a predefined set, which the said program does not have the required permissions to execute in its own flow of execution. System calls provide the interface between a process and the kernel. These calls are generally available as assembly language instructions and they are usually listed in the various manuals used by the assembly-language programmers. Most operations interacting with the system require permissions not available to a user level process, i.e. any I/O operation performed with any arbitrary device present on the system or any form of communication with other processes requires the use of system calls.

An application program interface (API) is a set of functions that can be called from an application program to access features of another program. API is the software interface to system services or software libraries. An API can consist of classes, function calls, subroutine calls, descriptive tags, etc. Example of System Calls is contained in Figure 1.4.

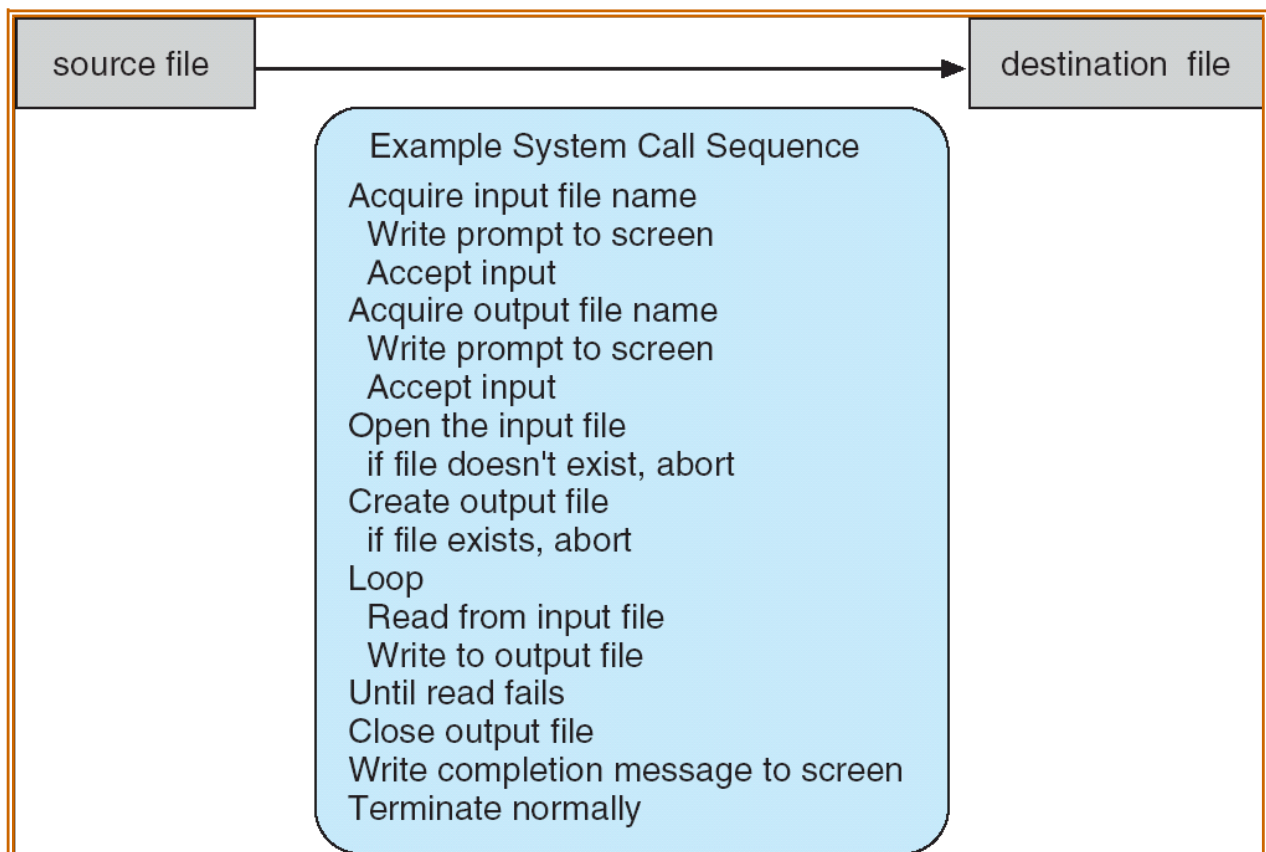


Figure 1.4.: Systems call sequence to copy the contents of one file to another file.

Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432



## **Types of system calls and their purposes**

### **Process control:**

This type of system calls are used to control processes. Some examples are: end, abort, load, execute, create process, terminate process; get process attributes, set process attributes, wait for time, wait event, signal event, allocate and free memory.

### **File management:**

This type of system calls are used to manage files. Some examples are: create file(s), delete file(s), open, close, read, write, reposition, get file attributes, set file attributes.

### **Device management:**

This type of system calls are used to manage devices. Some examples are: request device, release device, read, write, reposition, get device attributes, set device attributes, logically attach or detach devices.

### **Information maintenance:**

This type of system calls are used to get information from the computer. Some examples are: get time or date, set time or date; get system data, set system data; get process, file, or device attributes; set process, file, or device attributes.

### **Communication:**

This type of systems calls are used for communication. Some examples are: create, delete communication connection; send, receive messages; and transfer status information.

### **Activity A**

What is an operating system?

Describe the functions and services provided by an operating system.

Mention and explain three (3) types of system calls.

## **4.0 Conclusion**

A number of services and functions are offered by operating systems to application programs and users. Applications access these services through application programming interfaces (APIs) or system calls. Several types of system calls exist for controlling processes.

## **5.0 Summary**

In this unit, we have learnt that:

- (a) operating system acts as a host for applications that are run on the machine.
- (b) operating system comes with two interfaces - CLI and GUI.
- (c) operating system provides system call services such as file management, communication, etc.

## **6.0 Tutor Marked Assignment**

1. What are system calls?
2. Describe any two different types of system calls in operating systems.

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Ayo C. K (2009): *Information Systems and Technologies*, McKay Educational Series, Lagos.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Hennessy, J. and Patterson D. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2002.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

Sams teach yourself FreeBSD in 24 hours by Michael Urban and Brian Tiemann, 2003

## MODULE 1: BASIC CONCEPTS IN OPERATING SYSTEMS

### UNIT 2: PROCESS CONCEPTS

	Page
1.0	Introduction
2.0	Objectives
3.0	Process concept
3.1	Process State
3.2	Process Control Block (PCB)
3.3	Context Switch
4.0	Conclusion
5.0	Summary
6.0	Tutor marked Assignment
7.0	Further Reading and Other Resources

#### 1.0 Introduction

The focus of a traditional operating system is the management of processes. Each process is at any time in one of a number of executing states, including: Ready, Running and Blocked. The operating system keeps track of these execution states and manages the movement of processes among the states. The unit examines the status of a process during execution.

#### 2.0 Objectives

At the end of this unit, the reader should be able to:

- (a) define process and process concept
- (b) describe the process state and the PCB
- (c) describe the context switch

#### 3.0 Process Concept

The compositions of a process as used in operating systems are highlighted as follows:

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms job and process almost interchangeably
  - A process is a program in execution. It is a unit of work within the system. Its execution must progress in a sequential fashion. In other words, Process executes instructions sequentially, one at a time, until completion.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- A process includes:
  - program counter
  - stack
  - data section

- Process termination requires reclaim of any reusable resources
- Single-threaded process has one program counter specifying location of next instruction to execute
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads
- Program is a passive entity, process is an active entity.

### 3.1 Process State

- As a process executes, it changes state
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

#### Process Scheduling

Once the job scheduler has moved a job from new to ready state, it creates one or more processes for this job. The process scheduling module decides which processes in the ready state get the processor, when, and for how long?

Specifically, the process scheduler performs the following tasks:

1. Keeping track of the status of the process (all processes are either running, ready, or blocked). The module that performs this function has been called the traffic controller.
2. Deciding which process gets a processor and for how long. This is performed by the processor scheduler.
3. Allocating a processor to a process. This requires resetting of processor registers to correspond to the process' correct state. This task is performed by the traffic controller.
4. Deallocating a processor, such as when the running process exceeds its current quantum or must wait for an I/O completion. This requires that all processor state registers be saved to allow future reallocation. This task is performed by the traffic controller.

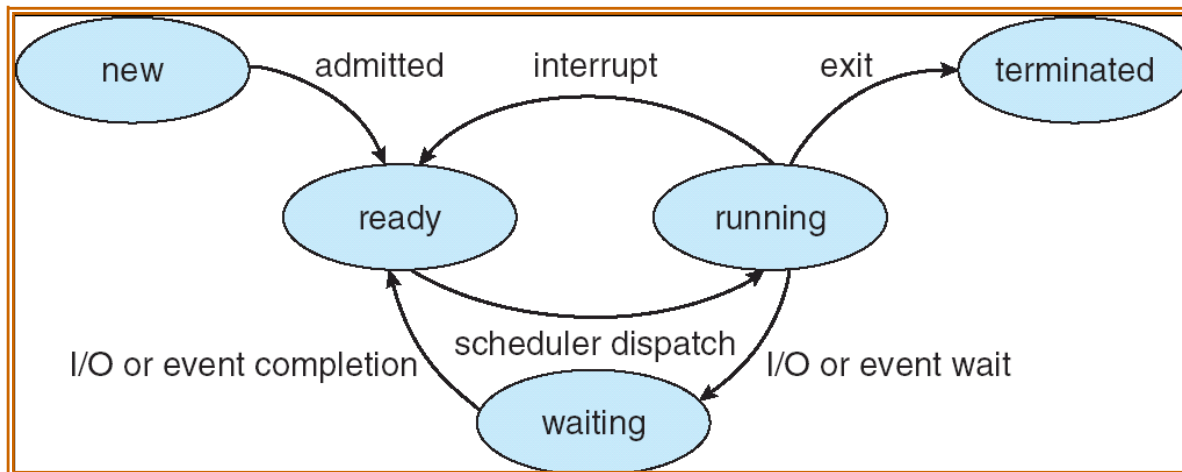


Figure 2.1: Diagram of Process State

Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Figure 1 indicates the types of events that lead to each state transition for a process. The possible transitions are as follows: Null → New, New → Ready, Ready → Running, Running → Ready, Running → Waiting, Waiting → Ready, and Running → terminated.

The above life Cycle can be explained as follows. A user submits his job to the system (new job). The job consists of several decks of programs preceded by job control cards. The job control cards pass information to the operating system as to what resources the job will need. The spooling routine reads the job and places it onto a disk. To find space for the machine readable copy of the users job, the spooling routine must call information management, which keeps track of all information and available space in the system. At a particular point the job-scheduling routine scans all the spooled files on the disc and picks a job to be admitted into the system. In picking a job, the job scheduler will call memory management to see if there is sufficient main memory available. The job scheduler equally determines the device requirements from the user's job cards. Once the job scheduler has decided which job to be admitted, the memory management is called to allocate the necessary main storage. The job is then loaded into memory and the process is then ready to run (Ready State). When the processor becomes free; the process scheduler scans the list of ready processes, chooses a process, and assigns a processor to it (running state). If the running process requests access to information (e.g. read a file), information management would call device management to initiate the reading of the file. Device management would initiate the I/O operation and then the process management to indicate that the process requesting the file is waiting for I/O operation (wait state). That is, the process requesting for I/O operation is taken from the running state to the wait state.

When the I/O is completed, the I/O hardware sends signal to the traffic controller of process management, which places the process back into the ready state. If the process should complete its computation when it is run again, then it is placed into terminated state and all allocated resources are freed.

The two arrows between the ready and run states can be explained thus: A process moves from ready to run after it is been assigned a processor to run. This same process is returned back

because the processor time is shared among all the ready processes in the ready state in a round robin fashion. That is, a process is only allowed to run for a quantum number of time and when it has elapsed it is sent back to the ready list if it still needs the processor time to complete operation. By and large, the processes are treated in turn. Another thing that can cause a similar thing prematurely is if a higher priority process job just enters the ready state, then the running process would be stopped automatically and returned to the end of the queue is ready list while the high-priority process takes hold of the processor.

### 3.2 Process Control Block (PCB)

Because of the fact that a process may be in any of the available states, the PCB is maintained for each of them so that the OS can know the state or stage it stopped its previous processing. Information associated with each process is stored in PCB. PCB contains the following:

- Process ID
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory management information
- Accounting information
- I/O status information

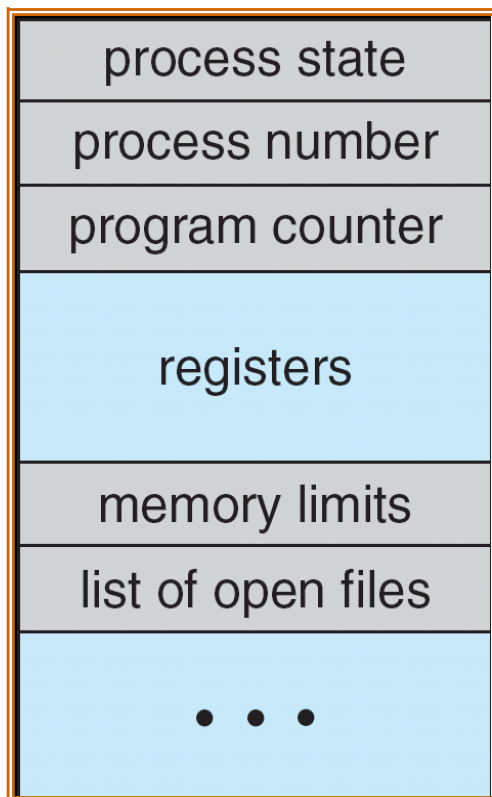


Figure 2.2: Process Control Block (PCB)

Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Figure 2.2 contains a simplified PCB that is created and managed by the operating system. The significant point about the PCB is that it contains sufficient information so that it is possible to interrupt a running process and latter resume execution as if the interruption had not occurred. The PCB is the key tool that enables the OS to support multiple processes and to provide for multiprocessing. When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block, and the state of the process is changed to some other values, such as *waiting* or *ready* described in Figure 2.1.

The process Control Blocks for all the existing processes are kept in a table, called the **Process Block**. Alongside this table, is maintained the three lists, the ready, running and blocked lists.

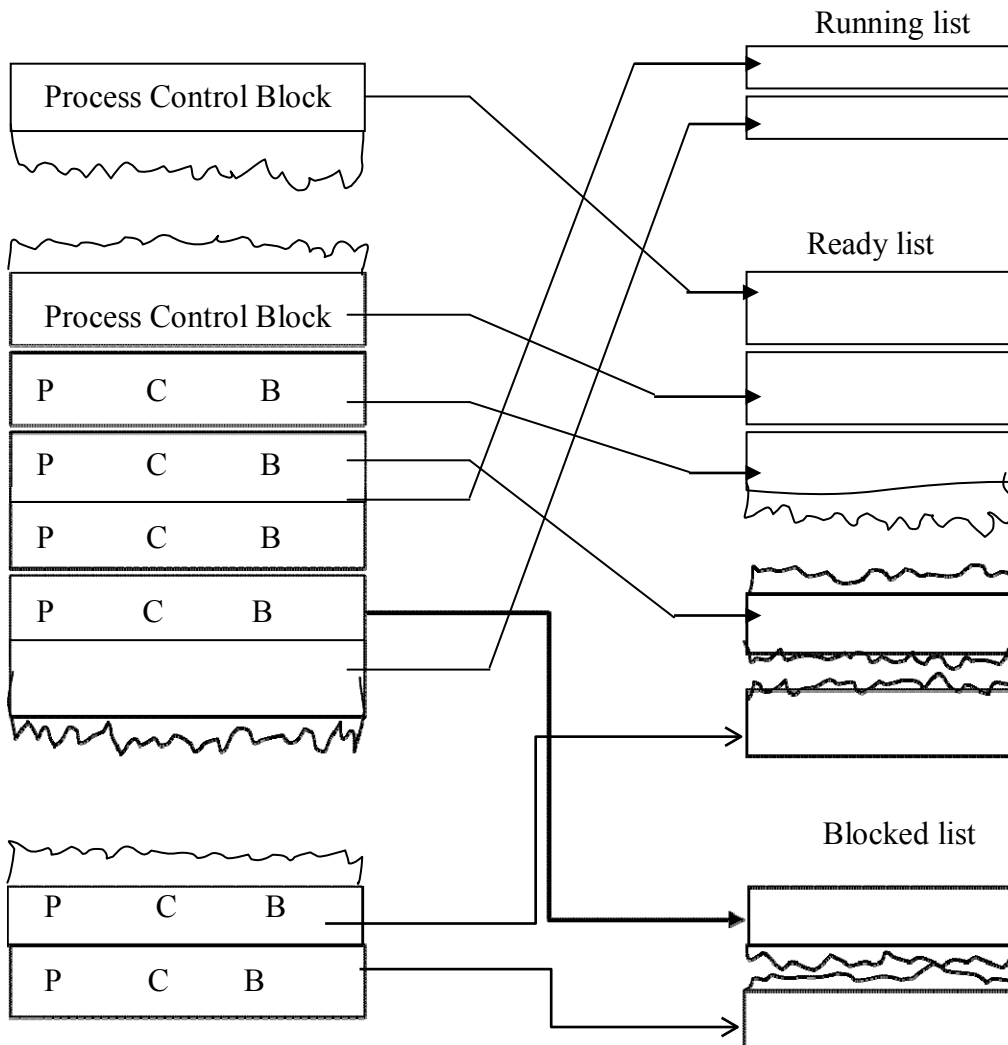


Figure 3.3: Simplified Process Control Block

The entries in these three lists are simply pointers to entries in the process table, it may be noted that the:

**Running List = The number of processors**

The length of the other two lists must be equal to the maximum number of processes that will be allowed to exist at any given time. Even if we have more than one processor (usually, we have only one processor), it is economically sound to share the processors amongst the numerous processes existing at a given time. Hence the problem of Process Management can be redefined as:

1. Keep track of the status of each process.
2. Select processes from the ready list to run.
3. Suspend a running process when it runs out of its allocated time.
4. Coordinate interprocess communication.

Accordingly an OS would have the following procedures to take care of the above situation.

1. **CREATE:** to establish a new process
2. **DESTROY:** to destroy a process
3. **BLOCK:** to block a process
4. **PRE-EMPT:** to pre-empts (i.e. forcibly stop) a process, from running to ready state.
5. **WAKE UP:** to wake up a blocked process, (from blocked to ready).
6. **START PROCESS:** to allocate a processor and start a process executing in it, i.e. running state.

### 3.3 Context Switch

When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process. Context-switch time is additional overhead, the system does no useful work while switching. Switching can be time dependent on hardware support.

The operating system is responsible for the following activities in connection with process management:

- ▶ Creating and deleting both user and system processes
- ▶ Suspending and resuming processes
- ▶ Providing mechanisms for process synchronization
- ▶ Providing mechanisms for process communication
- ▶ Providing mechanisms for deadlock handling

#### Activity A

For each of the following transitions between process states, indicate whether the transition is possible. If it is possible, give an example of one thing that would cause it.

- i. Run → ready
- ii. Run → blocked
- iii. Run → swapped-blocked
- iv. Blocked → run
- v. Run → terminated



#### **4.0 Conclusion**

An operating system executes a variety of programs. As a process executes, it changes state. Information associated with each process is stored in PCB. When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

#### **5.0 Summary**

In this unit, we have learnt that:

- (a) a process is a program in execution.
- (b) the PCB is composed of process ID, process state, program counter, CPU registers and scheduling information, memory management and accounting information, and I/O status information.
- (c) The OS switches from one process to another using context switching mechanism.

#### **6.0 Tutor marked Assignment**

With the aid of a diagram, enumerate the components and functions of a Process Control Block (PCB)

#### **7.0 Further Reading and Other Resources**

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

## MODULE 1: BASIC CONCEPTS IN OPERATING SYSTEMS

### UNIT 3: PROCESSES CREATION

Page

1.0	Introduction
2.0	Objectives
3.0	Process creation
	3.1 Process termination
	3.2 Cooperating processes
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### 1.0 Introduction

A process may be thought of as a sequence of actions, performed by executing a sequence of instructions (a program), whose net result is the provision of some system functions. We can extend the concept to include the provision of user functions as well as system functions, so that the execution of a user program is also called a process. This unit describes how processes are created, terminated and how they cooperate.

#### 2.0 Objectives

At the end of this unit, the reader should be able to:

- (a) Know how processes are created.
- (b) Know how processes get terminated after execution.
- (c) Know how processes cooperate.

#### 3.0 Process creation

When a new process is to be added to those currently being managed, the OS builds the data structures that are used to manage the process and allocates address in main memory to the process. These actions constitute the creation of a new process.

In process creation, parents process create children processes, which in turn can create other processes forming a tree of processes. Resources are shared among parents and children and parents and children execute jobs concurrently.

#### 3.1 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its parent process (via the `wait()` system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Any computer system must provide a means for a process to indicate its completion. A batch job should include a Halt instruction or an explicit operating system service call for termination. In the former case, the Halt instruction will generate an interrupt to alert the OS that a process has completed. For an interactive application, the action of the user will indicate when the process is completed. For example, in a time-sharing system, the process for a particular user is to be terminated when the user logs off or turns off his or her terminal. On a personal computer or workstation, a user may quit an application (e.g. word processing or spreadsheet). All of these actions ultimately result in a service request to the OS to terminate the requesting process.

### 3.2 Cooperating processes

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data (temporary or persistent) with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- *Information sharing:* Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.
- *Computation speedup:* If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- *Modularity:* We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as shall be discussed later in the lecture.
- *Convenience:* Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel. Concurrent execution that requires cooperation among the processes requires mechanisms to allow processes to communicate with one another and to synchronize their actions.

To illustrate the concept of cooperating processes, let us consider the producer consumer problem, which is a common paradigm for cooperating processes. A partial pseudocode is shown in Figure 1. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer-consumer problem also provides a useful metaphor for the client-server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a file server produces (that is, provides) a file which is consumed (that is, read) by the client requesting the file. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce one item while the consumer is consuming another item.

```
public interface Buffer
{
    // producers call this method
    public abstract void insert(Object item);

    // consumers call this method
    public abstract Object remove();
}
```

Figure 1. Interface for Buffer implementation.

The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced. Solutions to the producer-consumer problem may implement the Buffer interface shown in Figure 1. The producer process invokes the insert() method when it wishes to enter an item in the buffer, and the consumer calls the remove() method when it wants to consume an item from the buffer.

### Issues for Cooperating Processes

- Race condition- What does it mean ?
  - ▶ A race condition occurs when the scheduling of two processes is so critical that the various orders of scheduling them result in different computations. This results from the explicit or implicit sharing of data or resources among two or more processes.
- Critical regions
  - ▶ A critical region is a portion of a process that accesses shared memory. A section of code, or collection of operations, in which only one process may be executing at a given time.
- Mutual exclusion
  - ▶ Mutual exclusion is a mechanism to enforce that only one process at a time is allowed into a critical region.

Mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded).

## What is Race Condition?

The situation where several processes access and manipulate the same data concurrently, and where the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. Such situations occur frequently in operating systems as different parts of the system manipulate resources, and we do not want the changes to interfere with one another. Unexpected interference can cause data corruption and systems crashes.

To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable count. To make such a guarantee, we require some form of synchronization of the process.

## Cooperating Processes Approach

Any approach to process cooperation requires that no two processes may be simultaneously inside their critical regions. Other requirements include:

- ▶ No assumptions may be made about speeds or the number of CPUs
- ▶ No process running outside of its critical region may block other processes
- ▶ No process should have to wait forever to enter its critical region

There are many ways to achieve mutual exclusion. Most involve some sort of locking mechanism: prevent someone from doing something. Three elements of locking are:

1. Must lock before using.
2. Must unlock when done.
3. Must wait if locked.

## Activity A

Briefly explain the term cooperating processes.

## 4.0 Conclusion

Parent process creates children processes, which, in turn can create other processes, forming a tree of processes. Some operating system do not allow child to continue if its parent terminates. Cooperating process can affect or be affected by the execution of another process. Any approach to process cooperation requires that – No two processes may be simultaneously inside their critical regions. There are many ways to achieve mutual exclusion. Most involve some sort of locking mechanism: prevent someone from doing something.

## 5.0 Summary

At the end of this unit, we have learnt that:

- (a) Processes can be are created.
- (b) Processes get terminated after execution.
- (c) Processes can cooperate and the issues that may arise thereof.

## 6.0 Tutor marked Assignment

What are some of the common events responsible for “process creation”? What constitutes “process termination”? Discuss any five conditions for “process termination”.

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Goodheart B. and Cox J. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

# **CIT752: OPERATING SYSTEM CONCEPTS**

## **MODULE 2: SCHEDULING, THREADS AND SYNCHRONIZATION**

### **UNIT 1: DISPATCHING**

	<b>Page</b>
1.0	Introduction
2.0	Objectives
3.0	Scheduling
3.1	Preemptive and non-preemptive policy
3.2	Algorithms for Short-term Scheduling (STS)/ Dispatcher
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### **1.0 Introduction**

An operating system must allocate computer resources among the potentially competing requirements of multiple processes. In the case of the processor, the resources to be allocated is execution time on the processor and the means of allocation is scheduling. The unit examines the short-term scheduling algorithm.

#### **2.0 Objectives**

At the end of this unit, the reader should be able to:

- (a) Distinguish between the different types of scheduling
- (b) Distinguish between preemptive and non preemptive scheduling policy
- (c) Know the short-term scheduling policies

#### **3.0 Scheduling**

The aim of processor scheduling is to assign processes to be executed by the processor or processes over time in a way that meets system objectives such as response time, throughput and processor efficiency.

##### **There are four types of scheduling:**

**Long-term scheduling:** This involves the decision to add to the pool of processes to be executed. It is also known as high-level scheduling.

**Medium-term scheduling:** This involves the decision to add to the number of processes that are partially or fully in main memory.

**Short-term scheduling:** This involves the decision as to which available process will be executed by the processor. It is also known as low-level scheduling or the dispatcher.

**I/O scheduling:** This involves the decision as to which process's pending I/O requests shall be handled by the available I/O device.

