NATIONAL OPEN UNIVERSITY OF NIGERIA

SCHOOL OF SCIENCE AND TECHNOLOGY

COURSE CODE: CIT 211

COURSE TITLE: INTRODUCTION TO OPERATING SYSTEMS

COURSE GUIDE

CIT 211

INTRODUCTION TO OPERATING SYSTEMS

Course Developer	Afolorunso, A. A. National Open University of Nigeria Lagos.
Course Co-ordinator	Afolorunso, A. A. National Open University of Nigeria Lagos.
Course Editor	
Programme Leader	Prof. Afolabi Adebanjo

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Annex
245 Samuel Adesujo Ademulegun Street
Central Business District
Opposite Arewa Suites
Abuja

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

First Printed 2010

ISBN

Printed by ……………..
For
National Open University of Nigeria

TABLE OF CONTENTS **PAGE**

Introduction

CIT 211 – Introduction to Operating Systems is a three (3) credit unit course of twenty-two units. It deals with the various operating system design concepts and techniques.

It also gives an insight into evolution of operating systems. Since operating systems is the most important software in the computer system, this course takes you through the various types of operating system depending on the environment, the design considerations, the functions performed by the operating system and how these are achieved/implemented either through hardware or software..

This course is divided into six modules. The first module deals with the basic introduction to the concept of Operating Systems such as definition and functions of operating system, history and evolution of operating system.

The second module treats, extensively, the various types of operating system.

The third module deals with concept of process management and discusses the concepts of Co-operating Processes, Threads, and CPU Scheduling.

The fourth module discusses process synchronization issues such as Race Condition, Mutual Exclusion, Critical Section Problem, and other Classic Problems of Synchronization.

The fifth module treats deadlock issues such as deadlock Characterization and methods for dealing with deadlocks.

The last i.e. the sixth module discusses memory management functions of the operating system and issues such as memory management algorithms like paging, segmentation, contiguous memory allocation with their peculiar features were discussed.

This Course Guide gives you a brief overview of the course contents, course duration, and course materials.

What you will learn in this course

The main purpose of this course is to provide the necessary tools for designing and Operating system. It makes available the steps and tools that will enable you to make proper and accurate decision on designs issues and the necessary algorithms for a particular computing environment. Thus, we intend to achieve this through the following:

Course Aims

I. Introduce the concepts associated with Operating systems;

II. Provide necessary tools for analysing a computing environment and choosing/designing appropriate operating system.

III. Provide you with the necessary foundation in operating system designs

Course Objectives

Certain objectives have been set out to ensure that the course achieves its aims. Apart from the course objectives, every unit of this course has set objectives. In the course of the study, you will need to confirm, at the end of each unit, if you have met the objectives set at the beginning of each unit. By the end of this course you should be able to:

1. Define an OS
2. State the major functions of the OS
3. State the importance of the OS in the computer system./
4. Enumerate the various services rendered the user by the OS
5. Discuss the history and evolution of operating system
6. Describe the functions/responsibilities of the kernel
7. Distinguish a process from a program
8. List the possible states of a process
9. Describe a process control block (PCB)
10. Describe process creation and process termination
11. *Describe the concept of co-operating processes*
12. *State reasons for allowing process co-operation*
13. *Explain interprocess communication*
14. Distinguish between a thread and a process
15. Enumerate the advantages of threads over processes
16. Distinguish between preemptive and non-preemptive scheduling
17. State the goals for CPU scheduling
18. Give comparative analysis of the various CPU scheduling algorithms
19. Describe the various CPU scheduling evaluation algorithms.
20. Define Race condition
21. Describe some real life examples of race condition
22. Define process synchronization
23. Describe ways to enforce mutual exclusion
24. Explain the critical section problem
25. Define deadlock
26. State the necessary conditions for deadlock to occur
27. Describe some of the methods for handling deadlocks.
28. Describe address binding
29. Define logical and physical address space
30. Distinguish between dynamic loading and dynamic linking
31. Describe contiguous memory allocation and its various variants
32. Distinguish between internal and external fragmentation
33. Describe methods of solving external fragmentation
34. Describe the principle of overlays and its uses

35. Explain the differences between paging and segmentation
36. Describe a method for solving the problems of both paging and segmentation

Working Through This Course

In order to have a thorough understanding of the course units, you will need to read and understand the contents, practise the steps by designing an Information system of your own, and be committed to learning and implementing your knowledge.

This course is designed to cover approximately seventeen weeks, and it will require your devoted attention. You should do the exercises in the Tutor-Marked Assignments and submit to your tutors.

Course Materials

These include:

1. Course Guide
2. Study Units
3. Recommended Texts
4. A file for your assignments and for records to monitor your progress.

Study Units

There are twenty-two study units in this course:

Module 1: Operating System Fundamentals

Unit 1: Definition and Functions of Operating System (OS)
Unit 2: History and Evolution of Operating System
Unit 3: The Kernel
Unit 4: Types of Operating Systems

Module 2: Types of Operating System

Unit 1: Disk operating system

Unit 2: Real-time operating system

Unit 3: Time-Sharing and Object-oriented operating system

Module 3: Process Management

Unit 1: Processes

Unit 2: Co-operating Processes

Unit 3: Threads

Unit 4: CPU Scheduling

Unit 5: Algorithm Evaluation

Module 4: Process Synchronization
Unit 1: Race Condition
Unit 2: Synchronization
Unit 3: Mutual Exclusion
Unit 4: Critical Section Problem
Unit 5: Classic Problems of Synchronization

Module 5: Deadlocks
Unit 1: Deadlock Characterization
Unit 2: Methods for Dealing with Deadlocks

Module 6: Memory Management
Unit 1: Memory Management Fundamentals
Unit 2: Memory Management Algorithms I
Unit 3: Memory Management Algorithms II

Make use of the course materials, do the exercises to enhance your learning.