### 3.1 Preemptive and Non-preemptive policy

Each of the low-level scheduling policies described below can be categorized as either preemptive or non-preemptive. In a preemptive scheme, the short-term scheduler (STS) may remove a process from the running state. In a non-preemptive scheme, a process once given a processor will be allowed to run until it terminates or encounters an I/O wait, ie it cannot forcibly loss the processor.

### 3.2 Algorithms for short-term Scheduling / Dispatcher

#### First-Come, First-Served (FCFS)

The simplest scheduling policy is FCFS or First-In, First-Out (FIFO). As each current running process ceases to execute, the oldest process in the ready queue is selected to run. It simply assigns the processor to the process which is first in the READY queue, ie has being waiting for the longest time. This is a non-preemptive scheme, since it is actioned only when the current process relinquishes control. FCFS favours long jobs over short ones. Another problem with FCFS is that if a CPU bound process get the processor, it will run for relatively long period uninterrupted while I/O bound processes will be unable to maintain I/O activity at a high level. FCFS is rarely used on its own but is often employed in conjunction with other methods. FCFS performs much better for long processes than short ones.

For example, let us consider the job schedule in table 2.1, compute the average turnaround time.

Table 2.1

Job No.	Arrival Time	Run Time
1	10.00	2.00hrs
2	10.10	1.00hrs
3.	10.25	0.25hrs

Average turnaround time (P) is computed as:

$$P = (\sum T_i) / n$$

Where

**T<sub>i</sub>** = Turnaround Time

**F<sub>i</sub>** = Finish Time

**A<sub>i</sub>** = Arrival Time

**n** = no. of Jobs

**T<sub>i</sub>** = F<sub>i</sub> – A<sub>i</sub>



Therefore using FIFO, the above job mix would be computed as follows:

Job No.	ARR. TM (Ai)	START Time	FINISH TM. (Fi)	Turnaround Time (Ti)
1.	10.00	10.00	12.00	2.00hrs
2.	10.10	12.00	13.00	2.90hrs
3.	10.25	13.00	13.25	3.00hrs
				7.90hrs

$$P = 7.90/3 = 2.63\text{hrs}$$

### Shortest Job First (SJF)

This is also known as Shortest Job Next (SJN) or Shortest Process Next (SPN). It is another approach to reduce the bias in favour of long processes inherent in FCFS. It is a non-preemptive policy in which the process with the shortest expected processing time is selected next. Thus, a short process will jump to the head of the queue past longer ones. The main disadvantage of this approach is that a long job in the queue may be delayed indefinitely by a succession of smaller jobs arriving in the queue. SJF is more applicable to batch processes since it requires that an estimate of run time be made available which could be supplied in the Job Control Language (JCL).

Using the information in table 2.1, compute P based on SJF.

Job No.	ARR. TM (Ai)	START Time	FINISH TM. (Fi)	Turnaround Time (Ti)
1.	10.00	10.00	12.00	2.00hrs
2.	10.10	12.25	13.25	3.15hrs
3.	10.25	12.00	12.25	2.00hrs
				7.15hrs

$$P = 7.15/3 = 2.38\text{hrs}$$

### Round Robin (RR)

A straight way to reduce the penalty that short jobs suffer with FCFS is to use preemption based on a clock. In the RR scheme, each process has a specified quantum of time to run, and a process is selected to run from the ready queue in FIFO sequence. However, if the process runs beyond the fixed length of time, called the time quantum, it is interrupted and returned to the end of the ready queue. In other words, each active process is given a time slice in rotation. Figure 1 shows a description of round robin.

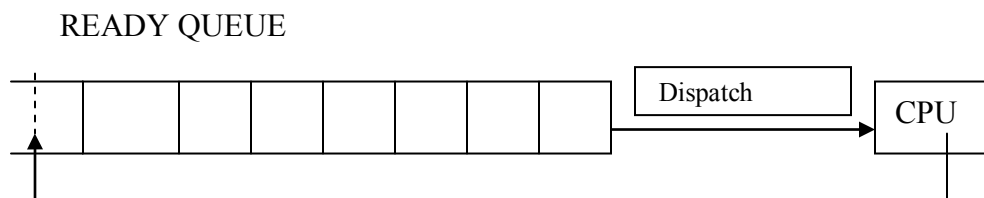


Figure 1: Round Robin Scheduling

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

The timing required by this scheme is obtained by using a hardware timer which generates an interrupt at precise intervals. RR is effective in time sharing environment or transaction-processing system, where it is desirable to provide an acceptable response time for every user and where the processing demands of each user will often be relatively low and sporadic. The RR is preemptive but preemption occurs only at the expiration of the time quantum.

The round robin algorithm ensures that short execution requests will finish before long execution requests. For example, let  $P_1, P_2, P_3$  be three processes having execution time of 0.1, 0.5, 0.4 Seconds. Let the quantum = 0.2 secs and let the queue initially have the ordering  $P_2 (0.5) P_1 (0.1) P_3 (0.4)$ . Where the head of the queue is on the left. Assuming the system has only one processor execution proceeds so that the queue shapes successively as:

$P_1(0.1) P_3(0.4) P_2(0.3)$   
 $P_3(0.4) P_2(0.3)$   
 { $P_1$  Completed execution in 0.1sec}  $P_2(0.3) P_3(0.2)$   
 $P_3(0.2) P_2(0.1)$   
 $P_2(0.1)$   
 { $P_3$  Completed execution 0.2secs}.

It is worth noticing that the three processes completed execution in the order of the increasing execution time requirement even though they were not initially ordered that way.

In the above example, the assignment of priority to the process has not been considered. In such cases, scheduling becomes more complicated. There are various ways of assigning priorities to processes.

1. In some, both execution time and memory requirements are considered in assigning a priority to a process.
2. The importance of the job can get a higher priority to a job.
3. One can attain higher priority by paying for computer time at a higher rate, and so on.

### **Shortest Remaining Time (SRT)**

SRT is a preemptive version of SJF. At the time of dispatching, the shortest queued process say job A will be started. However, if during running of this process, another job arrives whose run time is shorter than job A's remaining run time, then job A will be preempted to allow the new job to start. SRT favours short jobs even more than SJF, since a currently running job could be ousted by a new shorter one. The danger of starvation of long jobs also exist in this scheme. Implementation of SRT requires an estimate of total run time and measurement of elapse run time.

### **Highest Response Ratio Next (HRRN)**

This scheme is derived from the SJF method, modified to reduce SJF bias against long jobs and to avoid the danger of starvation. In effect, HRRN derives a dynamic priority value based on the estimated run time and the incurred waiting time. The priority for each process is calculated using the formula:

Priority,  $P = \frac{\text{timewaiting} + \text{run time}}{\text{Run time}}$

The process with the highest priority value will be selected to run when processes first appeared in the ready queue, the time waiting will be zero and hence  $P = 1$  for all processes. After a short period of waiting, however the shorter jobs will be favoured.

Consider two jobs A and B with run time  $s$  of 10 and 50 minutes respectively. After each has waited 5 minutes, their respective priorities are:

A:  $P = \frac{(5 + 10)}{10} = 1.5$

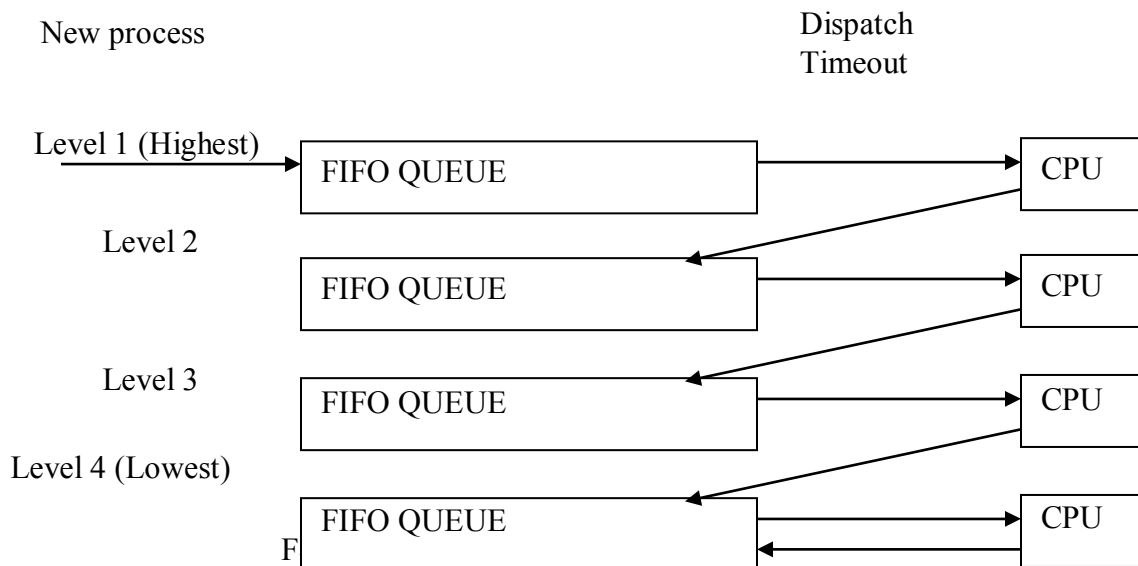
B:  $P = \frac{(5 + 50)}{50} = 1.1$

A has the highest priority.

On this basis, Job A will be selected. Note however, that if A has just started (waiting time = 0), B would be chosen in preference to A. This technique guarantees that a job cannot be started, since ultimately the effect of the wait time in the numerator of the priority expression will predominate over shorter jobs with a smaller wait time.

**Multilevel Feedback Queue (MFQ)**

Ideally, a scheduling scheme should give priority to short jobs and favour I/O bound jobs while otherwise being fair to all processes. The MFQ scheme is (see Figure 2) an attempt to provide a more adaptable policy which will treat processes on the basis of their past behaviour e.g.



Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

MFQ consists of a number of separate queues of entries which represent active processes. Each queue represents a different priority with the top queue being highest priority and lower queues successively have lower priorities. Within each queue, the queued processes are treated in a FIFO version, with a time quantum being applied to limit the amount of processor time given to each process. Processes in a lower level queue are not allocated the processor unless all the queues above that queue are empty. A new process enters the system at the end of the top queue and will eventually work its way to the front and be dispatched. If it uses up its time quantum it moves to the end of the queue in the next lower level, with the exception of the lowest queue where a RR scheme applies.

**Example**

Assume we have five processes with the following arrival time and service time, determine the finish time, waiting time, average waiting time for FCFS, RR (q=1) and SPN. What is the least average waiting time that a process will spend before execution ?

Note that the time is in seconds.

Process	A	B	C	D	E
Arrival Time	0	2	4	6	8
Service Time	3	6	4	5	2

**Solution**

	Process	A	B	C	D	E	
	Arrival Time	0	2	4	6	8	Average waiting time
	Service Time	3	6	4	5	2	
<hr/>							
FCFS	Finish time	3	9	13	18	20	
	Waiting time	3	7	9	12	12	8.60
<hr/>							
RR, q=1	Finish time	4	19	17	20	15	
	Waiting time	4	17	13	14	7	11.00
<hr/>							
SPN	Finish time	3	9	15	20	11	
	Waiting time	3	7	11	14	3	<b>7.60</b>

Of all the low level scheduling policies, SPN gives the least average waiting time that a process will spend before execution; this shows that it is the best scheduling policy among the rest.

**Activity A**

- a) What is the main objective of short-term scheduling?
- b) Distinguish between preemptive and non preemptive scheduling policies. Give one example in each case.
- c) For the processes listed in Table 1.0, draw a chart and table illustrating their execution using:
  - i) First-Come First-Served

ii) Round Robin (quantum = 2)

In the chart and table, show the finish time, waiting time and average waiting time for each technique.

Table 2.2: Process Scheduling Data

Process	Arrival Time	Service Time
A	0	3
B	1	6
C	4	4
D	6	2

#### 4.0 Conclusion

There are four types of scheduling policies: Long-term scheduling: Medium-term scheduling, Short-term scheduling and I/O scheduling. The Short-term scheduling policies have several techniques for processor scheduling. They either classified as Preemptive or Non-preemptive policies.

#### 5.0 Summary

In this unit, we have learnt that:

- (d) the aim of processor scheduling is to assign processes to be executed by the processor or processes over time in a way that meets system objectives.
- (e) several algorithms for short-term Scheduling / Dispatcher exist such as FCFS, SJF, RR, SRT, HRRN and MFQ.
- (f) each of the low-level scheduling policies can be categorized as either preemptive or non preemptive

The simplest allocation algorithm is first come-first serve (FCFS). Under this policy, a free processor is allocated to the process at the head of the queue. Once it is allocated a processor, it is removed from the queue and is allowed to execute for as long as it wishes.

When the process stops execution, either because it is destroyed or blocks itself for want of acceptance of some request, the process relinquishes the processor which is then allocated to the process currently at the head of the queue. A newly created process is placed at the end of the queue. A process is also placed at the end of the queue if it is unblocked. The queue is never reordered. The major deficiency of this algorithm is that it does not allow pre-emption which requires that the system be able to forcibly interrupt the execution of the process and take the processor away from it.

The simplest algorithm that does it, is called **Round-Robin**. The RR algorithm is the same as **fifo** with the exception that when the processor is allocated to process, the process is allowed to execute only up to a maximum length of time called the **quantum**. Thus when a processor is allocated to a process which is the head of the queue, execution continues until the process stops itself or until the quantum of time has elapsed whichever occurs first. If the process executes for the full quantum without stopping, preemption occurs.

This is achieved if setting the value of the timer equal to the length of the quantum. The preempted process is placed at the end to the queue and the processor is allocated to the process at the head of the queue.

## 6.0 Tutor Marked Assignment

- (a) Assume we have five processes with the following arrival time and service time, determine the finish time, waiting time, average waiting time for FCFS, RR ( $q=1$ ) and SPN. What is the least average waiting time that a process will spend before execution? Note that the time is in seconds.

Process	A	B	C	D	E
Arrival Time	0	2	4	6	8
Service Time	3	6	4	5	2

- (b) i. Distinguish between *long-term scheduler* and *short-term scheduler*.
- ii. Describe four short-term scheduling techniques. Indicate which one is preemptive and non-preemptive policy

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Conway R, Macwell W. and Miller L. *Theory of Scheduling*. Reading, MA: Addison-Wesley, 1967. Reprinted by Dover Publications, 2003.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Goodheart B. and Cox J. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

## MODULE 2: SCHEDULING, THREADS AND SYNCHRONIZATION

### UNIT 2: THREADS

	Page
1.0	Introduction
2.0	Objectives
3.0	Difference between process and thread
3.1	Characteristics of threads
3.2	Benefits of threads
3.3	Multithreading
3.4	Race condition
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### 1.0 Introduction

In many operating systems, the traditional concept of process has been split into two parts: one dealing with resource ownership (process) and one dealing with the stream of instruction execution (thread). A single process may contain multiple threads. A multithreaded organization has advantage both in the structure of applications and in performance. The unit examines the features and benefits of threads. It also discusses multithreading and race condition.

#### 2.0 Objectives

At the end of this unit, the reader should be able to:

- (a) Distinguish between Process & Thread
- (b) Know the characteristics and benefits of threads
- (c) Distinguish between Thread Control Block (TCB) and Process Control Block(PCB)
- (d) Know multithreading and race condition

#### 3.0 Difference between Process & Thread

**Process:** This is a general term for a program which is being executed. All work done by the CPU contributes to the execution of processes. Each process has a descriptive information structure associated with it (normally held by the kernel) called a process control block which keeps track of how far the execution has progressed and what resources the process holds.

**Thread:** (sometimes called a lightweight process) is different from process or task in that a thread is not enough to get a whole program executed. A thread is a kind of stripped down process, it is just one 'active hand' in a program, something which the CPU is doing on behalf of a program, but not enough to be called a complete process.

A thread is the basic unit of CPU utilization, which means that it is not a process that is doing the running - it's threads. Therefore, every process has at least one thread (the main thread).

This means that now we have to divide the metadata between a process and a thread:  
 A Process has its address space, global variables (that is, non-thread specific), handles (like open files), child processes (when applicable), signals and signal handlers and some accounting information).

A Thread, on the other hand, is an executing unit, and as such has its stack, registers, variables and current execution state. This is a part of the Thread Control Block.  
 Some benefits of threads over processes include: more lightweight (takes less time to create and destroy them), context switching is faster, they share memory and files without calling the kernel's system calls.

The Thread Model

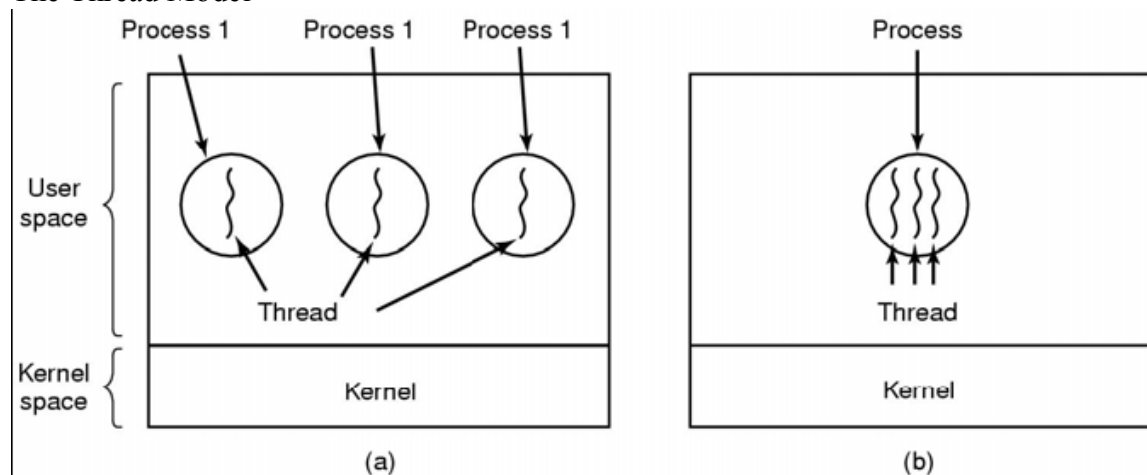


Figure 1: (a) Three processes each with one thread (b) One process with three threads  
 Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Threads remember what they have done separately, but they share the information about what resources a program is using, and what state the program is in. A thread is only a CPU assignment. Several threads can contribute to a single task. When this happens, the information about one process or task is used by many threads. Each task must have at least one thread in order to do any work.

### 3.1 Characteristics of Threads:

The TCB (thread control block) consist of

- program counter
- register set
- stack space



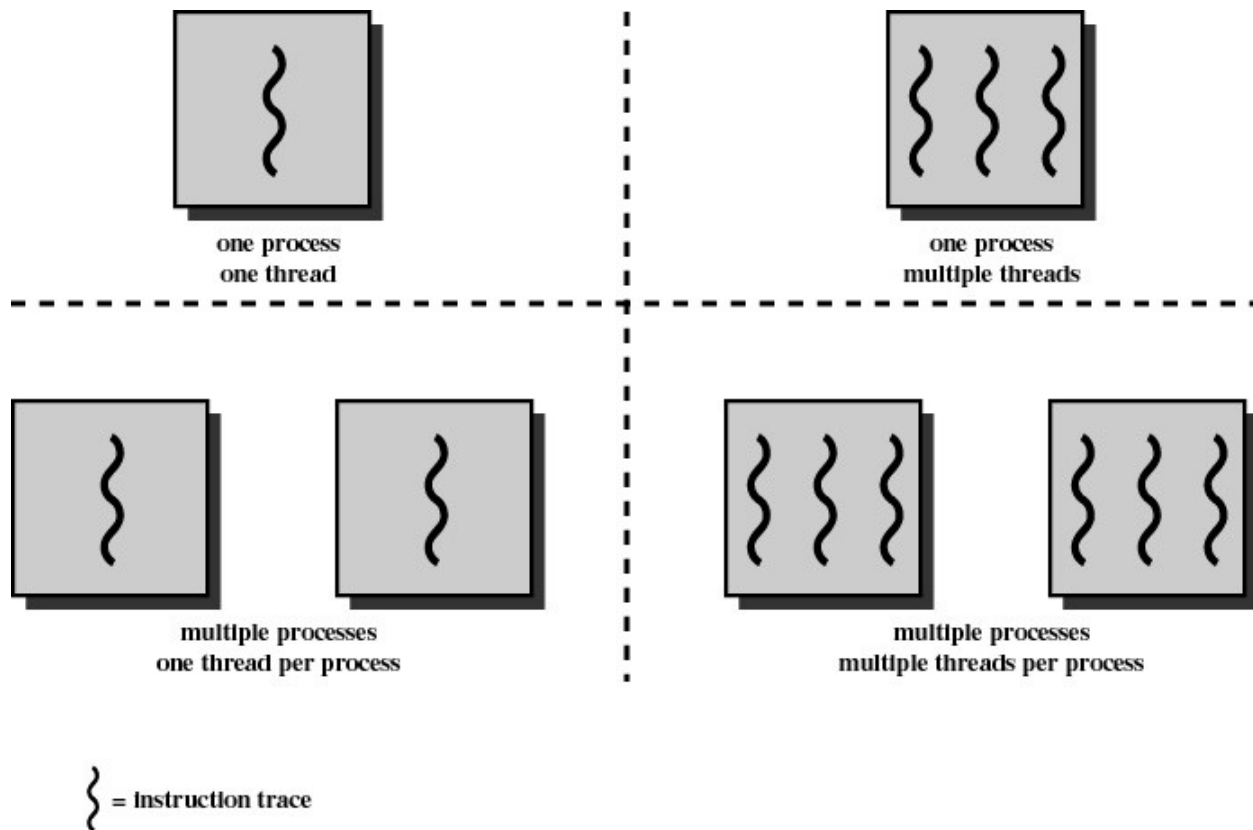


Figure 2: Threads and Processes

Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Thus the TCB is a reduced PCB

- A traditional process is equal to a task with one thread
- All threads in a process share the state of that process
- They reside in the exact same memory space (user memory), see the same code and data
- When one thread alters a process variable (say, the working directory), all the others will see the change when they next access it
- If one thread opens a file to read it, all the other threads can also read from it.
- Because no system calls are involved, threads are fast
- There are no kernel structures affected by the existence of threads in a program, so no kernel resources are consumed -- threads are cheap
- The kernel does not even know that threads exist

### 3.2 Benefits of Threads

- It takes less time to create a new thread than a process
- It takes less time to terminate a thread than a process
- It takes less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel.

- Suspending a process involves suspending all threads of the process since all threads share the same address space.
- Termination of a process, terminates all threads within the process

### **3.3 Multithreading**

- Operating system supports multiple threads of execution within a single process
- MS-DOS supports a single thread
- UNIX supports multiple user processes but only supports one thread per process
- Windows 2000, Solaris, Linux, Mach, and OS/2 support multiple threads

### **3.4 Race Condition**

We have a race condition if two processes or threads want to access the same item in shared memory at the same time. It is where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last. To prevent race conditions, concurrent processes must coordinate or be synchronized.

#### **Activity A**

- a) What is the advantage of using threads compared to processes?
- b) Distinguish between a process and a thread.

### **4.0 Conclusion**

A thread is a kind of stripped down process - it is just one 'active hand' in a program - something which the CPU is doing on behalf of a program, but not enough to be called a complete process. The TCB (thread control block) consist of: program counter, register set and stack space. Threads take less time to be created than a process. Multithreading in Operating system supports multiple threads of execution within a single process. We have a race condition if two processes or threads want to access the same item in shared memory at the same time.

### **5.0 Summary**

In this unit, we have learnt that:

- (a) A Process is different from a Thread
- (b) TCB is a reduced PCB
- (c) Windows 2000, Solaris, Linux, Mach, and OS/2 support multiple threads

### **6.0 Tutor Marked Assignment**

- (a) Briefly explain the term race condition.
- (b) Enumerate the characteristics of multithreading

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Chapin S. and Maccabe A., eds. “*Multiprocessor Operating Systems: Harnessing the Power.*” Special issue of IEEE Concurrency, April-June 1997.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Goodheart B. and Cox J. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

## MODULE 2: SCHEDULING, THREADS AND SYNCHRONIZATION

### UNIT 3: SYNCHRONIZATION

	Page
1.0	Introduction
2.0	Objectives
3.0	Semaphores
3.1	Mutual exclusion
3.2	Producer/Consumer (P/C) problem
3.3	Monitors
3.3.1	What are monitors?
3.3.2	General structure of a monitor
3.3.3	Comparison with Semaphores
3.3.4	Limitations of monitors
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### 1.0 Introduction

The two central themes of modern operating systems are multiprogramming and distributed processing. Fundamental to both themes, and fundamental to the technology of operating systems design, is concurrency. This unit looks at two aspects of concurrency control: mutual exclusion and synchronization. Mutual exclusion refers to the ability of multiple processes ( or threads) to share code, resources, or data in such a way that only one process has access to the shared object at a time. Related to mutual exclusion is synchronization: the ability of multiple processes to coordinate their activities by the exchange of information.

#### 2.0 Objectives

At the end of this unit, the reader should be able to:

- (a) Understand the mechanisms to support concurrency: semaphores, and monitors.
- (b) Apply mutual exclusion to solve real life semaphores problem.
- (c) Know the concepts and general structure of monitor.
- (d) Know the limitation of monitors.

#### 3.0 Semaphores

The term Synchronization means using atomic operations to ensure correct cooperation between processes.

The most important single contribution towards inter-process communication was the introduction of the concept of semaphores and the primitive operations wait and signal which act on them (Dijkstra, 1965).

A semaphore is a single integer variable which can take non negative value and upon which two operations, called wait and signal, are defined. Entering to critical regions of active processes is

controlled by the wait operations and exit from a critical region is signaled by the signaled operations. The definition of this operation using a semaphore S are given below.

Wait(s): if  $S > 0$  then set S to  $S - 1$

```
    Else block the calling process
        block
    endif
```

signal(s): If any processes are waiting on S start one of these processes

```
    else
        set S to  $S + 1$ 
    endif
```

Essentially a semaphore, such as S indicates the availability of some resource, if it has a none zero positive value it is available, while if it is zero it is unavailable. The signal operation signals the fact that some process has released a resource which cannot be used by a process waiting for it.

The effect of the wait and signal operations may be summarized as:

Wait(s) : when  $S > 0$  do decrement S

Signal(s): increment S

Where S is a semaphore

It can be seen from these definitions that every signal operation on a semaphore increases its value by 1 and every successful (that is, completed) wait operation decreases its value by 1.

**Example:**

Suppose that an operating system contains two processes A and B which respectively add items to and remove items from a queue. In order that the queue pointers do not become confused, it may be necessary to restrict access to the queue to only one process at a time. Thus the addition and removal of items would be coded as critical sections as shown below:

Program for process A

```
--
--
Wait (mutex);
  Add item to queue
Signal(mutex);
```

Program for process B

```
--
--
wait(mutex);
  Remove item from queue
signal(mutex);
```

Thus if mutex is 1, mutual exclusion is indeed assured.

### 3.1 Mutual Exclusion

Non-shareable resources, whether peripherals, files, or data in memory, can be protected from simultaneous access by several processes by preventing the processes from concurrently executing the pieces of program through which access is made. These pieces of program are called critical sections, and mutual exclusion in the use of resources can be regarded as mutual exclusion in the execution of critical sections. Thus each critical section is programmed as:

```
Wait(mutex);  
    critical section  
Signal(mutex);
```

where mutex is the name of a semaphore. It is clear that if the initial value is 1 then mutual exclusion is indeed assured.

#### Mutual Exclusions with Semaphores

A semaphore is a non-negative integer value. Two atomic operations are supported on a semaphore:

- down = decrement the value, since the value cannot be negative, the process blocks if the value is zero.
- up = increment the value, if there are any processes waiting to perform a down, then they are unblocked.
  - ▶ Initializing a semaphore to 1 creates a mutual exclusion (mutex) semaphore.
  - ▶ Initializing a semaphore to 0 creates a signaling semaphore.
  - ▶ Initializing a semaphore to other values can be used for resource allocation and synchronization.

**Other Examples of semaphore are presented below:**

- ▶ Example 1: a print program produces characters that are consumed by a printer.
- ▶ Example 2: an assembler produces object modules that are consumed by a loader.

### 3.2 Producer/Consumer (P/C) Problem

#### Paradigm for cooperating processes

Producer process produces information that is consumed by a Consumer process.

#### Producer/Consumer (P/C) Operation

The operations of the producer/consumer are described below:

- ▶ A producer process produces information that is consumed by a consumer process.
- ▶ At any time, a producer activity may create some data.
- ▶ At any time, a consumer activity may want to accept some data.

- ▶ The data should be saved in a buffer until they are needed.
- ▶ If the buffer is finite, we want a producer to block if its new data would overflow the buffer.

We also want a consumer to block if there are no data available when it wants them. We need a buffer to hold items that are produced and later consumed (see Figure 1):

- ▶ unbounded-buffer places no practical limit on the size of the buffer.
- ▶ bounded-buffer assumes that there is a fixed buffer size.

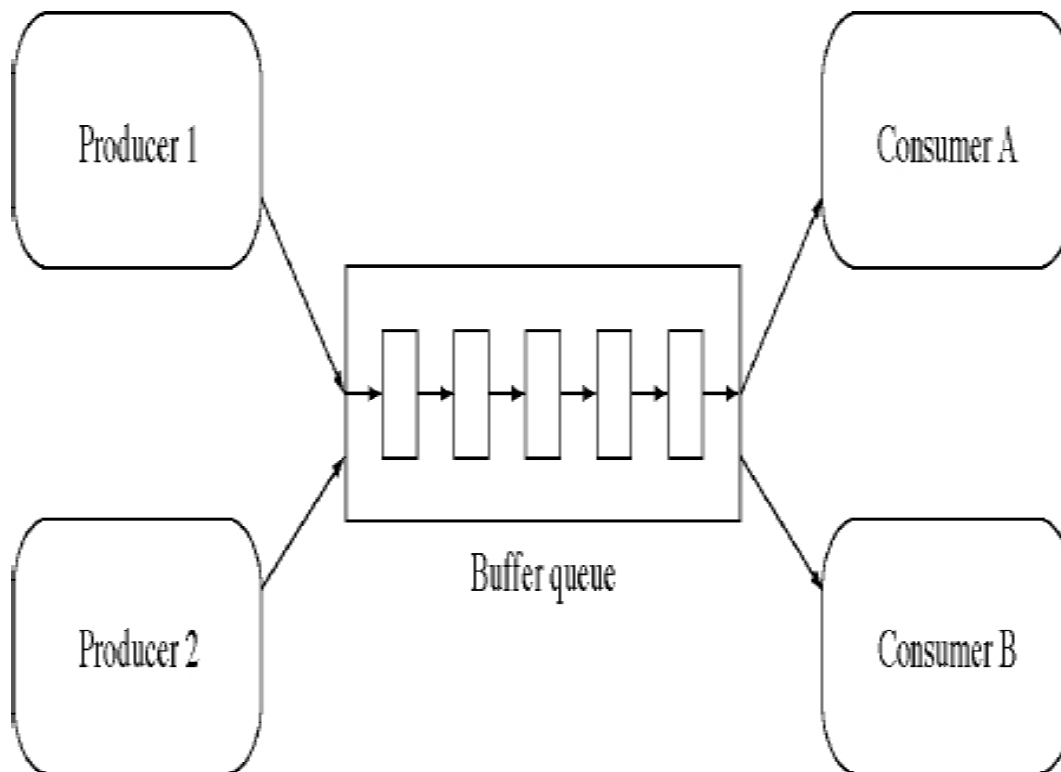


Figure 3.1: Multiple Producers and Consumer Problems  
 Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Figure 3.1 shows a multiple producers and consumers scenario whereby a producer process produces information in the buffer queue that is consumed by a consumer process.

## Solution to the Producer/Consumer Problem

### Introduction

This algorithm uses semaphores to solve the producer/consumer (or bounded buffer) problem. Note that *Up* is the same as signal, and *Down* is the same as wait

### Algorithm

```
1 var   buffer: array [0..n-1] of item;
2       full, empty, mutex: semaphore;
3       nextp, nextc: item;
4 begin
5     full := 0;
6     empty := n;
7     mutex := 1;
8     parbegin
9         repeat                                (* producer process *)
10            (* produce an item in nextp *)
11            down(empty);
12            down(mutex);
13            (* deposit nextp in buffer *)
14            up(mutex);
15            up(full);
16        until false;
17        repeat                                (* consumer process *)
18            down(full);
19            down(mutex);
20            (* extract an item in nextc *)
21            up(mutex);
22            up(empty);
23            (* consume the item in nextc *)
24        until false;
25    parend;
26 end.
```

### Comments

lines 1-3: Here, buffer is the shared buffer, and contains n spaces; full is a semaphore the value of which is the number of filled slots in the buffer, empty is a semaphore the value of which is the number of empty slots in the buffer, and mutex is a semaphore used to enforce mutual exclusion (so only one process can access the buffer at a time), nextp and nextc are the items produced by the producer and consumed by the consumer respectively.

line 5-7: This just initializes all the semaphores. It is the only time anything other than a down or an up operation may be done to them.

line 10: Since the buffer is not accessed while the item is produced, we don't need to put semaphores around this part.

lines 11-13: Depositing an item into the buffer, however, does require that the producer process obtains exclusive access to the buffer. First, the producer checks that there is an empty slot in the buffer for the new item and, if not, waits until there is (down(empty)). When there is, it waits until it can obtain exclusive access to the buffer (down(mutex)). Once both these conditions are met, it can safely deposit the item.



lines 14-15: As the producer is done with the buffer, it signals that any other process needing to access the buffer may do so (`up(mutex)`). It then indicates it has put another item into the buffer (`up(full)`).

lines 18-20: Extracting an item from the buffer, however, does require that the consumer process obtains exclusive access to the buffer. First, the consumer checks that there is a slot in the buffer with an item deposited and, if not, waits until there is (`down(full)`). When there is, it waits until it can obtain exclusive access to the buffer (`down(mutex)`). Once both these conditions are met, it can safely extract the item.

lines 21-22: As the consumer is done with the buffer, it signals that any other process needing to access the buffer may do so (`up(mutex)`). It then indicates it has extracted another item into the buffer (`up(empty)`).

line 23: Since the buffer is not accessed while the item is consumed, we do not need to put semaphores around this part.

Note that `up` is the same as `signal`, and `Down` is the same as `wait`

### 3.3 Monitors

#### 3.3.1 What are Monitors?

Monitors provide a structured concurrent programming primitive, which is used by processes to ensure exclusive access to resources, and for synchronizing and communicating among users. A monitor module encapsulates both a resource definition and operations/procedures that exclusively manipulate it. Those procedures are the gateway to the shared resource and called by the processes to access the resource. Only one call to a monitor procedure can be active at a time and this protects data inside the monitor from simultaneous access by multiple users. Thus mutual exclusion is enforced among tasks using a monitor. Processes that attempt monitor entry while the monitor is occupied are blocked on a monitor entry queue.

To synchronize tasks within the monitor, a condition variable is used to delay processes executing in a monitor. It may be declared only within a monitor and has no numeric value like semaphores do. Two operations, **wait** and **signal**, are defined on condition variables. The **wait** operation suspends/blocks execution of the calling process if a certain condition is true, then the monitor is unlocked and allows another task to use the monitor. When the same condition becomes false, then the **signal** operation resumes execution of some processes suspended after a **wait** on that condition, by placing it in the processor ready queue. If there are several such processes, choose one of them; if there are no waiting processes, the **signal** operator is ignored. Therefore, the introduction of condition variables allows more than one process to be in the same monitor at the same time, although only one of them will be actually active within that monitor.

A condition variable is associated with a queue of the processes that are currently waiting on that condition. First-in-first-out (FIFO) discipline is generally used with queues, but priority queues can also be implemented by specifying the priority of the process to be delayed as a parameter in the **wait** operation. (Condition variables are assumed to be fair in the sense that a process will not remain suspended forever on a condition variable that is signaled infinitely often.) Therefore, monitors allow flexibility in scheduling of the processes waiting in queues.

### 3.3.2 General Structure of a Monitor

```
< Monitor-Name > : monitor
begin
    Declaration of data local to the monitor.
    .
    .
    procedure < Name > ( < formal parameters > );
        begin
            procedure body
        end;

    Declaration of other procedures
    .
    .
    begin
        Initialization of local data of the monitor
    end;
end.
```

Note that a monitor is not a process, but a static module of data and procedure declarations. The actual processes which use the monitor need to be programmed separately.

### 3.3.3 Comparison with Semaphores

The **wait** and **signal** operations on condition variables in a monitor are similar to **P** and **V** operations on counting semaphores. A **wait** statement can block a process's execution, while a **signal** statement can cause another process to be unblocked. However, there are some differences between them. When a process executes a **P** operation, it does not necessarily block that process because the counting semaphore may be greater than zero. In contrast, when a **wait** statement is executed, it always blocks the process. When a task executes a **V** operation on a semaphore, it either unblocks a task waiting on that semaphore or increments the semaphore counter if there is no task to unlock. On the other hand, if a process executes a **signal** statement when there is no other process to unblock, there is no effect on the condition variable.

Another difference between semaphores and monitors is that users awakened by a **V** operation can resume execution without delay. Contrarily, users awakened by a **signal** operation are restarted only when the monitor is unlocked.

In addition, a monitor solution is more structured than the one with semaphores because the data and procedures are encapsulated in a single module and that the mutual exclusion is provided automatically by the implementation.

### 3.3.4 Limitations of Monitors

Since only one process can be active within a monitor at a time, the absence of concurrency is the major weakness in monitors and this leads to weakening of encapsulation. For example, in the Readers-Writers problem, the protected resource (a file, database, etc.) is separated from the monitor and so, there is a possibility that some malicious Reader or Writer could simply access the database directly without getting a permission from the monitor to do so.

Another problem is the possibility of deadlock in the case of nested monitor calls. For example, consider a process calling a monitor that in turn calls another lower-level monitor procedure. If a **wait** is executed in the last monitor called, the mutual exclusion will be relinquished by the process. However, mutual exclusion will not be relinquished by processes in monitors from which nested calls have been made. Processes that attempt to invoke procedures in these monitors will become blocked. If those blocked processes are the only ones which can cause signaling to occur in the lower level monitor, then deadlock occurs.

### Activity A

- a. What is a semaphore?
- b. What is the critical-section problem?
- c. State with explanation, each requirement that a solution to the critical-section problem must satisfy.

### 4.0 Conclusion

Mutual exclusion is a condition in which there is a set of concurrent processes, only one of which is able to access a given resources or perform a given function at any time. Mutual exclusion techniques can be used to resolve conflicts, such as competition for resources, and to synchronize processes so that they can cooperate. An example of the latter is the producer/consumer problem, in which one process is putting data into a buffer and one or more processes are extracting data from that buffer. Monitors provide a structured concurrent programming primitive, which is used by processes to ensure exclusive access to resources, and for synchronizing and communicating among users.

### 5.0 Summary

In this unit, we have learnt that:

- (a) Semaphores are used for signaling among processes and can be readily used to enforce a mutual-exclusion discipline.
- (b) Monitors can be engaged to realize concurrency.
- (c) Limitations exist for using monitors such as possibility of deadlock in terms of nested monitor calls.

### 6.0 Tutor marked Assignment

- (a) State the Mutual Exclusion Principle.
- (b) Briefly recall the Producer and Consumer Problem:  
*Producer process produces information that is consumed by a consumer process. We need a buffer to hold items that are produced and later consumed. At any time, a producer activity may create some data. At any time, a consumer activity may want to accept some data. The data should be saved in a buffer until they are needed. If the buffer is full, we want a producer to block to avoid new data overflowing the buffer. We also want a consumer to block if there are no data available when it wants them.*
- (c) Write an algorithm using a semaphore to solve the producer/consumer problem

## 7.0 Further Reading and Other Resources

Axford T. *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*. New York: Wiley, 1998.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

Tamaki Kurusu (1999), “*MONITORS in Concurrent Programming*”, (excerpted from <http://ei.cs.vt.edu/~cs5204/sp99/monitor.html>)

# **CIT752: OPERATING SYSTEM CONCEPTS**

## **MODULE 3: COMMUNICATION, DEADLOCKS AND INTERRUPTS**

### **UNIT 1: MESSAGES**

	<b>Page</b>
1.0	Introduction
2.0	Objectives
3.0	Inter-Process Communication (IPC)
3.1	Direct communication
3.2	Indirect communication
3.3	Synchronization
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources.

#### **1.0 Introduction**

When process interacts with one another, two fundamental requirements must be satisfied: synchronization and communication. Processes need to be synchronized to enforce mutual exclusion; and cooperating processes may need to exchange information. One approach to providing both of these functions is message passing. This unit examines inter process communication and communication model.

#### **2.0 Objectives**

At the end of this unit, the reader should be able to:

- (a) Understand message passing.
- (b) Know how more than one process communicate with each other.
- (c) Know the direct and indirect means of communication between processes

#### **3.0 Inter-Process Communication (IPC)**

IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. IPC is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected with a network. An example is a chat program used on the World Wide Web. IPC is best provided by a message-passing system. Message systems can be defined in many different ways.

#### **Message-Passing System**

The function of a message system is to allow processes to communicate with one another without the need to resort to shared data. An IPC facility provides at least the two operations send(message) and receive(message). Messages sent by a process can be of either fixed or

variable size. If only fixed sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen through operating system design. If processes  $P$  and  $Q$  want to communicate, they must send messages to and receive messages from each other; and a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network), but rather with its logical implementation. Here are several methods for logically implementing a link and the `send()/receive()` operations:

- Direct or indirect communication.
- Synchronous or asynchronous communication.
- Automatic or explicit buffering.

### **Naming**

Naming is another issue in effective communication. Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

#### **3.1 Direct Communication**

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the `send()` and `receive()` primitives are defined as follows:

- `send(P, message)` — Send a message to process  $P$ .
- `receive(Q, message)` — Receive a message from process  $Q$ .

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that wants to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits *symmetry* in addressing; that is, both the sender and the receiver processes have to be named in order to communicate. A variant of this scheme employs *asymmetry* in addressing. Only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the `send()` and `receive()` primitives are defined as follows:

- `send(P, message)` - Send a message to process  $P$ .
- `receive(id, message)` - Receive a message from any process; the variable *id* is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such hard-coding techniques where identifiers must be explicitly stated are less desirable than those involving a level of indirection, as described next.

### 3.2 Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other processes via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox, however. The `send()` and `receive()` primitives are defined as follows:

- `send(A, message)` — Send a message to mailbox A.
- `receive(A, message)` — Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now, suppose that processes *P1*, *P2*, and *P3* all share mailbox A. Process *P1* sends a message to A, while *P2* and *P3* each execute a `receive()` from A. Which process will receive the message sent by *P1*? The answer depends on the scheme that we choose:

- Allow a link to be associated with at most two processes.
- Allow at most one process at a time to execute a `receive()` operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either *P2* or *P3*, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, round robin). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user of the mailbox (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox
- Send and receive messages through the mailbox
- Delete a mailbox

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

### 3.3 Synchronization

Communication between processes takes place by calls to `send()` and `receive()` primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking—also known as synchronous and asynchronous.

- **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send:** The sending process sends the message and resumes operation.
- **Blocking receive:** The receiver blocks until a message is available.
- **Nonblocking receive:** The receiver retrieves either a valid message or a null.

Different combinations of `send()` and `receive()` are possible. When both the `send()` and `receive()` are blocking, we have a **rendezvous** between the sender and the receiver.

Note that the concepts of synchronous and asynchronous occur frequently in operating-system I/O algorithms, as will be seen throughout this module.

Primitives are defined as: `send(A, message)` to send a message to mailbox A. `Receive(A, message)` to receive a message from mailbox A. Mailbox sharing involves P1, P2, and P3 share of mailbox A. P1 will send and P2 and P3 will receive. The question is therefore who gets the message. The communication model is presented in Figure 1.



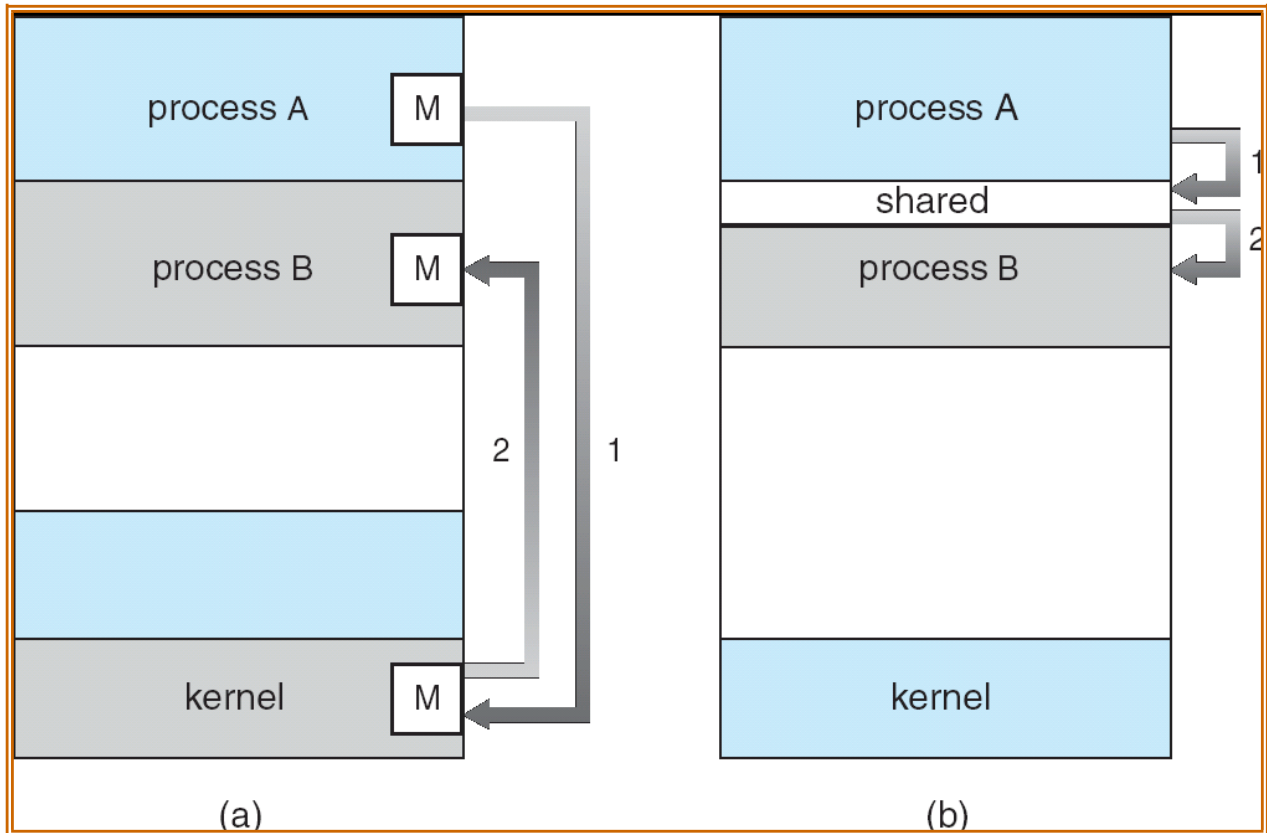


Figure 1.1: Communication models  
 Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

The communication of more than one process for data exchange with the kernel is shown in Figure 1a. First, the operating system kernel sends a message to process A requesting for process status. Second, the kernel receives the request and update process B with the process information. The shared segment of memory is presented in Figure 1b. Two processes A and B share the information stored in the shared segment.

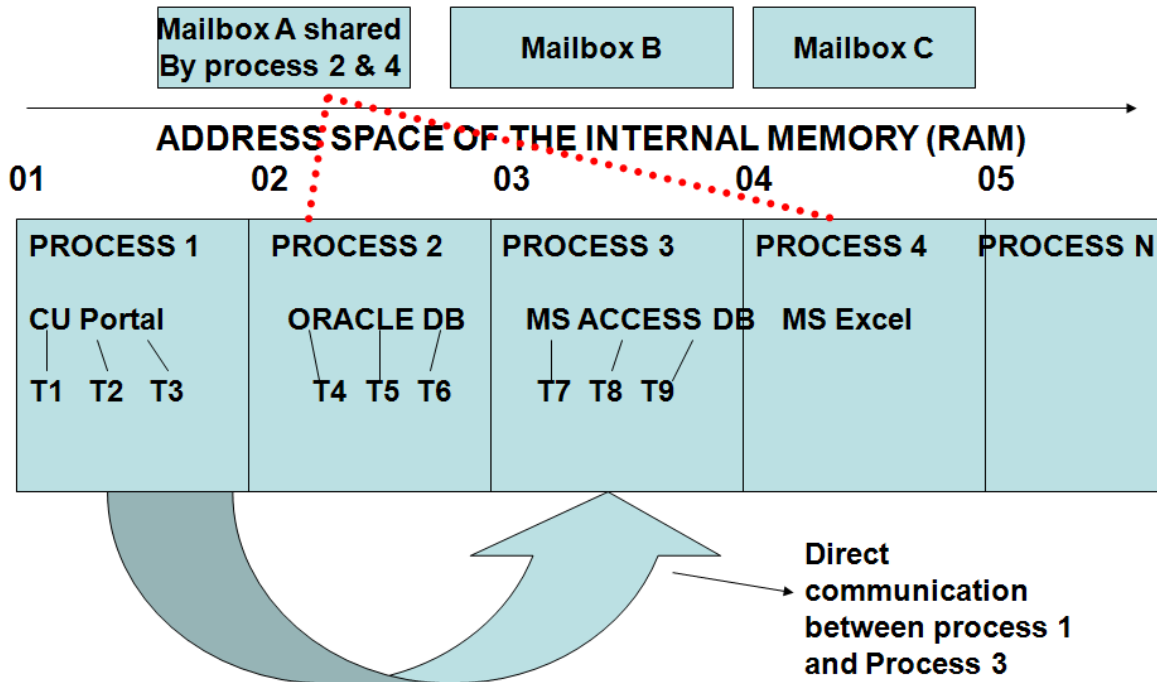


Figure 1.2. Address Space of the internal memory.

The address space of the internal memory is described in Figure 2. It contains Addresses (01,02..05), Processes(1,2,..N), and Thread T1, T2 and T3 belonging to process 1. Process 1 is CU portal, Process 2 is oracle DB, process 3 is MS Access DB and process 4 is MS Excel. A process has at least one thread. The arrow from process 1 to process 3 shows a direct communication between the two processes. It is also possible for two processes to share the same mail box. For instance, mailbox A is shared by process 2 and process 4.

The following recommendation and implementation questions are necessary for effective communication:

### Recommendations

- ▶ Allow a link to be associated with at most two processes.
- ▶ Allow only one process at a time to execute a received operation.
- ▶ Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

### Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bidirectional?

## Activity A

What is message passing?

## 4.0 Conclusion

Messaging passing system allow processes to communicate with one another without the need to resort to shared data. An IPC facility provides at least the two operations send (message) and receive(message). Communication between processes takes place by calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking—also known as synchronous and asynchronous. Processes communicate with each other directly or indirectly without resorting to shared variables. Processes must name each other explicitly.

## 5.0 Summary

In this unit, we have learnt that:

- (d) Processes can communicate with each other.
- (e) Direct communication uses send and receive primitive.
- (f) Indirect communication uses mailbox.

## 6.0 Tutor Marked Assignment

- (a) State the differences between direct and indirect communication.
- (b) Highlight the implementation issues affecting Synchronization and Communication.

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Bacon J. and Harris T. *Operating Systems: Concurrent and Distributed Software Design*. Reading, MA: Addison-Wesley, 1998.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

## MODULE 3: COMMUNICATION, DEADLOCKS AND INTERRUPTS

### UNIT 2: DEADLOCKS

	Page
1.0	Introduction
2.0	Objectives
3.0	Conditions for Deadlock
3.1	Deadlock Prevention
3.2	Deadlock Avoidance
3.3	Deadlock Detection
3.4	Classical example for deadlock
4.0	Conclusion
5.0	Summary
6.0	Tutor marked Assignment
7.0	Further Reading and Other Resources

#### 1.0 Introduction

Deadlock refers to a situation in which a set of two or more processes are waiting for other members of the set to complete an operation in order to proceed, but none of the members is able to proceed. Deadlock is a difficult phenomenon to anticipate, and there are no easy general solutions to this problem. This unit looks at three major approaches to dealing with deadlock: prevention, avoidance and detection.

#### 2.0 Objectives

At the end of this unit, the reader should be able to:

- (a) Understand the condition of deadlock.
- (b) Know the approaches to dealing with deadlock
- (c) Know a Classical example for deadlock

#### 3.0 Conditions for Deadlock

Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlock when each process in the set is blocked awaiting an event (typically the freeing up of some requested resources) that can only be triggered by another blocked process in the set. Deadlock is permanent when none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case. A common example is the traffic deadlock.

#### Three conditions for Deadlock

Three conditions of policy must be present for a deadlock to be possible:

1. Mutual exclusion. Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.

2. Hold and Wait. A process may hold allocated resources while awaiting assignment of other resources.
3. No preemption. No resource can be forcibly removed from a process holding it.

In many ways these conditions are quite desirable. For example, mutual exclusion is needed to ensure consistency of results and the integrity of a database. Similarly, preemption should not be done arbitrarily. For example, when data resources are involved, preemption must be supported by a rollback recovery mechanism, which restores a process and its resources to a suitable previous state from which the process can eventually repeat its actions.

Deadlock can occur with these three conditions but might not exist with just these three conditions. For deadlock to actually take place, a fourth condition is required.

4. Circular wait. A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

The first three conditions are necessary but not sufficient for a deadlock to exist. The fourth condition is, actually, a potential consequence of the first three. That is, given that the first three conditions exist, a sequence of events may occur that lead to an unresolved circular wait. The unresolved circular wait is in fact the definition of deadlock. The circular wait listed as condition 4 is unresolvable because the first three conditions hold. Thus, the four conditions, taken together, constitute necessary and sufficient conditions for deadlock.

### **3.1 Deadlock Prevention**

The strategy of deadlock prevention is, simply to design a system in such a way that the possibility of deadlock is excluded. We can view deadlock prevention methods as falling into two classes. An indirect method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously (items 1 through 3). A direct method of deadlock prevention is to prevent the occurrence of a circular wait (item 4). We now examine techniques related to each of the four conditions.

#### **Mutual Exclusion**

In general, the first of the four listed conditions cannot be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the operating system. Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes. Even in this case, deadlock can occur if more than one process requires write permission.

#### **Hold and Wait**

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously. This approach is inefficient in two ways. First, a process may be held up for a long time waiting for all of its resource requests to be fulfilled, when in fact it could have proceeded with only some of the resources. Second, resources allocated to a process may remain

unused for a considerable period during which time they are denied to other processes. Another problem is that a process may not know in advance all of the resources that it will require.

There is also the practical problem created by the use of modular programming or a multithreaded structure for an application. An application would need to be aware of all resources that will be required at all levels or in all modules to make the simultaneous request.

### **No preemption**

This condition can be prevented in several ways. First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource. Alternatively, if a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources. This latter scheme would prevent deadlock only if no two processes possessed the same priority.

This approach is practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor.

### **Circular wait**

The circular wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only resources of types following R in the ordering.

## **3.2 Deadlock Avoidance**

An approach to solving the deadlock problem that differs subtly from deadlock prevention is deadlock avoidance. In deadlock prevention, we constrain resource requests to prevent at least one of the four conditions of deadlock. This is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing circular wait. This leads to inefficient use of resources and inefficient execution of processes. Deadlock avoidance, on the other hand, allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows a more concurrency than prevention. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Deadlock avoidance thus requires knowledge of future process resource requests.

## **3.3 Deadlock Detection**

Deadlock prevention strategies are very conservative. They solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes. At the opposite extreme, deadlock detection strategies do not limit resource access or restrict process actions. With deadlock detection, requested resources are granted to processes whenever possible. Periodically, the operating system performs an algorithm that allows it to detect the circular wait condition described earlier in condition (4).

### 3.4 Classical example for deadlock

**The Dining Philosophers Problem** is stated as follows:

*“Five philosophers (the actual number is unimportant) sit around a table. In the middle of the table is a large bowl of spaghetti. Between each set of philosophers is a fork. That is, for five philosophers, there are five plates and five forks. Each philosopher sits around and thinks for a while and then talks for a while and then eats for a while. Since there are only five forks, each philosopher must reach for first fork and then the other. At any given moment, only one philosopher can hold a given fork, and a philosopher cannot pick up two forks simultaneously. In addition, once a philosopher has a fork, s/he holds onto it (after all, s/he is hungry) until s/he can get the other fork (s/he needs both forks to eat). Once a philosopher starts eating, the forks are not relinquished until the eating phase is over. When the eating phase concludes, which last for a finite time, both forks are put back in their original position and the philosopher re-enters the thinking phase. Note that no two neighbouring philosophers can eat simultaneously. The problem occurs when all the philosophers grab a right or left fork at once. Each then has one fork and waits forever to obtain the other”.*

#### **How the requirements for deadlock are associated with the Dining Philosophers Problem**

The requirements for deadlock can be summed up by the following four conditions:

1. **Mutual exclusion:** Only a single process can hold a resource or modify shared information at one time. In our case study, a fork can only be held by a single philosopher at a time.
2. **Circular waiting:** A thread or process A can wait for a process B, which in turn waits on C, which in turn waits on A; so the chain from A to C back to A forms a loop.
3. **Piecemeal allocation:** Resources can be allocated piecemeal or one at a time. For our case, each philosopher can grab a fork one at a time.
4. **Lack of preemption:** Once a resource has been granted to a process, it cannot be taken away. In the Dining Philosophers problem, once a fork is obtained it cannot be taken away.

#### **The Mutual Exclusion Principle and how it can resolve the classical Dining Philosophers Problem.**

In any solution to the above Dining Philosophers problem, the act of picking up a fork by a philosopher must be a critical section. The classical mutual exclusion problem is depicted by the fact that the execution of a concurrent program on a multiprocessor system may require the processes to access shared data structures. This may cause the processors to concurrently access a location in the shared memory. Clearly, a mechanism is needed to serialize this access to shared data structures to guarantee its correctness. The mechanism should make accesses to a shared data structure appear atomic with respect to each other. A solution to this problem requires that processes be synchronized such that only one process can access the variable at any one time. This is why the problem is widely referred to as the problem of mutual exclusion.

#### **A practical application in a fully distributed system**

The Mutual Exclusion Principle application to resolve the Dining Philosophers Problem employs the fully distributed algorithm characterized by: all nodes have equal amount of information, on the average; each node has a partial picture of the total system and must make decisions based on this information; all nodes bear equal responsibility for the final decision; all nodes expend equal

effort, on the average, in effecting a final decision; failure of a node, in general does not result in a total system collapse; and there exists no system wide common clock with which to regulate the timing of events. The above solution ensures that the successful use of concurrency among processes requires the ability to define critical sections and enforce mutual exclusion. This is fundamental for any concurrent processing scheme. When concurrent processes interact through a shared variable, the integrity of the variable may be violated if access to the variable is not coordinated.

### Activity A

- a. Distinguish between:
  - (i) Deadlock and Starvation
  - (ii) Deadlock Avoidance and Deadlock Prevention
- b. What strategies do deadlock avoidance and deadlock prevention algorithms deploy?
- c. How can the hold-and-wait condition be prevented?

### 4.0 Conclusion

Deadlock is the blocking of a set of processes that either compete for system resources or communicate with each other. There are three general approaches to dealing with deadlock: prevention, detection and avoidance. Deadlock prevention guarantees that deadlock will not occur, by assuring that one of the necessary conditions for deadlock is not met. Deadlock detection is needed if the operating system is always willing to grant resource requests; periodically, the operating system must check for deadlock and take action to break the deadlock. Deadlock avoidance involves analyzing each new resource to determine if it could lead to deadlock, and granting it only if deadlock is not possible.

### 5.0 Summary

In this unit, we have learnt that:

- a) Three conditions of policy must be present for a deadlock to be possible.
- b) For deadlock to actually take place, a fourth condition is required: ie the Circular wait.

### 6.0 Tutor marked Assignment

- a. What are the requirements for deadlock?
- b. What strategies do deadlock avoidance and deadlock prevention algorithms deploy?
- c. Recall a version of **The Dining Philosophers Problem** is stated as follows:

*“Five philosophers (the actual number is unimportant) sit around a table. In the middle of the table is a large bowl of spaghetti. Between each set of philosophers is a fork. That is, for five philosophers, there are five plates and five forks. Each philosopher sits around and thinks for a while and then talks for a while and then eats for a while. Since there are only five forks, each philosopher must reach for first fork and then the other. At any given moment, only one philosopher can hold a given fork, and a philosopher cannot pick up two forks simultaneously. In addition, once a philosopher has a fork, s/he holds onto it (after all, s/he is hungry) until s/he can get the other fork (s/he needs both forks to eat). Once a philosopher starts eating, the forks are not relinquished until the eating phase is over. When the eating phase concludes, which last for a finite time, both forks are put back in their*



*original position and the philosopher re-enters the thinking phase. Note that no two neighbouring philosophers can eat simultaneously. The problem occurs when all the philosophers grab a right or left fork at once. Each then has one fork and waits forever to obtain the other”.*

- i. How are the requirements for deadlock associated with the Dining Philosophers Problem?
- ii. State the Mutual Exclusion Principle and narrate how it can resolve the classical Dining Philosophers Problem.
- iii. Illustrate your narration with a practical application in a fully distributed system.

## **7.0 Further Reading and Other Resources**

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Coff E. and Elphick M. and Shoshani A. “*System Deadlocks.*” *Computing Survey*, June 1971.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

## **MODULE 3: COMMUNICATION, DEADLOCKS AND INTERRUPTS**

### **UNIT 3: INTERRUPTS**

**Page**

1.0	Introduction
2.0	Objectives
3.0	Instruction Cycle with Interrupts
3.1	What is an Interrupt Handler?
3.2	Interrupt processing
3.3	Classes of Interrupts
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### **1.0 Introduction**

Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor. Interrupts are provided as a way to improve processor utilization. This unit examines interrupt and the instruction cycle. It also discusses the classes of interrupt.

#### **2.0 Objectives**

At the end of this unit, the reader should be able to:

- (a) Understand the Instruction Cycle with Interrupts
- (b) How Interrupts improve CPU usage
- (c) Classes of Interrupts

#### **3.0 Instruction Cycle with Interrupts**

Interrupts are provided primarily as a way to improve processor utilization. For example, most I/O devices are much slower than the processor. With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress(see Figure 1). The CPU goes through the following steps in handling an interrupt:

1. Computers permit I/O modules to INTERRUPT the CPU.
2. For this the I/O module just assert an interrupt request line on the control bus
3. Then CPU transfer control to an Interrupt Handler Routine (normally part of the OS).
4. CPU checks for interrupts after each instruction.
5. If no interrupts, then fetch the next instruction for the current program.
6. If an interrupt is pending, then suspend execution of the current program, and execute the interrupt handler.

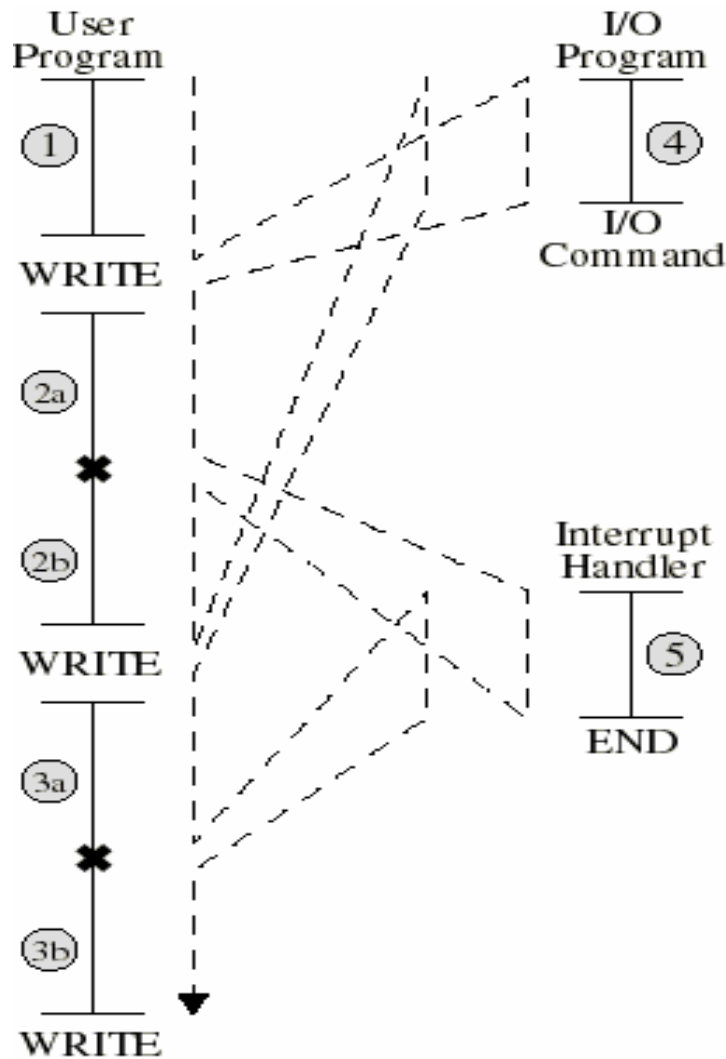


Figure 2.1: Interrupts; short I/O wait

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Consider the flow of control in Figure 1, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced, that is, when it is ready to accept more data from the processor, the I/O module for that external device sends an interrupt request signal to the processor. The processor responds by suspending operation of the current program; branching off to a routine to service that particular I/O device, known as an interrupt handler;

and resuming the original execution after the device is serviced. The point at which such interrupts occur are indicated by a X in Figure 1. Note that an interrupt can occur at any point in the main program, not just at one specific instruction.

### 3.1 What is an Interrupt Handler?

The interrupt handler routine is generally part of the operating system. Typically, this routine determines the nature of the interrupt and performs whatever actions are needed. The handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt-handler routine is completed, the processor can resume execution of the user program at the point of interruption.

### 3.2 Interrupt Processing

The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software.

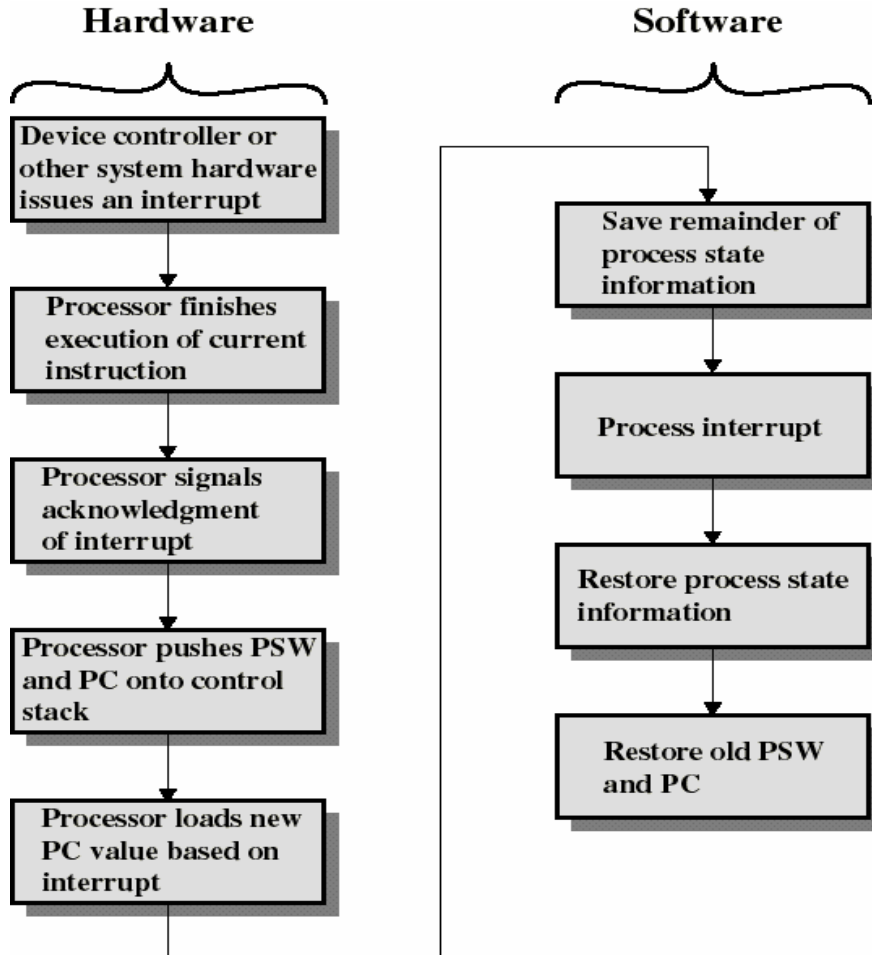


Figure 2.2: Simple Interrupt processing

Source: William Stallings (2005), Operating Systems, Internals & Design principles, fifth edition, Pearson Hall.

Figure 2 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgement signal to the device that issues the interrupt. The acknowledgement allows the device to remove its interrupt signal.
4. The processor now needs to transfer control to the interrupt routine. To begin, it needs to save information needed to resume the current program at the point of interrupt. The minimum information required is the program status word (PSW) and the location of the next instruction to be executed, which is contained in the program counter. These can be pushed onto the system control stack.
5. The processor now loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt. Depending on the computer architecture and operating system design, there may be a single program, one for each type of interrupt or one for each device and each type of interrupt.

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, the result is that control is transferred to the interrupt handler program. The execution of this program results in the following operations.

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the system stack. However, there is other information that is considered part of the state of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler.
7. The interrupt may now proceed to process the interrupt. This will include an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers.
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

### **3.3 Classes of Interrupts**

Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor. The most common classes of interrupts are listed as follows:

1. I/O  
Signals normal completion of operation or error
2. Program Exception

- overflows
  - try to execute illegal instruction
  - reference outside user's memory space
3. Timer  
Preempts a program to perform another task
  4. Hardware failure (e.g. memory parity error)

### Activity A

What effect does an interrupt have when it is sent to the CPU?

### 4.0 Conclusion

The interrupt handler routine is generally part of the operating system. Interrupts improve CPU usage. There are four classes of Interrupts: I/O, Program, Timer and hardware failure.

### 5.0 Summary

In this unit, we have learnt that:

- a) Interrupts are provided primarily to improve processor utilization.
- b) With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.
- c) The most common classes of interrupt are I/O, program execution, timer and hardware failure.

### 6.0 Tutor Marked Assignment

- (a) What is an Interrupt?
- (b) Explain four classes of interrupt.
- (c) With the aid of a diagram describe a simple interrupt processing.

### 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Coff E. and Elphick M. and Shoshani A. "System Deadlocks." *Computing Survey*, June 1971.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Hennessy J and Patterson D. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2002.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

# **CIT752: OPERATING SYSTEM CONCEPTS**

## **MODULE 4: DEUGGING AND MEMORY MANAGEMENT**

### **UNIT 1: OS DEBUGGING STRATEGIES**

	<b>Page</b>
1.0	Introduction
2.0	Objectives
3.0	Debugging strategies
3.1	Debugging techniques
3.1.1	Code and data breakpoints
3.1.2	Live, post mortem and remote debugging
4.0	Conclusion
5.0	Summary
6.0	Tutor marked Assignment
7.0	Further Reading and Other Resources

#### **1.0 Introduction**

Bugs refer to errors in a program, and the process of removing them (error correction) is called debugging. Debuggers allow us to step through the codes in a program, stepping over or into functions and methods with a view to correcting the errors (bugs) in the program. When we debug in a top-down fashion we initially step over bodies of code we consider irrelevant, narrowing down our search as we come nearer the problem's manifestation. This strategy requires patience and persistence. Often we step-over a crucial function and find ourselves having to repeat the search aiming to step-into the function the next time round

#### **2.0 Objectives**

At the end of this unit, the reader should be able to:

- (a) know the meaning of debugging.
- (b) know the various techniques used for debugging.

#### **3.0 Debugging strategies**

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

The testing, diagnostic, and repair equipment of many professions is horrendously expensive. For us, the cost of debuggers and logging frameworks is minimal; some of them are even free. All we need to become productive, is to invest some time and effort to learn how to use these tools in the most efficient and effective way.

With the existing debugging tools, we can often get a starting point for locating a bug, and then also verify our hypotheses on what is going wrong. As one would expect, adopting an appropriate strategy and mastering the corresponding techniques are the important factors for making the best out of these tools.

The most efficient debugging strategy is a bottom-up one: we start from the symptom and look for the cause. The symptom can be a memory access violation (for example the dereferencing of a NULL pointer), an endless loop, or an uncaught exception. A debugger will typically allow us to get a snapshot of the program at the point where the symptom occurred. From that snapshot we can examine the program's stack frame: the sequence of function or method invocations that led to the execution of the problematic code. At the very least we thus obtain an accurate picture of our program's runtime behavior. Even better, we can also examine the values of variables at each level of the stack frame to really understand what brought our program belly-up.

Unfortunately, there are times when we can not adopt a bottom-up strategy. This situation crops up when the bug's symptom can not be precisely tied to a debugger event. Our program may cause a problem in another application, or the contents of a variable may be wrong for reasons we can not explain. In such cases top-down is the name of the game. This process can be tiring, but sooner or later will produce results.

There are also cases where we may have to debug a program at the level of assembly code: either because we do not trust the compiler, or because we do not have access to the program's source code. What we have found over the years is that assembly code is a lot less intimidating than it appears. Even if we do not know the processor's architecture, a few educated guesses and a bit of luck often allow us to decipher the instructions needed to pinpoint the problem.

### **3.1 Debugging techniques**

#### **3.1.1 Code and data breakpoints**

Stack frame printouts and stepping commands are the basic and indispensable debugging tools, but there are more powerful commands that can often help us locate a tricky problem. A *code breakpoint* allows us to stop the program's execution at a specific line. We often use those to expedite a top down bug search, by placing a breakpoint before the point where we think the problem lies. In such cases we use the breakpoint as a bookmark for the location where we want to look at the program's operation in more detail.

Less known, but no less valuable, are *data breakpoints*—also known as watchpoints. Many modern processors provide hardware support that will interrupt a program's execution when the code accesses the contents of some specified memory locations. Data breakpoints leverage this support allowing programmers to specify that the program's execution will stop when its code reads or writes a variable, an array or an object. Note that debuggers that implement such commands without hardware support slow down the program's execution to a crawl rendering this command almost useless (Java tool builders should take note).



### 3.1.2 Live, post mortem and remote debugging

**Print (live) debugging** is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process.

Often the first step in debugging is to attempt reproduce the problem. This can be a non-trivial task, for example in case of parallel processes or some unusual software bugs. Also specific user environment and usage history can make it difficult to reproduce the problem.

After the bug is reproduced, the input of the program needs to be simplified to make it easier to debug. For example, a bug in a compiler can make it crash when parsing some large source file. However, after simplification of the test case, only few lines from the original source file can be sufficient to reproduce the same crash. Such simplification can be made manually, using a divide-and-conquer approach. The programmer will try to remove some parts of original test case and check if the problem still exists. When debugging the problem in GUI, the programmer will try to skip some user interaction from the original problem description and check if remaining actions are sufficient for bug to appear. To automate test case simplification, delta debugging methods can be used.

After the test case is sufficiently simplified, a programmer can use a debugger to examine program states (values of variables, the call stack) and track down the origin of the problem. Alternatively tracing can be used. In simple case, tracing is just a few print statements, which output the values of variables in certain points of program execution.

**Post-mortem debugging** is the act of debugging the core dump of process. The dump of the process space may be obtained automatically by the system, or manually by the interactive user. Crash dumps (core dumps) are often generated after a process has terminated due to an unhandled exception.

**Remote debugging** is the process of debugging a program running on a system different than the debugger. To start remote debugging, debugger connects to a remote system over a network. Once connected, debugger can control the execution of the program on the remote system and retrieve information about its state.

Although the typical set-up involves starting the misbehaving program under a debugger, there are also other debugging options that can often help to escape a tight corner.

Consider non-reproducible bugs, also known as Heisenbugs, because they make programs appear as if they are operating under the spell of Heisenberg's uncertainty principle. We can often pinpoint these by debugging a program after it has crashed. Typical Unix systems crashed programs will leave behind them an image of their memory, the *core dump*. By running a debugger on this core dump we get a snapshot of the program's state at the point of the crash. Windows, on the other hand, offers us the possibility to launch a debugger immediately after a program has crashed. In both situations we can then look at the location of the crash, and examine the values the variables had at the time. If the program has not crashed but is acting

weirdly, we can attach a debugger to that running process, and examine its operation from that point onward using the debugger's commands.

Another class of applications that is difficult to debug is that with an interface that is incompatible with the debugger's. Embedded systems, operating system kernels, games, and applications with a cranky GUI fall in this category. Here the solution is *remote debugging*. We run the process under a debugger, but interact with the debugger's interface on another system, connected through the network or a serial interface. This leaves the target system almost undisturbed, but still allows us to issue debugging commands and view their output from our debugging console.

### **Activity A**

What are the strategies for debugging a program?

### **5.0 Conclusion**

There are available techniques for debugging a program: Code and data breakpoints; and Live, post mortem and remote debugging.

### **5.0 Summary**

In this unit, we have learnt how:

To debug an OS program using the techniques: Code and data breakpoints; and Live, post mortem and remote debugging.

### **6.0 Tutor marked Assignment**

- (a) What is an OS debugging technique?
- (b) Mention and discuss two OS debugging technique

### **7.0 Further Reading and Other Resources**

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Hennessy J and Patterson D. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2002.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

Wikipedia, *the free encyclopedia*, [http://en.wikipedia.org/wiki/Operating\\_system](http://en.wikipedia.org/wiki/Operating_system)

## **MODULE 4: DEBUGGING AND MEMORY MANAGEMENT**

### **UNIT 2: MEMORY ALLOCATION TECHNIQUES**

	<b>Page</b>
1.0	Introduction
2.0	Objectives
3.0	Memory Management Concepts
3.1	Single Contiguous Memory Allocation
3.2	Partition Allocation
3.2.1	Static Memory allocation
3.2.2	Dynamic Memory allocation
3.3	Relocation partition
3.4	Paging
3.5	Segmentation
3.6	Translation lookaside buffer (TLB)
4.0	Conclusion
5.0	Summary
6.0	Tutor marked Assignment
7.0	Further Reading and Other Resources

#### **1.0 Introduction**

One of the difficult aspects of operating system design is memory management. A central task of the operating system is to manage memory. This unit examines the fundamental mechanisms used in memory management.

#### **2.0 Objectives**

At the end of this unit, the reader should be able to:

- (a) Know memory management concepts.
- (b) Know the requirement of memory management.
- (c) Activities of memory management.
- (d) Objectives of memory management.
- (e) Understand the various mechanisms and policies associated with memory management.

#### **3.0 Memory Management Concepts**

The main purpose of a computer is to execute programs to produce result (information). These programs, together with the data they access, must be in main memory (at least partially during execution).

In a uniprogramming system, main memory is divided into two parts: one part for the operating system and another for the program currently being executed. In a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as memory management. To improve both the utilization of the CPU and the speed of its response to users, the computer must keep several processes in memory. The several memory management schemes shall be discussed in the next section.

## **Activities of memory management**

The tasks of the memory management module of the OS include:

- ▶ Keeping track of which parts of memory are currently being used and by whom.
- ▶ Deciding which processes (or parts thereof) and data to move into and out of memory.
- ▶ Allocating memory space to processes as required.
- ▶ Deallocating memory spaces from processes.

## **The Objectives of Memory Management**

The following are the requirements that memory management is intended to satisfy.

### **Relocation**

In a multiprogramming system, the available main memory is generally shared among a number of processes. Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of a program. In addition, we would like to be able to swap active processes in and out of main memory to maximize processor use by providing a large pool of ready processes. Once a program has been swapped out to disk, it would be quite limiting to declare that when it is next swapped back in it must be placed in the same main memory region as before.

### **Protection**

Each process should be protected against unwanted interference by other processes, whether accidental or intentional. Thus, programs in other processes should not be able to reference memory locations in a process, for reading or writing purposes, without permission.

### **Sharing**

Any protection mechanisms that are implemented must have the flexibility to allow several processes to access the same portion of main memory. For example, if a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program rather than have its separate copy.

### **Logical Organisation**

Almost invariably, main memory in a computer system is organized as a linear, or one-dimensional, address space that consists of a sequence of bytes or words. Secondary memory, at its physical level, is similarly organized. Although this organization closely mirrors the actual machine hardware, it does not correspond to the way in which programs are typically constructed. Most programs are organized into modules, some of which are unmodifiable (read-only, executed-only) and some of which contain data that may be modified. If the operating system and computer hardware can effectively deal with user programs and data in the form of modules of some sort, then a number of advantages can be realized such as modules can be

written and compiled independently, with all references from one module to another resolved by the system at run time.

## Physical Organization

As we are aware, computer memory is organized into at least two levels: main memory and secondary memory. Main memory provides fast access at relatively high cost. In addition, main memory is volatile; that is, it does not provide permanent storage. Secondary memory is slower and cheaper than main memory, and it is usually not volatile. Thus, secondary memory of large capacity can be provided to allow for long-term storage of programs and data, while a smaller main memory holds programs and data currently in use.

## Swapping

A process needs to be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling techniques. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that is in the queue (see Figure 1).

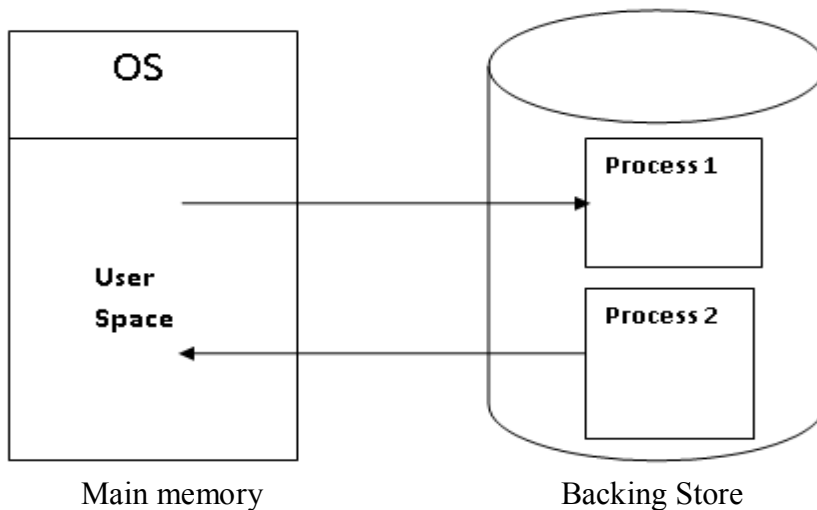


Figure 2.1: Swapping of two processes using a disk as backing store.

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

## Memory Management Schemes

The OS make use of the following memory management schemes to allocate jobs in memory.

- 1 Single Contiguous Memory Allocation
- 2 Partition Allocation
  - Static Memory allocation
  - Dynamic Memory allocation
- 3 Relocation partition
- 4 Paging
- 5 Segmentation

### 3.1 Single Contiguous Memory Allocation

Single contiguous allocation is a simple memory management scheme that requires no special hardware features. It is usually associated with small stand-alone computers (or minicomputers) with simple batch operating system, e.g. IBM OS/360 Primary Control Program, IBM 1130 Disk Monitor System, IBM 7094 Fortran Monitor System. In such a systems there is no multiprogramming, and one to one correspondence exist between a user, and a job, or a process. Thus the terms job, or process may be used interchangeably. Memory is allocated to the job, as depicted in figure 2.

The term job or process is defined as a program in execution or an instance of a program running on a computer. A user is a person that submits a job or request to the computer for execution.

Memory is conceptually divided into three contiguous regions. A portion of memory is permanently allocated to the operating system. All of the remainder of memory is available (and allocated) to the single job being processed. The job actually uses a portion of the allocated memory, leaving an allocated but unused region of memory. For example, if there are 256K bytes of memory, a simple operating system may require 32k bytes, leaving 224k bytes allocated for user jobs. If a typical job requires only 64k bytes, then 160k bytes of memory (over 50 % of the memory) unused.

#### Advantages

- Simplicity, compared to more sophisticated operating system.
- It does not require great expertise to understand or use the system.

#### Disadvantages

- Memory is not fully utilized (ie wasted).
- Processor is underutilized.

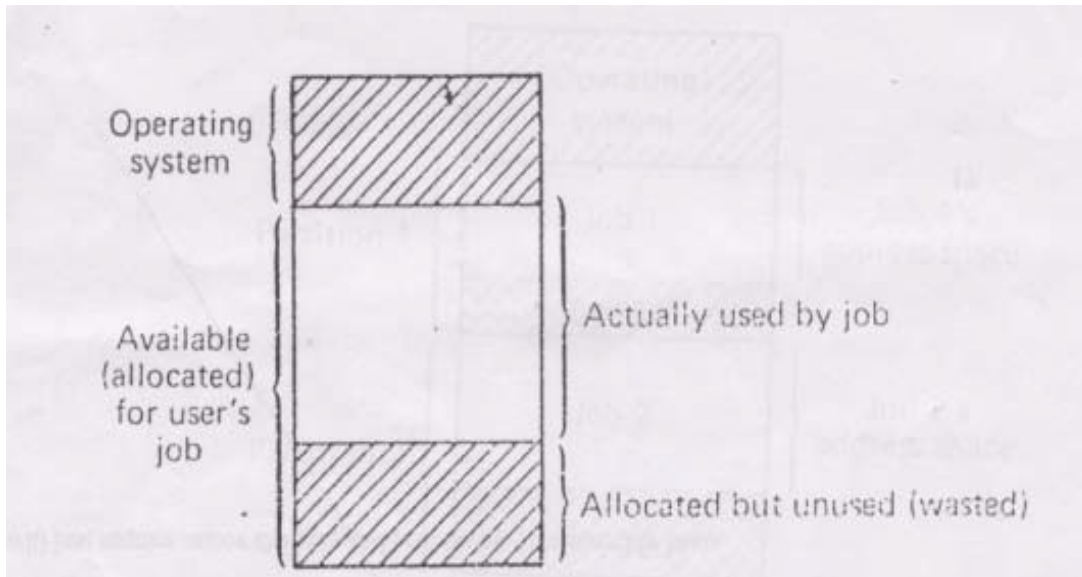


Figure 2.2. Single contiguous allocation

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

### 3.2 Partition Allocation

Partition allocation, in its various forms, is one of the simplest memory management techniques for supporting multiprogramming. Main memory is divided into separate memory regions or memory partitions. Each partition holds a separate job's address space, as illustrated in Figure 3.

The two common versions of partition technique are static partition and dynamic partition.

#### 3.2.1 Static memory allocation

In static partition, the memory is divided into partitions prior to the processing of any jobs. This is similar to the technique used in IBM's OS/360 MFT (Multiprogramming with a Fixed number of Tasks). Each job step supplied by a user must specify the maximum amount of memory needed. A partition of sufficient size is then found and assigned.

The technique of static specification is especially appropriate when the sizes and frequency of jobs are well known. In such a case, the partition sizes are chosen to correspond closely to the most common job sizes. However, there can be considerable memory wasted if the sizes and frequencies of jobs are not known, or if they are diverse. For example, if many jobs of sizes 1K, 9K, 33K, and 121K are to be run, we could assign these to partitions as follows:

Partition	Partition Size	Job size	Wasted space
1	8k	1k	7k
2	32k	9k	23k
3	32k	9k	23k
4	120k	33k	87k
5	520k	121k	399k
	-----	-----	-----
	712k	173k	539k

In this case, all the partitions are assigned in the best possible way, yet only 173k of the available 712k is actually used. Thus, over 75k percent of the available memory is wasted.

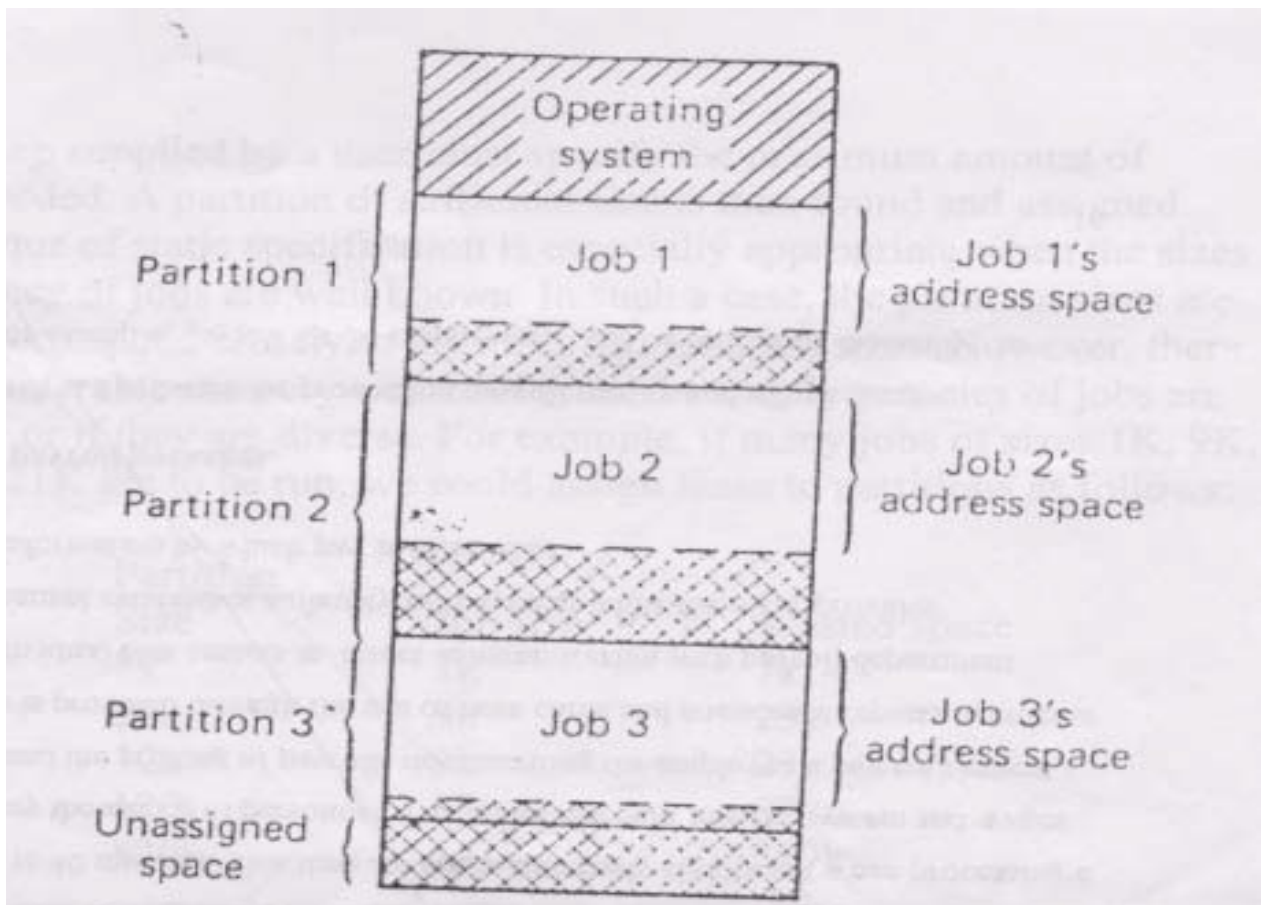


Figure 2.3: Partition allocation

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.



### 3.2.2 Dynamic memory allocation

In dynamic partitioning, the partitions are created during job processing so as to match partition sizes to job sizes.

An example of dynamic partitioning is presented in Figure 4. At some point, three partitions are allocated, each containing a job of corresponding size (Figure 4a). Three additional jobs are then selected to be multiprogrammed, and new partitions of appropriate sizes are created from the unallocated free areas (Figure 4b). Eventually, the partitions can be deallocated after the corresponding job is terminated. Figure 4c depicts the memory status after jobs 2 and 3 terminate.

Various algorithms are available to accomplish the allocation and deallocation functions. Certain obvious steps must be performed for allocation. First a free area at least as large as the partition desired must be found. Second, if the area is larger than needed, it must be split into two pieces – one becomes the allocated partition, the other becomes a smaller free area. Conversely, when a partition is deallocated, we try to merge it with any adjacent free areas so as to make one contiguous free area rather than many small pieces. (See, for example the handling of job 3's partition in Figure 4b and c).

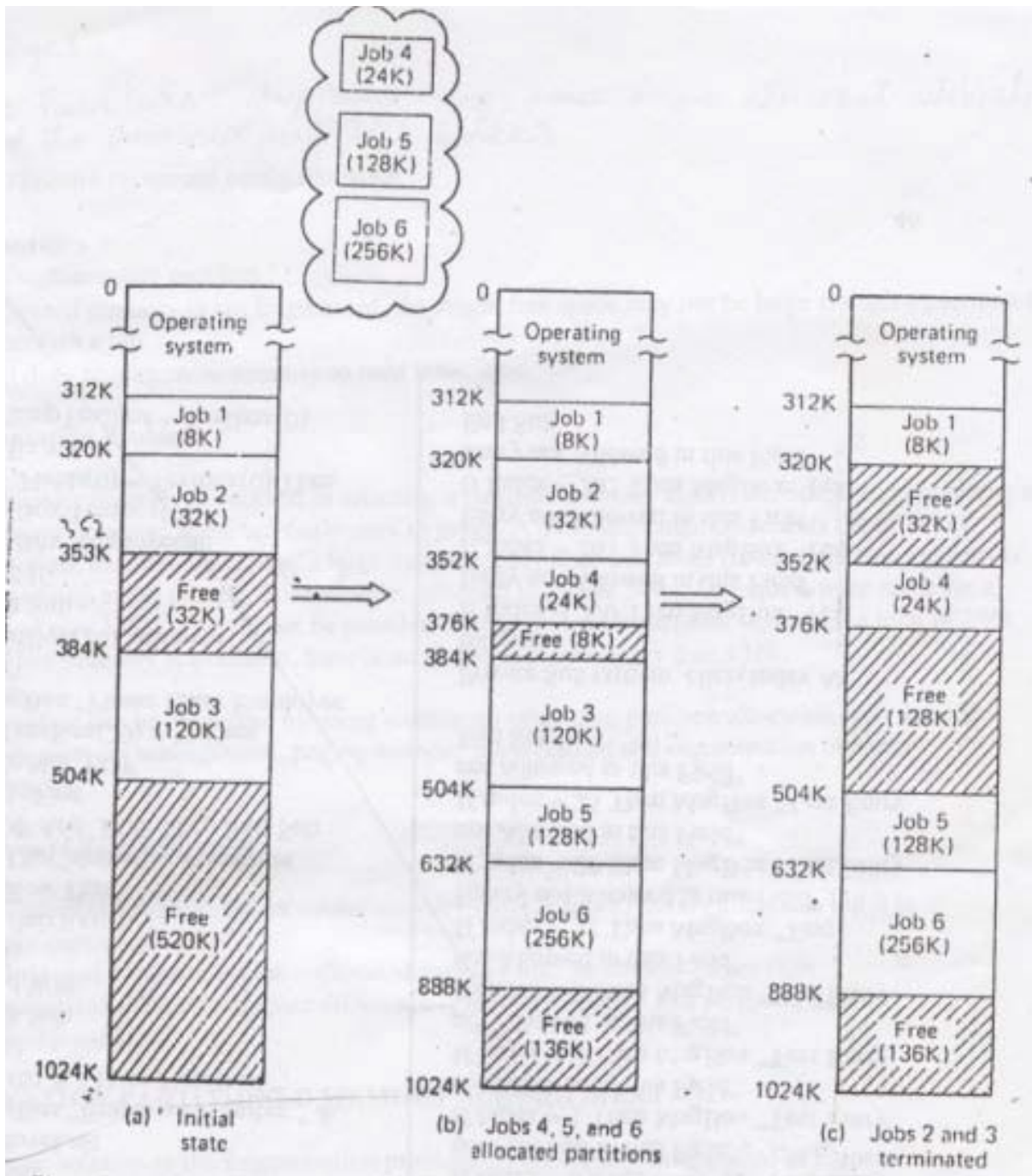


Figure 2.4: Partition allocation and deallocation

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

### **Advantages**

- It facilitates multiprogramming, hence more efficient utilization of the processor and I/O devices.
- It requires no special costly hardware.

### **Disadvantages**

- Fragmentation problem (the development of a large number of separate free areas (ie the total free memory is fragmented into small pieces).
- Even if memory is not fragmented, the single free space may not be large enough a partition to contain a job.
- It does require more memory to hold more jobs.

### **Fragmentation Problem**

Several factors must be considered in selecting a partition memory algorithm. Speed and simplicity are among them. However, these are fairly easy to judge. A more important concern is the effect of fragmentation, the development of a large number of separate free areas (ie the total free memory is fragmented into small pieces). For example, referring to Figure 4c, if a request were made for a partition of size 138k, it would not be possible to allocate such a partition. Although a total of 296k bytes of free memory is available, there is no single free area larger than 136k.

Fragmentation can be overcome by using techniques other than partition allocation, and these are relocation memory management, paging memory management and segmentation memory management.

### **Types of Fragmentation**

- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous.
- Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

### **3.3 Relocation partition**

An obvious solution to the fragmentation problem (as previously exemplified in Figure 4c) is to periodically combine all free areas into one contiguous area. This can be done by moving the contents of all allocated partitions so that they become contiguous, as illustrated in Figure 5. The process is called compaction (or recompaction, since it is done many times). The term “burping” the memory has also been used to describe this technique.

Although it is conceptually simple, moving a job’s partition does not guarantee that the job will run correctly at its new location. This is because there are many location-sensitive items, such as: (1) base register, (2) memory referencing instructions, (3) parameter lists, and (4) data structures (e.g.) lists, pointers etc) that use address pointers. To operate correctly, all location-sensitive items must be suitably modified. For example, in Figure 5, job 4 is moved from location 352k to

320k. All addresses within job 4's partition must then be decreased by 32k. This process of adjusting location-sensitive addresses is called relocation.

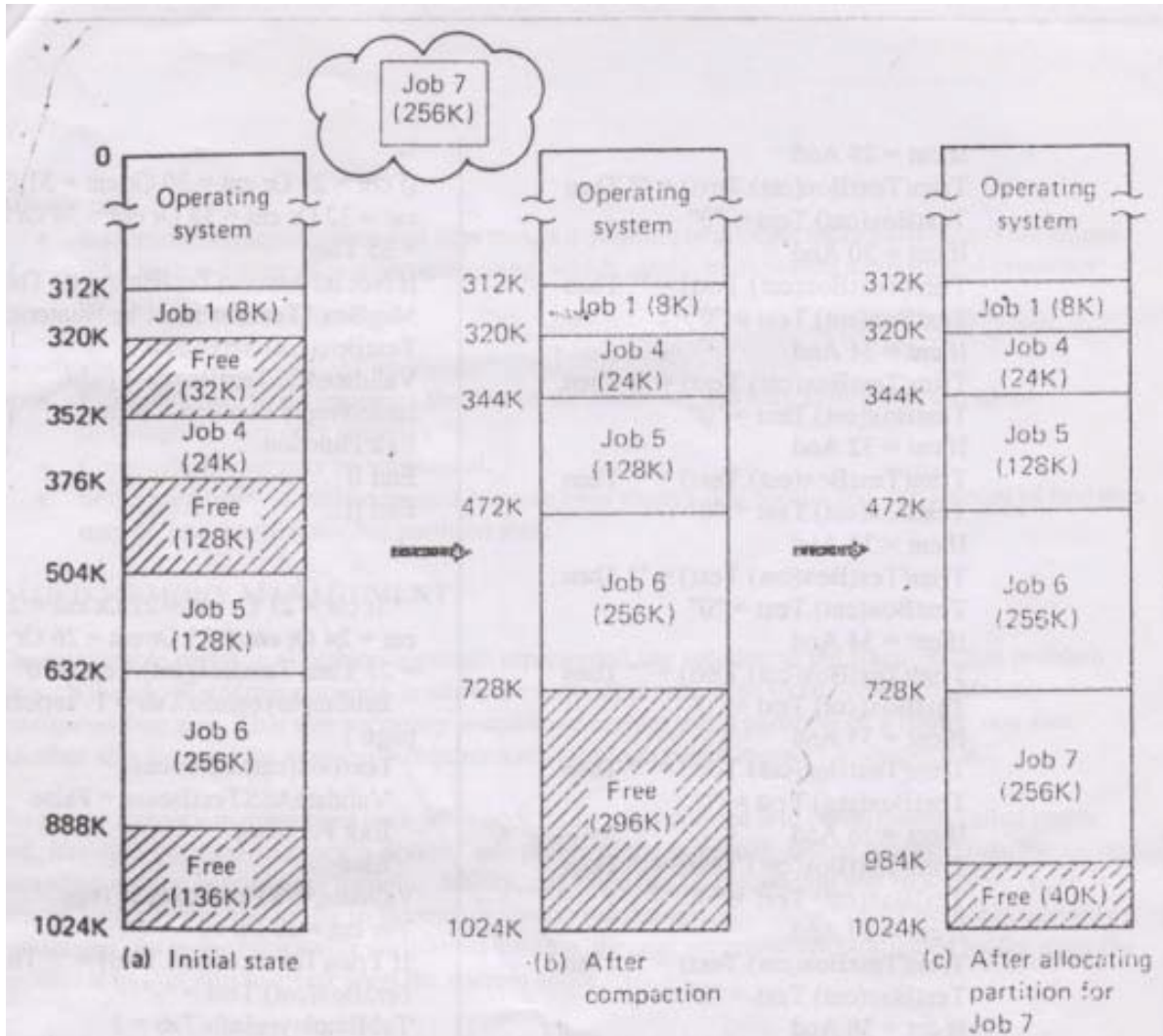


Figure 2.5: Relocatable partition compaction

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

### **Advantages**

- It eliminates fragmentation and thus makes it possible to allocate more partitions.
- It allows for a higher degree of multiprogramming, which results in increased memory and processor utilization.

### **Disadvantages**

- Relocation hardware increases the cost of the computer and may slow down the speed (although usually only slightly).
- Compaction time may be substantial.
- Some memory will still be unused because even though it is compacted, the amount of free area may be less than the needed partition size.

## **3.4 Paging**

The relocatable partition allocation approach represented one solution to the fragmentation problem through the use of address mapping to allow the individual free areas to be coalesced into one contiguous free area. This was necessary because it is required that a partition be a contiguous area. Another way to avoid the contiguity requirement is through paged memory management.

In paging memory management, each job's address space is divided into equal pieces, called pages, and, likewise, physical memory is divided into pieces of the same size, called blocks. Then, by providing a suitable hardware mapping facility, any page can be placed into any block. The pages remain logically contiguous (ie to the user program) but the corresponding blocks are not necessarily contiguous. As in the relocatable partitioned scheme, the user programs are unaffected by the mapping because it has no visible effect upon the address space.

For the hardware to perform the mapping from address space to physical memory, there must be a separate register for each page. These registers are often called page maps or Page Map Tables (PMTs). They may be either special hardware registers or a reserved section of memory. Since each page can be separately relocated, there is no need for a job's partition to be completely contiguous in memory; only locations in a single page must be contiguous.

A simple example, using a 1000-byte page size is shown in Figure 6. Job 2, which has an address space 3000 bytes, is divided into three pages. The page table associated with job 2 indicates the location of its pages. In this case, page 0 is in block 2, page 1 is in block 4, and page 2 is in block 7. In this case, page 0 is in block 2, page 1 is in block 4, and page 2 is in block 7. The LOAD 1,2108 instruction at location 0518 (block 2, byte 518). Likewise, the data 015571 logically located at 2108 is stored at physical memory location 7108.

The paged memory management approach solves the fragmentation problem without physically moving partitions. For example, in Figure 6 there are 2000 bytes of available memory, but they are not contiguous. If there were a fourth job requiring 2000 bytes, we could compact memory, as is done with relocatable partitions, to produce a single 2000-byte free area. Alternatively, we can assign job 4's two pages to the available blocks, such as page 0=block 3 and page 1 = block

9. If the PMT is set correspondingly, the address space will still appear contiguous without making it necessary to physically move any partitions.

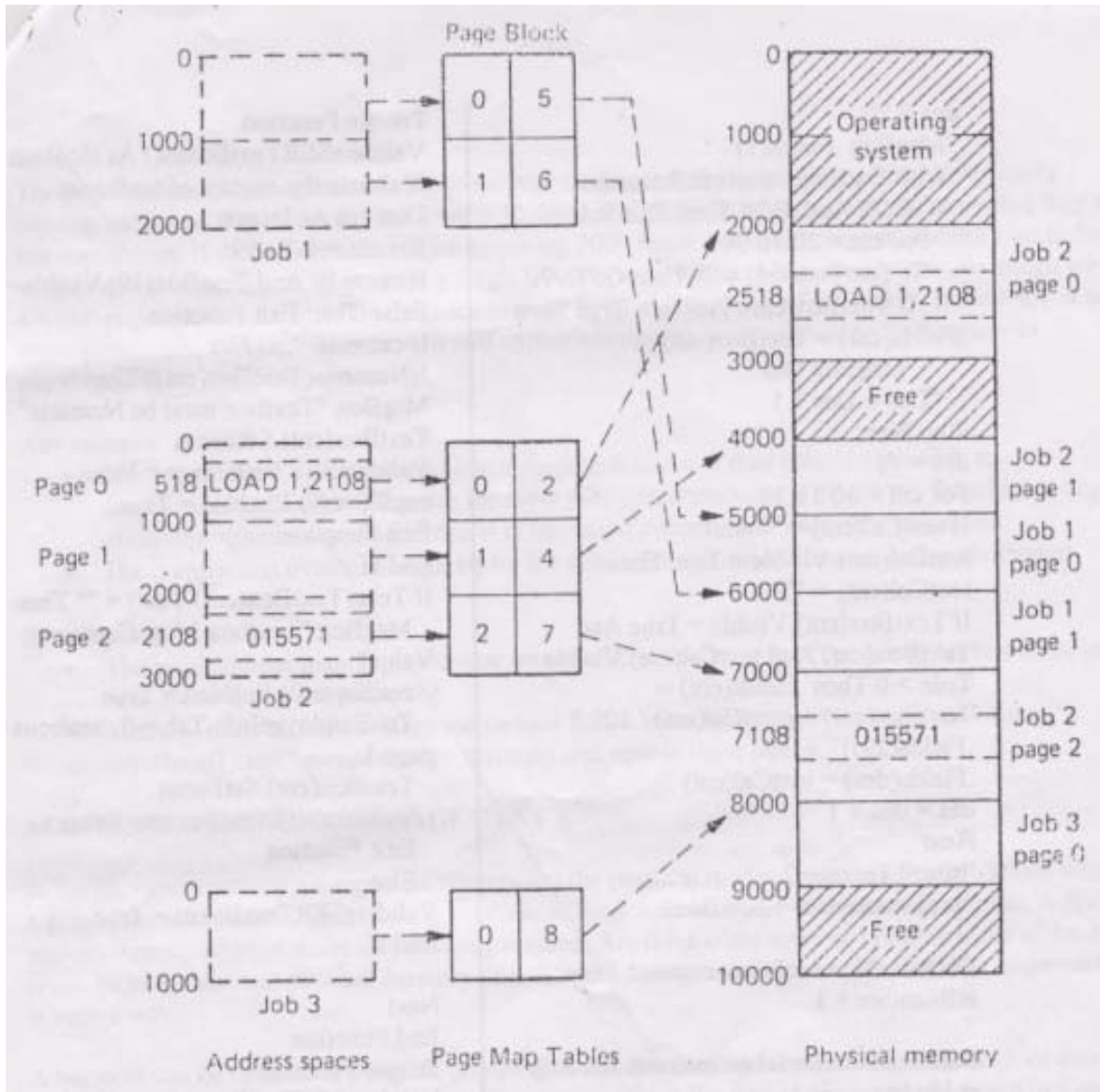


Figure 2.6: Paged mapped memory

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

### Advantages

- The paged memory scheme eliminates fragmentation and thus makes it possible to accommodate the address spaces for more jobs simultaneously. This allows for a higher degree of multiprogramming, which results in increased memory and processor utilisation.
- The compaction overhead required for the relocatable partition scheme is also eliminated.

### Disadvantages

- The page address mapping hardware usually increases the cost of the computer and at the same time slows down the processor.
- Memory must be used to store the various tables, principally the PMT, processor time (overhead) must be expended to maintain and update these tables.

## 3.5 Segmentation

In the previous memory management approaches the physical memory was handled in various ways, e.g. partitioned, relocated, pages, etc.; yet these actions were invisible to the user's program. A linear and contiguous address space was always provided. Are there other ways of presenting the address space so as to facilitate efficient memory utilization and programmer convenience ? Yes, segmentation is such a way.

A segment can be defined as a logical grouping of information, such as a subroutine, array, or data area. Thus, each job's address space actually consists of a collection of segments, as shown in Figure 8. Segmentation is the technique for managing these segments.

In a segmented environment all address space of references requires two components: (1) a segment specifier and (2) the location within the segment. For example, references may be as shown in Figure 7.

CALL [X] <Y>	TRANSFER TO ENTRY POINT Y WITHIN SUBROUTINE X.
LOAD 1,[A] 6	LOAD THE 6 <sup>TH</sup> LOCATION OF ARRAY A INTO REGISTER 1.
STORE 1,[B] <C>	STORE REGISTER 1 INTO LOCATION C WITHIN SEGMENT B.

Figure 2.7: Example of a segment

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

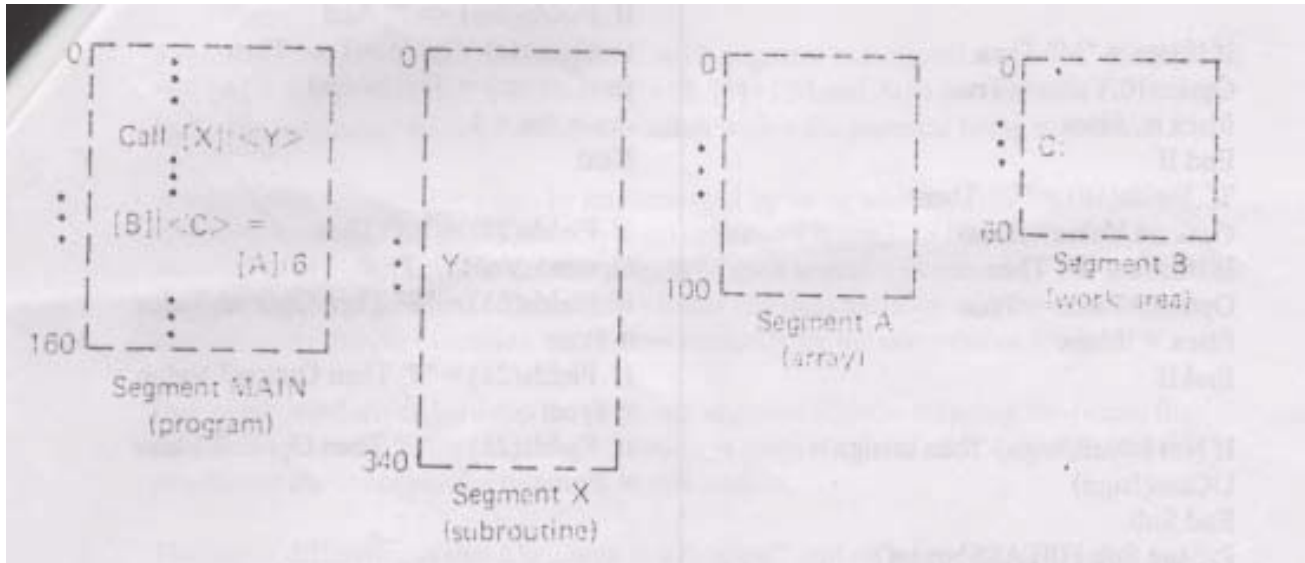


Figure 2.8: Segment address space

Source: William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Although the segmented address space is explicit and “visible” to the programmer, the differences between this space and conventional address space need not affect his programming. In most programming languages, there are statements, such as:

CALL Y;

C = A(6);

In a single linear address space the compiler and/or loader must convert these segment like references into a single address. For example, if Y is an entry point at location 120 within subroutine X and subroutine X is loaded at location 3460, then CALL Y becomes CALL 3580. Likewise, if A is an array of bytes at location 3800, A(6) refers to location 3806. Notice that the original structure is lost in this transformation. If an error occurred at location 3803, there is no explicit indication of the fact that 3803 is really A(3).

Typically, in a segmented address space, each segment is assigned a number, such as [X] = 3, [A] = 5. Then, CALL Y becomes CALL [3] | 120 and A(6) becomes [5] | 6, explicitly indicating the segment and location within the segment being referenced.

A segmented address space can be implemented by using address mapping hardware similar to that used for paged memory management. Due to the similarities between the page and segment address mapping hardware, the distinction between paging and segmentation is often confused. We will attempt to emphasize the conceptual differences in this section.



The major difference is that a segment is a “logical” unit of information – visible to the user’s program and is arbitrary size. A page is a “physical” unit of information – strictly used for memory management, invisible to the user’s program – and is of a fixed size, e.g 4 k bytes. In retrospect, a segment is a more precise form of the job “Portion” notion such as the multiple allocation scheme.

### **Advantages**

- It eliminates fragmentation.
- It provides virtual memory.
- It allows dynamic segment growth.
- It facilitates shared segments.

### **Disadvantages**

- Considerable compaction overhead is incurred in order to support dynamic segment growth and eliminate fragmentation.
- There is difficulty in managing variable size segments on secondary storage.
- The maximum size of a segment is limited by the size of main memory.

## **3.6 Translation lookaside buffer (TLB)**

A Translation lookaside buffer (TLB) is a CPU cache that memory management hardware uses to improve virtual address translation speed. It was the first cache introduced in processors. All current desktop and server processors (such as x86) use a TLB. A TLB has a fixed number of slots that contain page table entries, which map virtual addresses to physical addresses. It is typically a content-addressable memory (CAM), in which the search key is the virtual address and the search result is a physical address. If the requested address is present in the TLB, the CAM search yields a match quickly, after which the physical address can be used to access memory. This is called a TLB hit. If the requested address is not in the TLB, the translation proceeds by looking up the page table in a process called a page walk. The page walk is a high latency process, as it involves reading the contents of multiple memory locations and using them to compute the physical address. Furthermore, the page walk takes significantly longer if the translation tables are swapped out into secondary storage, which a few systems allow. After the physical address is determined, the virtual address to physical address mapping and the protection bits are entered in the TLB.

The TLB references physical memory addresses in its table. It may reside between the CPU and the CPU cache or between the CPU cache and primary storage memory. This depends on whether the cache uses physical or virtual addressing. If the cache is virtually addressed, requests are sent directly from the CPU to the cache, which then accesses the TLB as necessary. If the cache is physically addressed, the CPU does a TLB lookup on every memory operation, and the resulting physical address is sent to the cache. There are pros and cons to both implementations. A common optimization for physically addressed caches is to perform the TLB lookup in parallel with the cache access. The low-order bits of any virtual address (e.g., in a virtual memory system having 4KB pages, the lower 12 bits of the virtual address) represent the offset of the desired address within the page, and thus they do not change in the virtual-to-physical translation. During a cache access, two steps are performed: an index is used to find an entry in the cache's

data store, and then the tags for the cache line found are compared. If the cache is structured in such a way that it can be indexed using only the bits that do not change in translation, the cache can perform its "index" operation while the TLB translates the upper bits of the address. Then, the translated address from the TLB is passed to the cache. The cache performs a tag comparison to determine if this access was a hit or miss. It is possible to perform the TLB lookup in parallel with the cache access even if the cache must be indexed using some bits that may change upon address translation; see the address translation section in the cache article for more details about virtual addressing as it pertains to caches and TLBs.

### 3.7 Memory Partition Selection Algorithm

#### First-fit

An incoming process is placed in the first available hole which can accommodate it.

#### Best-fit

An incoming process is placed in a hole in which it fits mostly tightly, ie for all the choices of hole, the difference between the hole size and the new process size is least.

#### Worst-fit

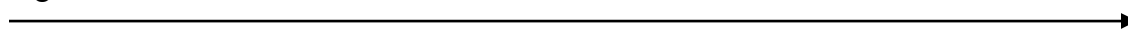
An incoming process is placed in the hole which leaves the maximum amount of unused space, which logically must be the current largest hole.

#### Example

Assume memory is allocated as specified in Fig. 2.1 below, beginning from left to right, before additional requests for 20K, 10K, and 5K (in that order) are received. With a corresponding resulting diagram, at what starting address will each of the additional requests be allocated in the case of: First-fit, Best-fit, and Worst-fit?

Used	Hole	Used	Hole	Used	Hole	Used	Hole	Used	Hole	Used	Hole
10K	10K	20K	30K	10K	5K	30K	20K	10K	15K	20K	20K

Figure 2.9



#### Solution

##### First-fit

Allocation for 20K with starting address at location 40K

Used	Hole	Used	<b>U</b>	<b>H</b>	Used	Hole	Used	Hole	Used	Hole	Used	Hole
10K	10K	20K	<b>20k</b>	<b>10k</b>	10K	5K	30K	20K	10K	15K	20K	20K

Allocation for 10k with starting address at location 10K

Used	<b>U</b>	Used	U	H	Used	Hole	Used	Hole	Used	Hole	Used	Hole
10K	<b>10K</b>	20K	20k	10k	10K	5K	30K	20K	10K	15K	20K	20K

Allocation for 5k with starting address at location 60K

Used	<b>U</b>	Used	U	<b>U</b>	<b>H</b>	Used	Hole	Used	Hole	Used	Hole	Used	Hole
10K	<b>10K</b>	20K	20k	<b>5k</b>	<b>5k</b>	10K	5K	30K	20K	10K	15K	20K	20K

**Best - fit:** Allocation for 20k with starting address at location 115K

Used	Hole	Used	Hole	Used	Hole	Used	<b>Used</b>	Used	Hole	Used	Hole
10K	10K	20K	30K	10K	5K	30K	<b>20K</b>	10K	15K	20K	20K

Allocation for 10k with starting address at location 10K

Used	<b>Used</b>	Used	Hole	Used	Hole	Used	Used	Used	Hole	Used	Hole
10K	<b>10K</b>	20K	30K	10K	5K	30K	20K	10K	15K	20K	20K

Allocation for 5k with starting address at location 80K

Used	Used	Used	Hole	Used	<b>Used</b>	Used	Used	Used	Hole	Used	Hole
10K	10K	20K	30K	10K	<b>5K</b>	30K	20K	10K	15K	20K	20K

**Worst - fit**

Allocation for 20k with starting address at location 40K

Used	Hole	Used	<b>Used</b>	<b>Hole</b>	Used	Hole	Used	Hole	Used	Hole	Used	Hole
10K	10K	20K	<b>20K</b>	<b>10K</b>	10K	5K	30K	20K	10K	15K	20K	20K

Allocation for 10k with starting address at location 115K

Used	Hole	Used	Used	Hole	Used	Hole	Used	<b>Used</b>	<b>hole</b>	Used	Hole	Used	Hole
10K	10K	20K	20K	10K	10K	5K	30K	<b>10K</b>	<b>10k</b>	10K	15K	20K	20K

Allocation for 5k with starting address at location 180K

Used	Hole	Used	Used	Hole	Used	Hole	Used	Used	ole	Used	Hole	Used	<b>Used</b>	<b>hole</b>
10K	10K	20K	20K	10K	10K	5K	30K	10K	10k	10K	15K	20K	<b>5K</b>	<b>15k</b>

### Activity A

- Highlight the main features of memory allocation schemes.
- What do you understand by each of the following memory partition selection algorithms: First-fit, Best-fit, and Worst-fit?

### 4.0 Conclusion

Memory management policies exist for managing single and multiple allocation schemes.

## 5.0 Summary

In this unit, we have learnt that:


- (a) the objectives of memory management include relocation, protection, sharing, logical and physical organization.
- (b) the activities of memory management involves allocating and deallocating memory space as needed.
- (c) the policies used by memory management include: single contiguous, partition, relocation partition, paging and segmentation.

## 6.0 Tutor Marked Assignment

1. Assume memory is allocated as specified in figure 2.9 below, beginning from left to right, before additional requests for 20K, 10K, and 5K (in that order) are received. With a corresponding resulting diagram, at what starting address will **each** of the additional requests be allocated in the case of: **First-fit, Best-fit, and Worst-fit?**

Used	Hole	Used	Hole	Used	Hole	Used	Hole	Used	Hole	Used	Hole
10K	10K	20K	30K	10K	5K	30K	20K	10K	15K	20K	20K

Figure 2.9

- 
2. What is segmentation? How is the problem solved?

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Hennessy J and Patterson D. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2002.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

Translation lookaside buffer (TLB), From Wikipedia, the free encyclopedia, 2009. [http://en.wikipedia.org/wiki/Translation\\_lookaside\\_buffer](http://en.wikipedia.org/wiki/Translation_lookaside_buffer).

## MODULE 4: DEBUGGING AND MEMORY MANAGEMENT

### UNIT 3: VIRTUAL MEMORY

	Page
1.0	Introduction
2.0	Objectives
3.0	Virtual memory concepts
3.1	Page replacement algorithm for single process
3.2	Thrashing
3.3	Page replacement algorithm for multiple process
4.0	Conclusion
5.0	Summary
6.0	Tutor marked Assignment
7.0	Further Reading and Other Resources

#### 1.0 Introduction

The term virtual memory is usually associated with systems that employ paging although virtual memory based on segmentation is also used. This unit examines various virtual memory concepts and page replacement algorithm for single and multiple processes.

#### 2.0 Objectives

At the end of this unit, the reader should be able to:

- (a) Know the virtual memory concepts.
- (b) Know the page replace algorithms for single and multiple processes.
- (c) Distinguish between trashing and page fault.

#### 3.0 Virtual memory concepts

Virtual Memory refers to the concept whereby a process with a larger size than available memory can be loaded and executed by loading the process in parts. The program memory is divided into pages and the available physical memory into frames. The page size is always equal to the frame size. The page size is generally in a power of 2 to avoid the calculation involved to get the page number and the offset from the CPU generated address. The virtual address contains a page number and an offset. This is mapped to the physical address by a technique of address resolution after searching the Page Map Table.

#### Paging

In Virtual Memory Systems, a program in execution or a process is divided into equal sized logical blocks called pages that are loaded into frames in the main memory. The size of a page is always in a power of 2 and is equal to the frame size. Dividing the process into pages allows non-contiguous allocation in these systems.

#### Segmentation

Segmentation is a memory management technique that supports Virtual Memory. The available memory is divided into segments and consists of two components- a base address that denotes the address of the base of that segment and a displacement value that refers to the length of an

address location from the base of that segment. The effective physical address is the sum of the base address value and the length of the displacement value.

### **Page Fault**

A Page Fault occurs when there is a request for a page that is not available in the main memory. The Page Map Table for such a page has its presence bit not set. When a page fault occurs, the Operating System schedules a disk read operation to retrieve the page from the secondary storage and load the same to the main memory.

### **Cache Memory**

Cache Memory is a high speed memory in the Random Access Memory (RAM). The processor looks for data first in the Cache Memory and then depending on whether there is a Cache hit or miss, searches for the same in other parts of the primary memory.

A Cache hit indicates that the data searched for in the Cache by the CPU is available. The reverse is Cache miss. Typically the size of the Cache in a system is limited and varies depending on the system's configuration.

### **Demand Paging**

In Virtual Memory Systems the pages are not loaded in memory until they are "demanded" by a process; therefore the term, demand paging. Demand paging allows the various parts of a process to be brought into physical memory as the process needs them to execute. This normally involves a memory management unit which looks up the virtual address in a page map to see if it is paged in. If it is not then the operating system will page it in, update the page map and restart the failed access. This implies that the processor must be able to recover from and restart a failed memory access or must be suspended while some other mechanism is used to perform the paging.

Paging in a page may first require some other page to be moved from main memory to disk ("paged out") to make room. If this page has not been modified since it was paged in, it can simply be reused without writing it back to disk. This is determined from the "modified" or "dirty" flag bit in the page map. A replacement algorithm or policy is used to select the page to be paged out, often this is the least recently used (LRU) algorithm.

In computer operating systems, demand paging is an application of virtual memory. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it (i.e., if a page fault occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory. This is an example of a lazy loading techniques.

### **Advantages of Demand Paging:**

- It does not load the pages that are never accessed, so it saves the memory for other programs and increases the degree of multiprogramming.
- It has less loading latency at the program startup.
- It has less disk overhead because of fewer page reads.
- Pages will be shared by multiple programs until they are modified by one of them, so a technique called copy on write will be used to save more resources.
- It has ability to run large programs on the machine, even though it does not have sufficient memory to run the program. This method is better than an old technique called overlays.
- It does not need extra hardware support than what paging needs, since protection fault can be used to get page fault.

### **Disadvantages:**

- Individual programs face extra latency when they access a page for the first time. So prepaging, a method of remembering which pages a process used when it last executed and preloading a few of them, is used to improve performance.
- Memory management with page replacement algorithms becomes slightly more complex.
- Possible security risks, including vulnerability to timing attacks.

### **Prepaging**

A technique whereby the operating system in a paging virtual memory multitasking environment loads all pages of a process's working set into memory before the process is restarted.

Under demand paging a process accesses its working set by page faults every time it is restarted. Under prepaging, the system remembers the pages in each process's working set and loads them into physical memory before restarting the process. Prepaging reduces the page fault rate of reloaded processes and hence generally improves CPU efficiency. Prepaging is generally more efficient than demand paging.

#### **3.1 Page replacement algorithm for single process**

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms.

It is worth noting that the problem of "page replacement" occurs in other areas of computer design as well. For example, most computers have one or more memory caches consisting of recently used 32-byte or 64-byte memory blocks. When the cache is full, some block has to be chosen for removal. This problem is precisely the same as page replacement except on a shorter time scale (it has to be done in a few nanoseconds, not milliseconds as with page replacement).

A second example is in a Web server. The server can keep a certain number of heavily used Web pages in its memory cache. However, when the memory cache is full and a new page is referenced, a decision has to be made which Web page to evict. The considerations are similar to pages of virtual memory, except for the fact that the Web pages are never modified in the cache, so there is always a fresh copy on disk.

### **Not recently used**

The not recently used (NRU) page replacement algorithm is an algorithm that favours keeping pages in memory that have been recently used. This algorithm works on the following principle: when a page is referenced, a referenced bit is set for that page, marking it as referenced. Similarly, when a page is modified (written to), a modified bit is set. The setting of the bits is usually done by the hardware, although it is possible to do so on the software level as well.

At a certain fixed time interval, the clock interrupt triggers and clears the referenced bit of all the pages, so only pages referenced within the current clock interval are marked with a referenced bit. When a page needs to be replaced, the operating system divides the pages into four classes:

- Class 0: not referenced, not modified
- Class 1: not referenced, modified
- Class 2: referenced, not modified
- Class 3: referenced, modified

Although it does not seem possible for a page to be not referenced yet modified, this happens when a category 3 page has its referenced bit cleared by the clock interrupt. The NRU algorithm picks a random page from the lowest category for removal. Note that this algorithm implies that a referenced page is more important than a modified page.

### **First-in, first-out**

The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little book keeping on the part of the operating system. The idea is obvious from the name - the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the earliest arrival in front. When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected. While FIFO is cheap and intuitive, it performs poorly in practical application.

### **Second-chance**

A modified form of the FIFO page replacement algorithm, known as the Second-chance page replacement algorithm, fares relatively better than FIFO at little cost for the improvement. It works by looking at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set. If it is not set, the page is swapped out. Otherwise, the referenced bit is cleared, the page is inserted at the back of the queue (as if it were



a new page) and this process is repeated. This can also be thought of as a circular queue. If all the pages have their referenced bit set, on the second encounter of the first page in the list, that page will be swapped out, as it now has its referenced bit cleared.

As its name suggests, Second-chance gives every page a "second-chance" - an old page that has been referenced is probably in use, and should not be swapped out over a new page that has not been referenced.

The term job or process is defined as a program in execution or an instance of a program running on a computer.

The operation of this algorithm, called second chance, is shown in Figure 1. In Figure 1(a) we see pages A through H kept on a linked list and sorted by the time they arrived in memory. Figure 1 shows operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their loading times.

Suppose that a page fault occurs at time 20. The oldest page is A, which arrived at time 0, when the process started. If A has the R bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is clean). On the other hand, if the R bit is set, A is put onto the end of the list and its "load time" is reset to the current time (20). The R bit is also cleared. The search for a suitable page continues with B.

What second chance is doing is looking for an old page that has not been referenced in the previous clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO. Specifically, imagine that all the pages in Figure 1(a) have their R bits set. One by one, the operating system moves the pages to the end of the list, clearing the R bit each time it appends a page to the end of the list. Eventually, it comes back to page A, which now has its R.

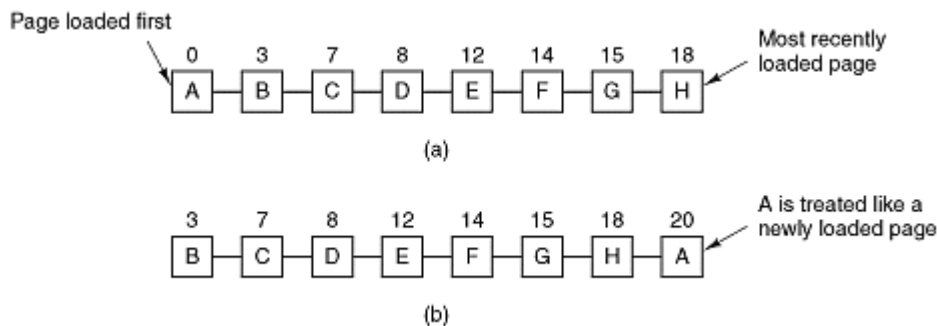


Figure 3.1: Operation of Second chance

Source: Andrew S. Tanenbaum, 2002. Available online at: <http://www.informit.com/articles/article.aspx?p=25260&seqNum=6>

## RAND

(Random) Simply pick a random frame. This algorithm is also pretty bad.

## OPT

(Optimum) Pick the frame whose page will not be used for the longest time in the future. If there is a page in memory that will never be used again, it's frame is obviously the best choice for replacement. Otherwise, if (for example) page A will be next referenced 8 million instructions in the future and page B will be referenced 6 million instructions in the future, choose page A. This algorithm is sometimes called Belady's MIN algorithm after its inventor. It can be shown that OPT is the best possible algorithm, in the sense that for any reference string (sequence of page numbers touched by a process), OPT gives the smallest number of page faults. Unfortunately, OPT, like SJF processor scheduling, is unimplementable because it requires knowledge of the future. Its only use is as a theoretical limit. If you have an algorithm you think looks promising, see how it compares to OPT on some sample reference strings.

## Clock

Clock is a more efficient version of FIFO than Second-chance because pages don't have to be constantly pushed to the back of the list, but it performs the same general function as Second-Chance. The clock algorithm keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the oldest page in the list. When a page fault occurs and no empty frames exist, then the R (referenced) bit is inspected at the hand's location. If R is 0, the new page is put in place of the page the "hand" points to, otherwise the R bit is cleared and the hand is incremented. This is repeated until a page is replaced. Figure 2 shows operation of clock.

A hand points to the oldest page. When a page fault occurs, the page being pointed to by the hand is inspected. If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If R is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R = 0. Not surprisingly, this algorithm is called clock. It differs from second chance only in the implementation.

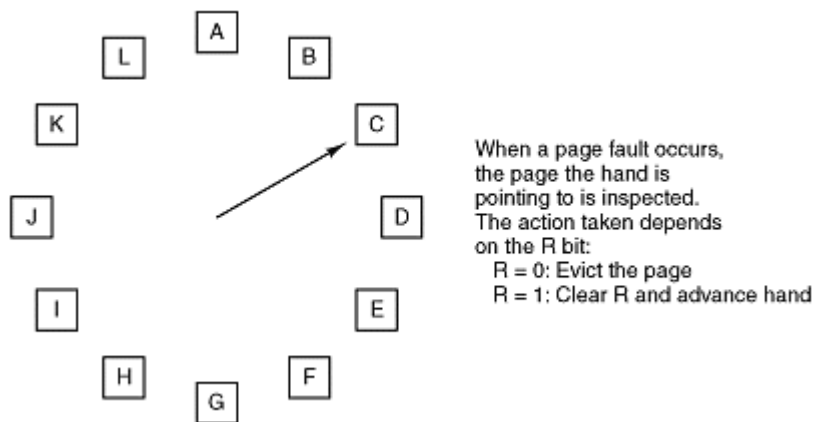


Figure 3.2: Operation of clock

Source: Andrew S. Tanenbaum, 2002. Available online at:  
<http://www.informit.com/articles/article.aspx?p=25260&seqNum=6>

### Least recently used

The least recently used page (LRU) replacement algorithm, though similar in name to NRU, differs in the fact that LRU keeps track of page usage over a short period of time, while NRU just looks at the usage in the last clock interval. LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory, it is rather expensive to implement in practice. There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible.

The most expensive method is the linked list method, which uses a linked list containing all the pages in memory. At the back of this list is the least recently used page, and at the front is the most recently used page. The cost of this implementation lies in the fact that items in the list will have to be moved about every memory reference, which is a very time-consuming process.

Another method that requires hardware support is as follows: suppose the hardware has a 64-bit counter that is incremented at every instruction. Whenever a page is accessed, it gains a value equal to the counter at the time of page access. Whenever a page needs to be replaced, the operating system selects the page with the lowest counter and swaps it out. With present hardware, this is not feasible because the required hardware counters do not exist.

Because of implementation costs, one may consider algorithms (like those that follow) that are similar to LRU, but which offer cheaper implementations.

One important advantage of LRU algorithm is that it is amenable to full statistical analysis.

On the other hand, LRU's weakness is that its performance tends to degenerate under many quite common reference patterns. Figure 3 shows operation of LRU.

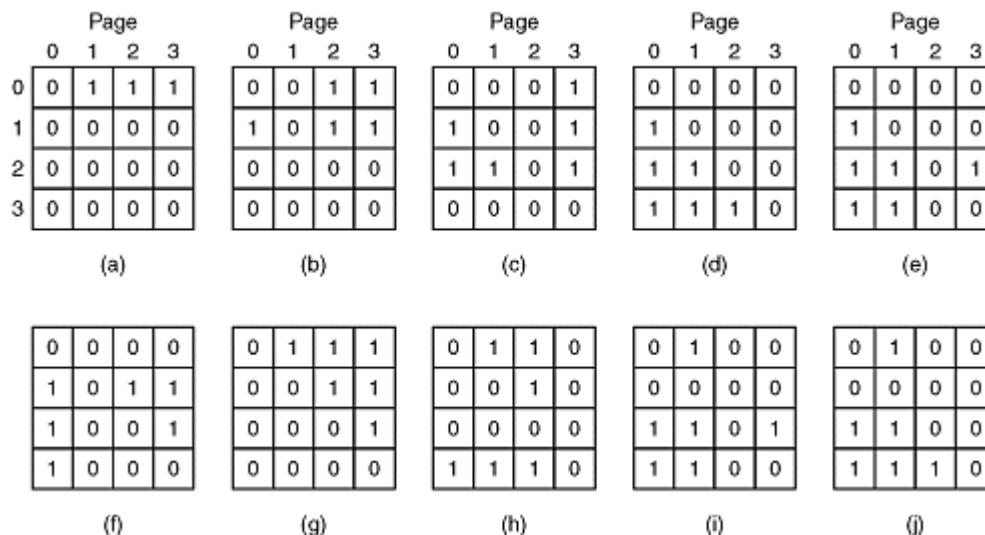


Figure 3.3: Operation of LRU

Source: Andrew S. Tanenbaum, 2002. Available online at:

<http://www.informit.com/articles/article.aspx?p=25260&seqNum=6>

## Summary of Page Replacement Algorithms

We have now looked at a variety of page replacement algorithms. In this section we will briefly summarize them. The list of algorithms discussed is given in Table 1.

The optimal algorithm replaces the page referenced last among the current pages. Unfortunately, there is no way to determine which page will be last, so in practice this algorithm cannot be used. It is useful as a benchmark against which other algorithms can be measured, however.

The **NRU** algorithm divides pages into four classes depending on the state of the R and M bits. A random page from the lowest numbered class is chosen. This algorithm is easy to implement, but it is very crude. Better ones exist.

**FIFO** keeps track of the order pages were loaded into memory by keeping them in a linked list. Removing the oldest page then becomes trivial, but that page might still be in use, so FIFO is a bad choice.

**Second chance** is a modification to FIFO that checks if a page is in use before removing it. If it is, the page is spared. This modification greatly improves the performance. Clock is simply a different implementation of second chance. It has the same performance properties, but takes a little less time to execute the algorithm.

**LRU** is an excellent algorithm, but it cannot be implemented without special hardware. If this hardware is not available, it cannot be used. NFU is a crude attempt to approximate LRU. It is not very good. However, aging is a much better approximation to LRU and can be implemented efficiently. It is a good choice.

Table 3.1: Summary of page replacement algorithms for single processes

Source: Andrew S. Tanenbaum, 2002. Available online at:  
<http://www.informit.com/articles/article.aspx?p=25260&seqNum=6>

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU

## 3.2 Thrashing

Thrashing is a degenerate case that occurs when there is insufficient memory at one level in the memory hierarchy to properly contain the working set required by the upper levels of the memory hierarchy. This can result in the overall performance of the system dropping to the speed of a lower level in the memory hierarchy. Therefore, thrashing can quickly reduce the

performance of the system to the speed of main memory or, worse yet, the speed of the disk drive.

There are two primary causes of thrashing: (1) insufficient memory at a given level in the memory hierarchy, and (2) the program does not exhibit locality of reference. If there is insufficient memory to hold a working set of pages or cache lines, then the memory system is constantly replacing one block (cache line or page) with another. As a result, the system winds up operating at the speed of the slower memory in the hierarchy. A common example occurs with virtual memory. A user may have several applications running at the same time and the sum total of these programs' working sets is greater than all of physical memory available to the program. As a result, as the operating system switches between the applications it has to copy each application's data to and from disk and it may also have to copy the code from disk to memory. Since a context switch between programs is often much faster than retrieving data from the disk, this slows the programs down by a tremendous factor since thrashing slows the context switch down to the speed of swapping the applications to and from disk.

If the program does not exhibit locality of reference and the lower memory subsystems are not fully associative, then thrashing can occur even if there is free memory at the current level in the memory hierarchy. For example, suppose an eight kilobyte L1 caching system uses a direct-mapped cache with 16-byte cache lines (i.e., 512 cache lines). If a program references data objects 8K apart on each access then the system will have to replace the same line in the cache over and over again with each access. This occurs even though the other 511 cache lines are currently unused.

If insufficient memory is the cause of thrashing, an easy solution is to add more memory (if possible, it is rather hard to add more L1 cache when the cache is on the same chip as the processor). Another alternative is to run fewer processes concurrently or modify the program so that it references less memory over a given time period. If lack of locality of reference is causing the problem, then you should restructure your program and its data structures to make references local to one another.

Thrashing is a condition that indicates that due to excessive paging a particular process is in the halted state or executing very slowly. It is a condition in which a multi-programmed environment is equivalent to a mono-programmed environment. The causes of thrashing can be attributed to one or more of the following:

1. Increase in the degree of multi-programming.
2. Insufficient memory at a particular point of time.
3. The program does not exhibit locality of reference.

Thrashing can be reduced by analyzing the CPU utilization and reducing the degree of multi-programming, which in turn is a non-negative integer that indicates how many programs are in the memory at the same point in time in a multi-programmed environment waiting for its turn to get the processor.

**Clean pages** – refers to pages that have not been modified since it was read into memory.

**Dirty pages** – refers to changed or modified pages.

How does the operating systems handle page fault? Same as basic page replacement process

**When a page fault occurs, the operating system will:**

1. Find the location of the desired page on disk
2. Find a free frame
  - a. If a free frame exists, use it.
  - b. If no free frame exist, use a page replacement algorithm to find victim frame
3. Read the desired page into the freed frame.
4. Update the page frame table
5. Restart the process

### **3.3 Page replacement algorithm for multiple process**

Up to this point, we have been assuming that there is only one active process. When there are multiple processes, things get more complicated. Algorithms that work well for one process can give terrible results if they are extended to multiple processes in a naive way.

LRU would give excellent results for a single process, and all of the good practical algorithms can be seen as ways of approximating LRU. A straightforward extension of LRU to multiple processes still chooses the page frame that has not been referenced for the longest time. However, that is a lousy idea. Consider a workload consisting of two processes. Process A is copying data from one file to another, while process B is doing a CPU-intensive calculation on a large matrix. Whenever process A blocks for I/O, it stops referencing its pages. After a while process B steals all the page frames away from A. When A finally finishes with an I/O operation, it suffers a series of page faults until it gets back the pages it needs, then computes for a very short time and blocks again on another I/O operation.

Various specific algorithms have been proposed. As in the single process case, some are theoretically good but unimplementable, while others are easy to implement but bad. The trick is to find a reasonable compromise.

#### **Fixed Allocation**

Give each process a fixed number of page frames. When a page fault occurs use LRU or some approximation to it, but only consider frames that belong to the faulting process. The trouble with this approach is that it is not at all obvious how to decide how many frames to allocate to each process. If you give a process too few frames, it will thrash. If you give it too many, the extra frames are wasted; you would be better off giving those frames to another process, or starting another job (in a batch system). In some environments, it may be possible to statically estimate the memory requirements of each job. For example, a real-time control system tends to run a fixed collection of processes

for a very long time. The characteristics of each process can be carefully measured and the system can be tuned to give each process exactly the amount of memory it needs. Fixed allocation has also been tried with batch systems: Each user is required to declare the memory allocation of a job when it is submitted. The customer is charged both for memory allocated and for I/O traffic, including traffic caused by page faults. The idea is that the customer has the incentive to declare the optimum size for his job. Unfortunately, even assuming good will on the part of the user, it can be very hard to estimate the memory demands of a job. Besides, the working-set size can change over the life of the job.

### **Page-Fault Frequency (PFF)**

This approach is similar to fixed allocation, but the allocations are dynamically adjusted. The OS continuously monitors the fault rate of each process, in page faults per second of virtual time. If the fault rate of a process gets too high, either give it more pages or swap it out. If the fault rate gets too low, take some pages away. When you get back enough pages this way, either start another job (in a batch system) or restart some job that was swapped out. This technique is actually used in some existing systems. The problem is choosing the right values of “too high” and “too low.” You also have to be careful to avoid an unstable system, where you are continually stealing pages from a process until it thrashes and then giving them back.

### **Working Set**

The Working Set (WS) algorithm (as contrasted with the working set model) is as follows: Constantly monitor the working set (as defined above) of each process. Whenever a page leaves the working set, immediately take it away from the process and add its frame to a pool of free frames. When a process page faults, allocate it a frame from the pool of free frames. If the pool becomes empty, we have an overload situation--the sum of the working set sizes of the active processes exceeds the size of physical memory--so one of the processes is stopped. The problem is that WS, like SJF or true LRU, is not implementable. A page may leave a process' working set at any time, so the WS algorithm would require the working set to be monitored on every single memory reference. That's not something that can be done by software, and it would be totally impractical to build special hardware to do it. Thus all good multi-process paging algorithms are essentially approximations to WS.

### **Clock**

Some systems use a global CLOCK algorithm, with all frames, regardless of current owner, included in a single clock. As we said above, CLOCK approximates LRU, so global CLOCK approximates global LRU, which, as we said, is not a good algorithm. However, by being a little careful, we can fix the worst failing of global clock. If the clock “hand” is moving too “fast” (i.e., if we have to examine too many frames before finding one to replace on an average call), we can take that as evidence that memory is over-committed and swap out some process.

## Activity A

- a) Explain the concept of virtual memory.
- b) List and explain three page replacement algorithms for single process and two for multiple processes.
- c) What happens during a page fault?
- d) What happens during trashing (in the memory system) and why it happens?

## 4.0 Conclusion

Memory management policies exist for managing single and multiple allocation schemes. Clean pages refers to pages that have not been modified since it was read into memory. Dirty pages refers to changed or modified pages.

## 5.0 Summary

In this unit, we have learnt:

- a) that the page replacement algorithms for single are: NRU, FIFO, Second chance, Clock, LRU; and for multiple processes are: fixed allocation, working set, clock.
- b) that the OS handle page fault by first finding the location of the desired page on disk and goes ahead to read the desired page into the freed memory frame.

## 6.0 Tutor Marked Assignment

- (a) Briefly explain the following:
  - i) dirty page ii) clean page iii) thrashing iv) demand paging v) compaction vi) coalescing
- (b) State and explain four (4) page replacement algorithm for single process and two (2) for multiple processes.

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Carr R. *Virtual memory management*. Ann Arbor, MI: UMI Research Press, 1984.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

Translation lookaside buffer (TLB), *From Wikipedia, the free encyclopedia*, 2009.  
[http://en.wikipedia.org/wiki/Translation\\_lookaside\\_buffer](http://en.wikipedia.org/wiki/Translation_lookaside_buffer)



## **CIT752: OPERATING SYSTEM CONCEPTS (2 UNITS)**

### **MODULE 5: DEVICE AND FILE MANAGEMENT**

#### **UNIT 1: INPUT/OUTPUT DEVICE MANAGEMENT**

	<b>Page</b>
1.0	Introduction
2.0	Objectives
3.0	Input/Output (I/O) Operations
3.1	Programmed I/O
3.2	Interrupt – Driven I/O
3.3	Direct Memory Access (DMA)
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### **1.0 Introduction**

Perhaps the difficult part of the design of an operating system deals with the I/O facility and the file management system. With respect to I/O, the key issue is performance. This unit examines I/O operations and the techniques used by the operating system for data exchange between the I/O, memory and the processor.

#### **2.0 Objectives**

At the end of this unit, the reader should be able to:

- (a) Know the operations of I/O.
- (b) Know the techniques used by I/O for data exchange among memory, processor and I/O: program I/O, Interrupt driven I/O and DMA.

#### **3.0 I/O Operations**

There are three techniques used for I/O operations:

- a. Programmed I/O
- b. Interrupt-Driven I/O, and
- c. Direct Memory Access (DMA)

#### **3.1 Programmed I/O**

With programmed I/O, data are exchanged between the CPU and I/O module. The CPU executes a program that gives it direct control of the I/O operations including sending a read or write command and transferring the data. When the CPU issues a command to the I/O module, it must wait until the I/O operation is complete. If the CPU is faster than the I/O module, the CPU time is wasted.

The I/O module takes no further action to alert the CPU. In particular, it does not interrupt the CPU. Thus it is the responsibility of the CPU to periodically check the status of the I/O module until it finds that the operation is complete.

### **3.2 Interrupt – Driven I/O**

With Interrupt-driven I/O, the CPU issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work. The problem with program I/O is that the CPU has to wait a long time for the I/O module to be ready for either reception or transmission of data. The CPU, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of performance of the entire system is severely degraded.

An alternative is for the CPU to issue an I/O command to a module and then request service when it is ready to exchange data with the CPU. The CPU then executes the data transfer, as before, and resumes its former processing.

With both programmed and Interrupt I/O, the CPU is responsible for extracting data from main memory for output and storing data in main memory for input. The alternative is known as Direct Memory Access (DMA).

#### **Drawbacks of programmed I/O and Interrupt-Driven I/O**

Interrupt-driven I/O, though more efficient than simple programmed I/O, it requires the active intervention of the CPU to transfer data between memory and an I/O module, any data transfer must traverse a path through the CPU. Thus both forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the CPU can test and service a device.
2. The CPU is tied up in managing an I/O transfer, a number of instructions must be executed for each I/O transfer.

There is a tradeoff between these two drawbacks. Consider the transfer of a block of data. Using simple programmed I/O, the CPU is dedicated to the task of I/O and can move data at a rather high rate at the cost of doing nothing else. Interrupt I/O frees up the CPU to some extent at the expense of I/O transfer rate. Nevertheless, both methods have an adverse impact on both CPU activity and I/O transfer rate.

When large volumes of data are to be moved, a more efficient technique is required: Direct Memory Access (DMA).

### 3.3 Direct Memory Access (DMA)

With this technique, the I/O module and the main memory exchange data directly without CPU involvement.

DMA involves an additional module on the system bus. The DMA module is capable of mimicking the CPU and indeed capable of taking over control of the system from the CPU. The technique works as follows: when the CPU wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested.
- The address of the I/O device involved.
- The starting location in memory to read from or write to.
- The number of words to be read or written.

The CPU then continues with other works. It has delegated this I/O operation to the DMA module, and that module will take care of it. The DMA module transfers the entire block of data, one word at a time directly to or from memory, without going to the CPU. When the transfer is complete, the DMA module sends an Interrupt signal to the CPU. Thus the CPU is involved only at the beginning and end of the transfer. Figure 1 shows the interaction between memory, CPU and I/O module.

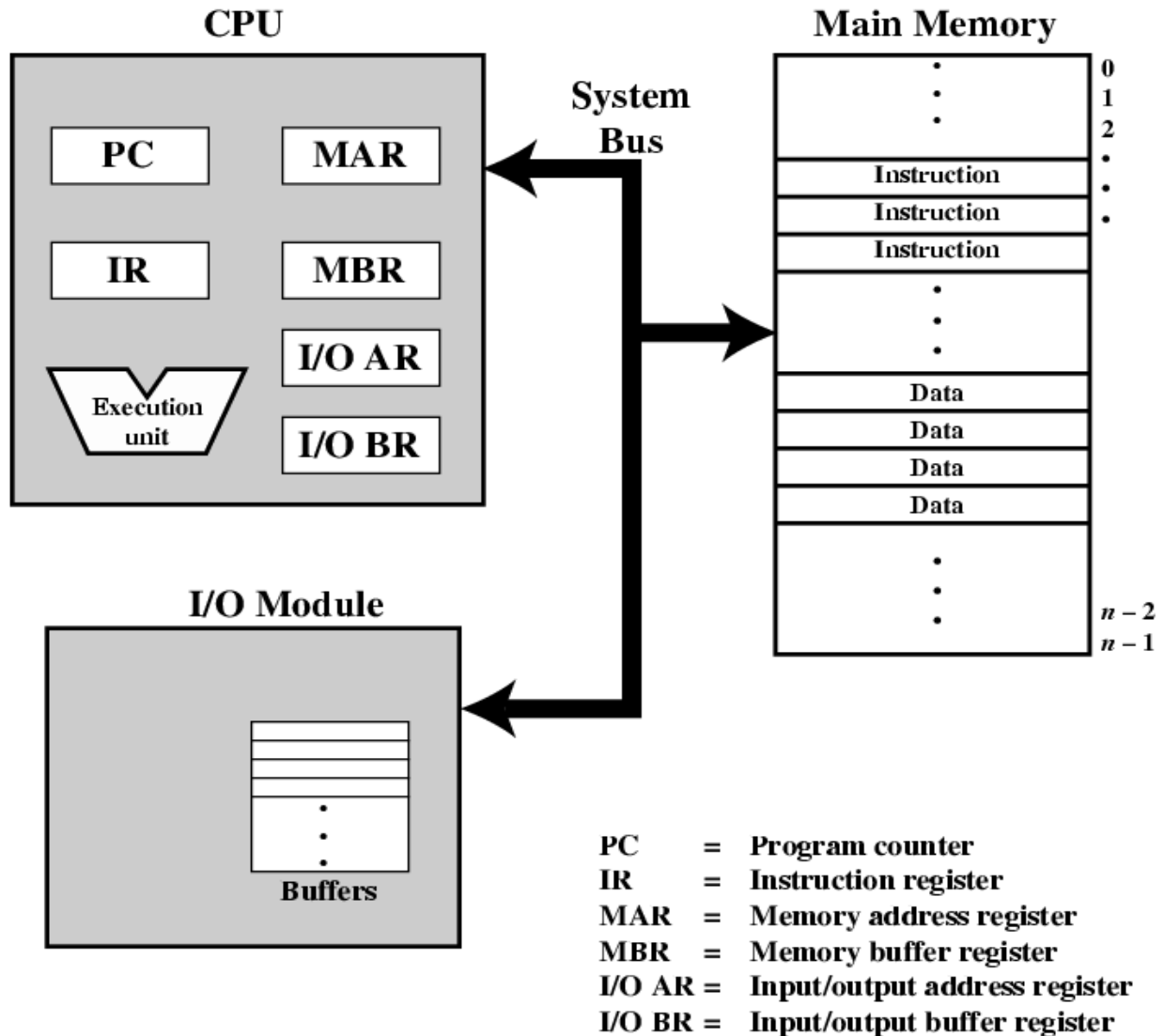


Figure 1.1: Data exchange between Memory, CPU and I/O module

Source: Module 2, Top-Level View of Computer Organization by Nguyen Thi Hoang Lan, 2009. Retrieved October 2009 from <http://cnx.org/content/m29708/latest/>

The CPU exchanges data with memory. For this purpose, it typically makes use of two internal registers: a memory address register (MAR), which specifies the address in memory for the next read or write, and a memory buffer register (MBR), which contains the data to be written into memory or receives the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the CPU.

**Program Counter (PC):** Contains the address of an instruction to be fetched.

**Instruction Register (IR):** Contains the instruction most recently fetched.

**Memory Address Register (MAR):** Contains the address of a location in memory.

**Memory Buffer Register:** Contains a word of data to be written to memory or the word most recently read.

**I/O address register (I/OAR):** Specifies a particular I/O device

**I/O buffer register (I/OBR):** Used for the exchange of data between an I/O module and the CPU.

A memory module consists of a set of locations, defined by sequentially numbered addresses. An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.

### **Activity A**

Describe the techniques of data exchange between I/O Module, main memory and CPU.

### **4.0 Conclusion**

With programmed I/O, data are exchanged between the CPU and I/O module. With Interrupt-driven I/O, the CPU issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work. In the DMA technique, the I/O module and the main memory exchange data directly without CPU involvement.

### **5.0 Summary**

In this unit, we have learnt:

the differences techniques of I/O operations, which include: programmed I/O, Interrupt-Driven I/O and DMA.

### **6.0 Tutor Marked Assignment**

Explain the difference between interrupt-driven I/O and DMA. State their advantages and disadvantages.

### **7.0 Further Reading and Other Resources**

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems, McGraw-Hill International editions*.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

Translation lookaside buffer (TLB), From Wikipedia, the free encyclopedia, 2009. [http://en.wikipedia.org/wiki/Translation\\_lookaside\\_buffer](http://en.wikipedia.org/wiki/Translation_lookaside_buffer).

Module 2, Top-Level View of Computer Organization by Nguyen Thi Hoang Lan, 2009. Retrived October 2009 from <http://cnx.org/content/m29708/latest/>

## **MODULE 5: DEVICE AND FILE MANAGEMENT**

### **UNIT 2: FILE CONCEPTS**

	<b>Page</b>
1.0	Introduction
2.0	Objectives
3.0	File concepts
3.1	Disk allocation and scheduling
3.2	Directory structure
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### **1.0 Introduction**

With file systems, performance is an issue. Other design requirements such as reliability and security also come into play. From a user's point of view, the file system is perhaps the most important aspect of the operating system. The user wants rapid access to files but also guarantees that the files are not corrupt and that they are secure from unauthorized access. Over the last 40 years, the increase in the speed of processors and main memory has far outstripped that for disk access, with processor and main memory speeds increasing by about two orders of magnitude compared to one order of magnitude for disk. This unit examines file concepts, disk allocation, scheduling and directory structure.

#### **2.0 Objectives**

At the end of this unit, the reader should be able to:

- (a) Know the attribute, operations and types of file.
- (b) Know the access method for files.
- (c) Know the disk allocation and scheduling methods.
- (d) Know the directory structure.

#### **3.0 File Concepts**

For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system. Computers can store information on several different storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit (the file). Files are mapped, by the operating system, onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots. A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary

storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, or alphanumeric etc. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general. The information in a file is defined by its creator. Many different types of information may be stored in a file – source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

### **File Attributes**

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as `example.c`. Some systems differentiate between upper and lowercase characters in names, whereas other systems consider the two cases to be equivalent. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file `example.c`, whereas another user might edit that file by specifying its name. The file's owner might write the file to a floppy disk, send it in an e-mail, or copy it across a network, and it could still be called `example.c` on the destination system. A file has certain other attributes, which vary from one operating system to another, but typically consists of these: Name, Identifier, Type, Location, Size, Protection, Time, date and user identification.

The information about all files is kept in the directory structure that also resides on secondary storage.

### **File Operations**

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. The following are five basic operations of the operating systems on files:

- Creating a file
- Writing a file
- Reading a file
- Deleting a file
- Truncating a file

These five basic operations certainly comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming the file.

### **File Types**

When we design a file system, we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.

A common technique for implementing file types is to include the type as a set of the file name. The name is split into two parts – a name and an extension usually separated by a period (see Table 2.1)

Table 2.1 Some common file types

File Type	Usual extension	Function
Executable	exe, com, bin	Read to run machine- Language program
Object	obj	Compiled, machine Language
Source code	c, cc, java, pas	Source code in Various language
Batch	bat, sh	Commands to the Command Interpreter
Text	txt, doc	Textual data, documents
Word processor	wp, tex, rrf, doc	Various word-processor formats
Library	lib, dll, mpeg	Libraries of routines for programmers
Archive	arc, zip, tar	Related files grouped into one file, Compressed, for archiving or storage
Multimedia	mpeg, mov, rm	Binary file containing audio

## ACCESS METHODS

Files store information, when it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems, support many access methods.

### Sequential File

In sequential file, records are stored one after the other in a predetermined form. This order is determined by the key fields on each record such as student matriculation number. An example is a magnetic tape where records/files are stored sequentially. To locate a particular record/file implies searching from the beginning of the sequentially file, one after the other until the record is located.

#### Advantages

- It is the simplest access method.
- It is easily stored on sequential access devices (ie tape) and random access devices (ie disk).

#### Disadvantages

- For interactive applications that involve queries or updates of individual records, the sequential file provides poor performance.
- Addition to the file also present problems.

### Direct File

Records in a direct file are not stored physically one after the other, rather, they are stored on a disk with a particular address or location that can be determined by their key field. The file allows programs to read and write records rapidly in no particular order. The direct access is based on disk model since disk allows random access to any file block.

#### Advantages

- Direct access are often used and are very efficient where very rapid access is required ie queries on database.
- It supports random access.



### **Disadvantage**

- Searching for a record may take a little time since there is no index

### **Index Sequential file**

This is a compromise between sequential and direct file. It stores records in a file in a sequential order but has an index associated with it. The index lists the key to each group of records stored and the corresponding disk address for that group. This is similar to the index at the end of a book that gives easy access to the content.

### **Advantage**

- The index to the file support random access.

### **Disadvantage**

- Creation of index table causes addition overhead

## **3.1 Disk allocation and scheduling**

In multiprogramming systems several different processes may want to use the system's resources simultaneously. For example, processes will contend to access an auxiliary storage device such as a disk. The disk drive needs some mechanism to resolve this contention, sharing the resource between the processes fairly and efficiently.

A magnetic disk consists of a collection of platters which rotate on a central spindle. These platters are metal disks covered with magnetic recording material on both sides. Each disk surface is divided into concentric circles called *tracks*. Each track is divided into *sectors* where information is stored. The reading and writing device, called the *head*, moves over the surface of the platters until it finds the track and sector it requires. This is like finding someone's home by first finding the street (track) and then the particular house number (sector). There is one head for each surface on which information is stored each on its own *arm*. In most systems the arms are connected together so that the heads move in unison, so that each head is over the same track on each surface. The term *cylinder* refers to the collection of all tracks which are under the heads at any time. Figure 1 and 2 shows a typical structure of a hard disk and a platter.

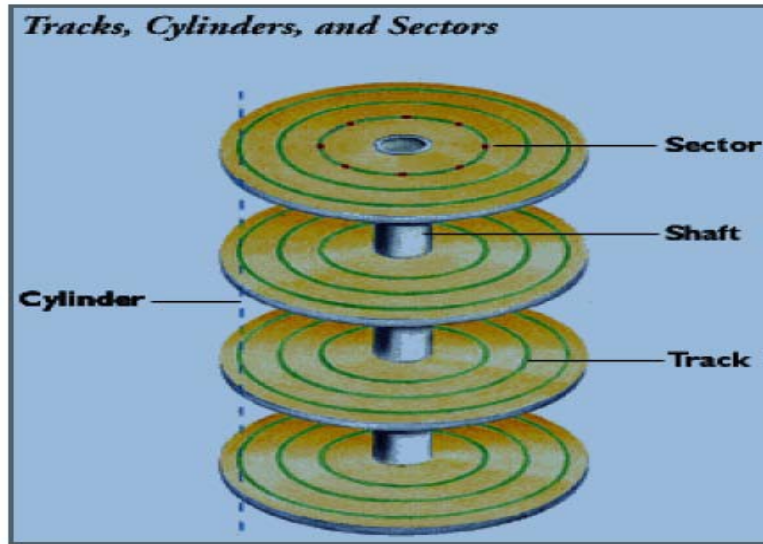


Figure 2.1: The typical structure of a hard disk  
Source: CSc33200: Operating Systems, CS-CCNY, Fall 2003 Jinzhong Niu December 8, 2003

A hard disk is usually made up of multiple platters, as illustrated in Figure 1, each of which use two heads to write and read data, one for the top of the platter and one for the bottom (this is not always the case, but usually is).

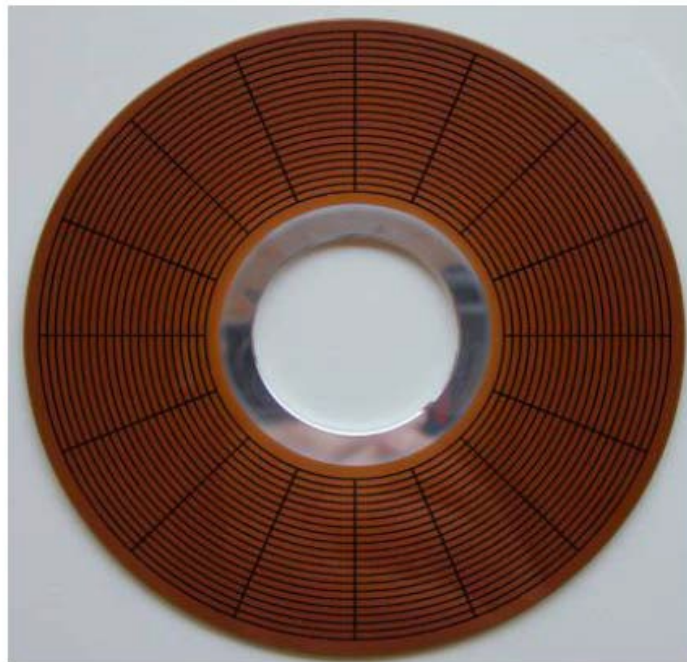


Figure 2.2: The layout of one side of a platter  
Source: CSc33200: Operating Systems, CS-CCNY, Fall 2003 Jinzhong Niu December 8, 2003

Either side of each platter is made up of multiple tracks, which in turn are divided into several sectors, as depicted in Figure 2.2.

The heads that access the platters are locked together on an assembly of head arms, which may move in only two directions towards the spindle or the opposite. This means that all the heads move in and out together, so each head is always physically located at the same track number. It is not possible to have one head at track 0 and another at track 1,000. Besides the movement of the heads, the spindle may rotate so that the heads may access a specific sector on the track they are located at. And because of this arrangement, often the track location of the heads is not referred to as a track number but rather as a cylinder number. A cylinder is basically the set of all tracks that all the heads are currently located at. The addressing of individual sectors of the disk is traditionally done by referring to cylinders, heads and sectors.

In order to satisfy an I/O request the disk controller must first move the head to the correct track and sector. Moving the head between cylinders takes a relatively long time so in order to maximise the number of I/O requests which can be satisfied the scheduling policy should try to minimise the movement of the head. On the other hand, minimising head movement by always satisfying the request of the closest location may mean that some requests have to wait a long time. Thus, there is a trade-off between *throughput* (the average number of requests satisfied in a unit time) and *response time* (the average time between a request arriving and it being satisfied). The various disk scheduling policies include:

### **First Come First Served (FCFS)**

The disk controller processes the I/O requests in the order in which they arrive, thus moving backwards and forwards across the surface of the disk to get to the next requested location each time. Since no reordering of requests take place the head may move almost randomly across the surface of the disk. This policy aims to minimise response time with little regard for throughput.

### **Shortest Seek Time First (SSTF)**

Each time an I/O request has been completed the disk controller selects the waiting request whose sector location is closest to the current position of the head. The movement across the surface of the disk is still apparently random but the time spent in movement is minimised. This policy will have better throughput than FCFS but a request may be delayed for a long period if many closely located requests arrive just after it.

### **SCAN**

The drive head sweeps across the entire surface of the disk, visiting the outermost cylinders before changing direction and sweeping back to the innermost cylinders. It selects the next waiting requests whose location it will reach on its path backward and forward across the disk. Thus, the movement time should be less than FCFS but the policy is clearly fairer than SSTF.

## **Circular SCAN (C-SCAN)**

C-SCAN is similar to SCAN but I/O requests are only satisfied when the drive head is travelling in one direction across the surface of the disk. The head sweeps from the innermost cylinder to the outermost cylinder satisfying the waiting requests in order of their locations. When it reaches the outermost cylinder it sweeps back to the innermost cylinder without satisfying any requests and then starts again.

## **LOOK**

Similarly to SCAN, the drive sweeps across the surface of the disk, satisfying requests, in alternating directions. However the drive now makes use of the information it has about the locations requested by the waiting requests. For example, a sweep out towards the outer edge of the disk will be reversed when there are no waiting requests for locations beyond the current cylinder.

## **Circular LOOK (C-LOOK)**

Based on C-SCAN, C-LOOK involves the drive head sweeping across the disk satisfying requests in one direction only. As in LOOK, the drive makes use of the location of waiting requests in order to determine how far to continue a sweep, and where to commence the next sweep. Thus it may curtail a sweep towards the outer edge when there are locations requested in cylinders beyond the current position, and commence its next sweep at a cylinder which is not the innermost one, if that is the most central one for which a sector is currently requested.

## **3.2 Directory structure**

The directory structure records information such as name, location, size and type for all files stored on the storage devices (eg hard disk). When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory such as:

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file, etc

The various directory structures include:

### **1. Single-Level directory**

The simplest directory structure is the single level directory. All files are contained in the same directory, which is easy to support and understand (see Figure 3). This is very common on single-user OS. A single-level directory has significant limitations, however, when the number of files increases or when there is more than one user. Since all files are in the same directory, they must have unique names. If there are two users who call their data file "test", then the unique-

name rule is violated. Although file names are generally selected to reflect the content of the file, they are often quite limited in length. Even with a single-user, as the number of files increases, it becomes difficult to remember the names of all the files in order to create only files with unique names. There is no operating system that currently support single-level directory structure.

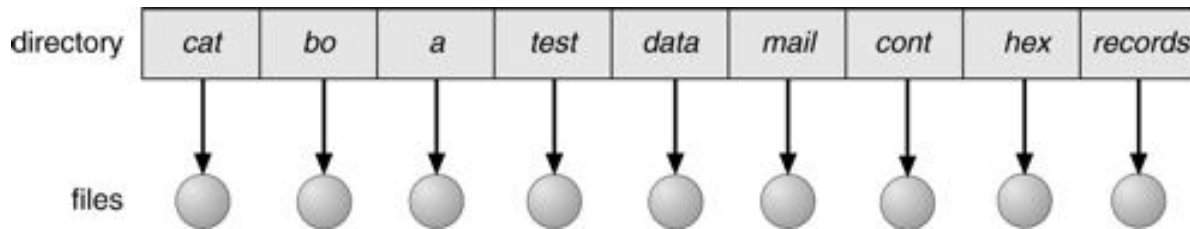


Figure 2.3: Single-level directory

Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Figure 2.3 contains a single-level directory with nine files named: *cat*, *bo*, *a*, *test*, *data*, *mail*, *cont*, *hex* and *records*). Each of the square boxes represents a name of a file. The file names are unique to avoid conflict.

## 2. Two-Level Directory

A single level directory often leads to confusion of file names between different users. The standard solution is to create separate directory for each user. In the two-level directory structure, each user has her own user file directory (UFD). Each UFD has a similar structure, but lists only the files of a single user. When a user job starts or a user logs in, the system’s master File Directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD of that user (see Figure 4).

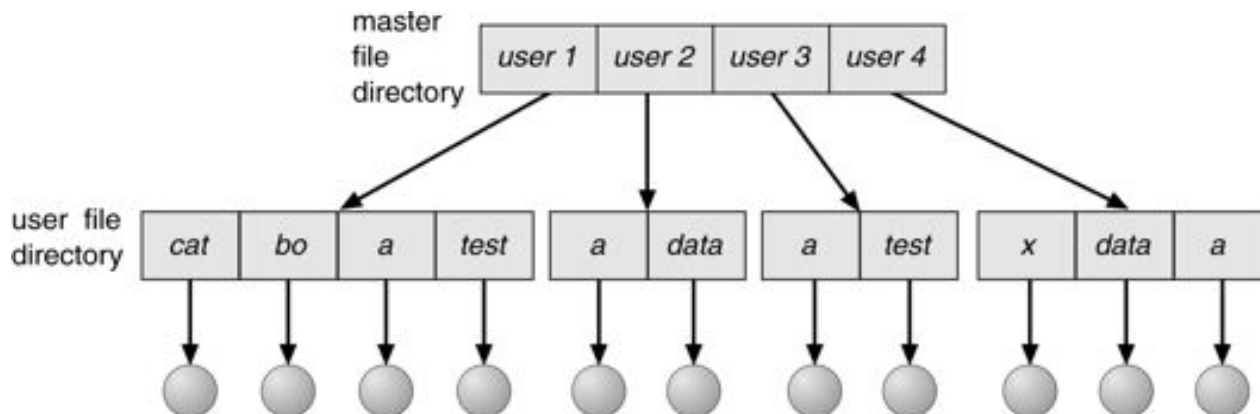


Figure 2.4. Two-level directory structure

Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

In Figure 2.4, the topmost level is the master file directory (MFD). It contains the directories for the number of users in the system. For instance, we could have a case of four directories named *user 1*, *user 2*, *user 3* and *user 4*. Immediately following the MFD is the user file directory (UFD). Selecting any of the four directories will display the files contained in the directory.

To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. Thus, a user name and a file name define a *path name*. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired. For example, if user 1 wishes to access her own test file named *test*, she can simply refer to *test*. To access the file named *test* of user 2 (with directory-entry name *user2*), however, she might have to refer to */user2/test*. Every system has its own syntax for naming files in directories other than the user's own.

There are still problems with two-level directory structure. This structure effectively isolates one user from another. This is an advantage when the users are completely independent, but a disadvantage when the users want to cooperate on some task and access files of other users. Some systems simply do not allow local files to be accessed by other users. This method is the one most used in UNIX and MS-DOS.

### **3. Tree – Structured Directory**

A more powerful and flexible approach, and one that is almost universally adopted, is the hierarchical, or tree structured directory (see Figure 5). As before, there is a master directory, which has under it a number of user directories. Each of these user directories in turn may have subdirectories and files as entries. This is true at any level. That is, at any level, a directory may consist of entries for subdirectories and/or entries for files.

Thus, to delete a directory, someone must first delete all the files in that directory. If these are any sub-directories, this procedure must be applied recursively to them, so that they can be deleted also. An alternative approach is just to assume that, when a request is made to delete a directory, all of that directory's files and sub-directories are also to be deleted. The Microsoft Windows family of operating systems (95, 98, NT, 2000, XP) maintains a tree-structured directory.

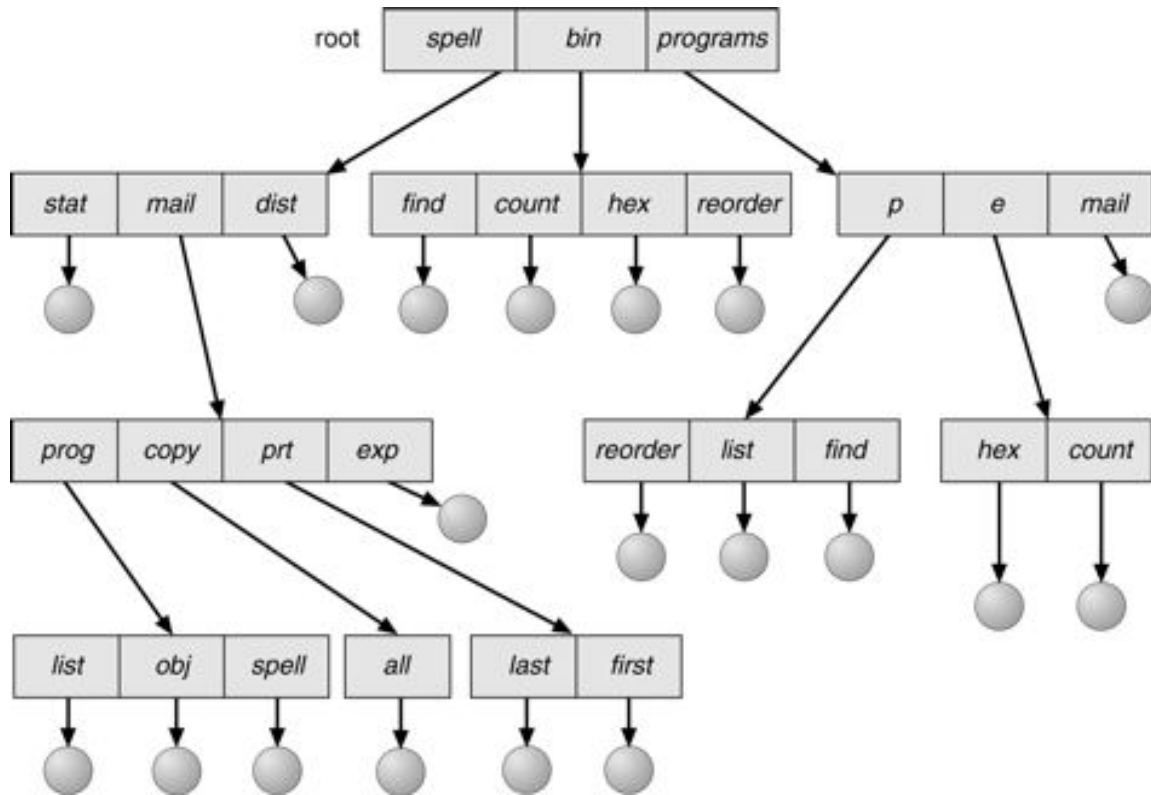


Figure 2.5: Tree-structured directories

Source: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Figure 5 presents a tree directory structure. The highest level of the directory is the root directory containing directory name *spell*, *bin* and *programs*. Subdirectories and files are created and can be accessed through the root level recursively. For instance, when *spell* directory is selected, it shows the content of the subdirectories *stat*, *mail* and *dist*. Each of the shaded circle represent files. In this case, *stat* subdirectory has a file and no subdirectory. *Mail* is a subdirectory with four subdirectories: *prog*, *copy*, *prt* and *exp*. Access to directories and files is recursively done in a tree form until there are no more directories and files to view.

### Activity A

Write short notes on the following access methods and state one advantage in each case: Sequential, Direct and Index sequential.

## 4.0 Conclusion

A file is named, for the convenience of its human users, and is referred to by its name. The following are five basic operations of the operating systems on files: Creating a file, Writing a file, Reading a file, Deleting a file and Truncating a file. Some systems provide only one access method for files. Other systems, support many access methods. Various different disk scheduling policies are used: FCFS, SSTF, SCAN, C-SLAN, LOOK and C-LOOK.

## 5.0 Summary

In this unit, we have learnt:

- (a) the attributes of files to include: identifier, type, location, size, protection, time, date and user identification.
- (b) the operations of files involves: creating, writing, reading, deleting, renaming, appending and truncating.
- (c) the access method of files such as: sequential, direct and index sequential file.
- (d) the disk allocation and scheduling methods: FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK.
- (e) the directory structure of files with: single, two and three levels.

## 6.0 Tutor Marked Assignment

- (a) With the aid of a diagram, explain the different levels of directories.
- (b) State five basic operations that can be performed on a:
  - i. File
  - ii. Directory
- (c) Briefly describe the different disk scheduling policies

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Ayo C. K. (2005), *Computer Literacy*, First Ed. Alanukitan Press.

Bovet D. and Cesati M. *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly, 2003.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.

Topics in Computer Science, *The university of Edinburgh*, 2009,  
<http://www.dcs.ed.ac.uk/home/stg/pub/D/disk.html>

CSc33200: Operating Systems, CS-CCNY, Fall 2003 Jinzhong Niu December 8, 2003



## MODULE 5: DEVICE AND FILE MANAGEMENT

### UNIT 3: PROTECTION

	Page
1.0	Introduction
2.0	Objectives
3.0	File protection and security
3.1	Hardware protection
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	Further Reading and Other Resources

#### 1.0 Introduction

The introduction of multiprogramming brought about the ability to share resources among users. This sharing involves not just the processor but also the following: files, memory, I/O devices, such as disks and printers, programs and data. The ability to share these resources brought the need for protection. This unit examines the objectives of protection and security of these resources.

#### 2.0 Objectives

At the end of this unit, the reader should be able to:

- (a) Know file protection and security.
- (b) Distinguish among I/O protection, memory protection and CPU protection.

#### 3.0 File Protection and Security

When information is kept in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally, bugs in the file-system software can also cause file contents to be lost.

Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multi-user system, however, other mechanisms are needed.

## Types of Access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.

Protection mechanisms provide controlled access by limited the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- Read: Read from a file.
- Write: Write or rewrite the file.
- Execute: Load the file into memory and execute it.
- Append: Write new information at the end of the file.
- Delete: Delete the file and free its space for possible reuse.
- List: List the name and attributes of the file.

Other operations, such as renaming, copying, or editing the file, may also be controlled.

### 3.1 Hardware protection

A properly designed OS must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

Many programming errors are detected by the hardware. These errors are normally handled by the operating system. If a user program fails in some way, such as by making an attempt either to execute an illegal instruction, or to access memory that is not in the user's address space, then the hardware will trap to the operating system, just like an interrupt.

Whenever a program error occurs, the operating system must normally terminate the program.

The following will be discussed: I/O Protection, Memory Protection and CPU Protection.

#### I/O Protection

A user program may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. We can use various mechanisms to ensure that such disruptions cannot take place in the system. To prevent users from performing illegal I/O, we define all I/O instructions to privileged instructions. Thus, users cannot issue I/O instructions directly, they must do it through the operating system.

#### Memory Protection

To ensure correct operation, we must protect the interrupt vector from Modification by a user program. In addition, we must also protect the interrupt-service routines in the operating system from modification. Even if the user did not gain unauthorized control of the computer, modifying the interrupt service routines would probably disrupt the proper operation of the computer system.

We see that we must provide memory protection at least for the interrupt vector and the

interrupt-service routines of the operating system. In general, we want to protect the operating system from access by user programs, and in addition, to protect user programs from one another. This protection must be provided by the hardware.

### **CPU Protection**

In addition to protecting I/O and memory, we must ensure that the operating system maintains control. We must prevent a user program from getting stuck in an infinite loop or not calling system services, and never returning control to the operating system. To accomplish this goal, we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed or variable.

Before turning over control to the user, the OS ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the OS, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the operation of the timer are privileged.

One of the most published threats to security is the intruder (the other is viruses), generally referred to as a hacker or cracker. The front of defense against intruders is the password system.

### **Activity A**

Write short notes on the different types of protection mechanism.

### **4.0 Conclusion**

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. To prevent users from performing illegal I/O, we define all I/O instructions to privileged instructions. Thus, users cannot issue I/O instructions directly, they must do it through the operating system. To ensure correct operation, we must protect the interrupt vector from modification by a user program. In addition to protecting I/O and memory, we must ensure that the operating system maintains control of the system resources. We must prevent a user program from getting stuck in an infinite loop or not calling system services, and never returning control to the operating system

### **5.0 Summary**

In this unit, we have learnt:

- (a) about file access prohibition methods for the purpose of file security.
- (b) that I/O protection mechanism can be used to prevent users from performing illegal I/O.
- (c) the protection of memory, CPU and OS from unauthorized access by user's programs.

### **6.0 Tutor Marked Assignment**

Describe protection mechanism for any two (2) of the following: memory, I/O device, Programs, Data, file.

## 7.0 Further Reading and Other Resources

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne (2005). *Operating Systems Concepts*, 7th Edition, John Wiley & Sons Inc; ISBN 0471417432

Cass S. “Anatomy of malice.” IEEE Spectrum, November 2001.

C. M. Krishna & Kang G. Shin (2006), *Real-Time Systems*, McGraw-Hill International editions.

Jean Bacon & Tim Harris (2003), *Operating Systems, Concurrent and Distributed Software Design*, Pearson Education Publishers.

William Stallings (2005), *Operating Systems, Internals & Design principles*, fifth edition, Pearson Hall.

Gary Nutt (2003), *Operating Systems*, third edition. Addison Wesley.

Kai Hwang and Fye A. Briggs (2006), *Computer Architecture & Parallel Processing*, McGraw-Hill, Book Company.