Textbooks and References

1. Roch, Benjamin (2004). Monolithic kernel vs. Microkenrel (pdf). Retrieved on 2006-10-12.
2. Silberschatz, Abraham; James L. Peterson, Peter B. Galvin (1991). Operating system concepts. Boston, Massachusetts: Addison-Wesley, 696. ISBN 0-201-51379-X.
3. Hansen, Per Brinch (April 1970). "The nucleus of a Multiprogramming System". Communications of the ACM **13** (4): 238-241. ISSN 0001-0782.
4. Deitel, Harvey M. [1982] (1984). An introduction to operating systems, revisited first edition, Addison-Wesley, 673. ISBN 0-201-14502-2.
5. Denning, Peter J. (April 1980). "Why not innovations in computer architecture?". ACM SIGARCH Computer Architecture News **8** (2): 4-7. ISSN 0163-5964.
6. Hansen, Per Brinch [1973]. Operating System Principles. Englewood Cliffs: Prentice Hall, 496. ISBN 0-13-637843-9.
7. Per Brinch Hansen (2001). "The evolution of operating systems" (pdf). Retrieved on 2006-10-24. included in book: [2001] "1", in Per Brinch Hansen: Classic operating systems: from batch processing to distributed systems. New York,: Springer-Verlag, 1-36. ISBN 0-387-95113-X.
8. Levin, R.; E. Cohen, W. Corwin, F. Pollack, W. Wulf (1975). "Policy/mechanism separation in Hydra". ACM Symposium on Operating Systems Principles / Proceedings of the fifth ACM symposium on Operating systems principles: 132-140.
9. Linden, Theodore A. (December 1976). "Operating System Structures to Support Security and Reliable Software". ACM Computing Surveys (CSUR) **8** (4): 409 - 445. ISSN 0360-0300.
10. Lorin, Harold (1981). Operating systems. Boston, Massachusetts: Addison-Wesley, pp.161-186. ISBN 0-201-14464-6.

11. Schroeder, Michael D.; Jerome H. Saltzer (March 1972). "A hardware architecture for implementing protection rings". Communications of the ACM **15** (3): 157 - 170. ISSN 0001-0782.
12. Shaw, Alan C. (1974). The logical design of Operating systems. Prentice-Hall, 304. ISBN 0-13-540112-7.
13. Tanenbaum, Andrew S. (1979). Structured Computer Organization (in English). Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-148521-0.
14. Wulf, W.; E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack (June 1974). "HYDRA: the kernel of a multiprocessor operating system". Communications of the ACM **17** (6): 337 - 345. ISSN 0001-0782.
15. Andrew Tanenbaum, Operating Systems - Design and Implementation (Third edition);
16. Andrew Tanenbaum, Modern Operating Systems (Second edition);
17. Baiardi, F.; A. Tomasi, M. Vanneschi (1988). Architettura dei Sistemi di Elaborazione, volume 1 (in Italian). Franco Angeli. ISBN 88-204-2746-X.
18. Daniel P. Bovet, Marco Cesati, The Linux Kernel;
19. David A. Peterson, Nitin Indurkhya, Patterson, Computer Organization and Design, Morgan Koffman (ISBN 1-55860-428-6);
20. B.S. Chalk, Computer Organisation and Architecture, Macmillan P.(ISBN 0-333-64551-0).
21. Stallings, William (2005). Operating Systems: internals and design principles (5th edition). Prentice Hall. ISBN 0-13-127837-1.
22. Allen B. Downey The Little Book of Semaphores, http://greenteapress.com/semaphores
23. Dijkstra, E. W. (1971, June). Hierarchical ordering of sequential processes. Acta Informatica 1(2): 115-138.
24. Chandy, K.M.; Misra, J. (1984). The Drinking Philosophers Problem. ACM Transactions on Programming Languages and Systems.
25. Lehmann, D. J., Rabin M. O, (1981). On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem. Principles Of Programming Languages 1981 (POPL'81), pages 133-138.

Assignments File

These are of two types: the self-assessment exercises and the Tutor-Marked Assignments. The self-assessment exercises will enable you monitor your performance by yourself, while the Tutor-Marked Assignment is a supervised assignment. The assignments take a certain percentage of your total score in this course. The Tutor-Marked Assignments will be assessed by your tutor within a specified period. The examination at the end of this course will aim at determining the level of mastery of the subject matter. This course includes twelve Tutor-Marked Assignments and each must be done and submitted accordingly. Your best scores however, will be recorded for you. Be sure to send these assignments to your tutor before the deadline to avoid loss of marks.

Presentation Schedule

The Presentation Schedule included in your course materials gives you the important dates for the completion of tutor marked assignments and attending tutorials. Remember, you are required to submit all your assignments by the due date. You should guard against lagging behind in your work.

Assessment

There are two aspects to the assessment of the course. First are the tutor marked assignments; second, is a written examination.

In tackling the assignments, you are expected to apply information and knowledge acquired during this course. The assignments must be submitted to your tutor for formal assessment in accordance with the deadlines stated in the Assignment File. The work you submit to your tutor for assessment will count for 30% of your total course mark.

At the end of the course, you will need to sit for a final three-hour examination. This will also count for 70% of your total course mark.

Tutor Marked Assignments (TMAS)

There are twenty-two tutor marked assignments in this course. You need to submit all the assignments. The total marks for the best three (3) assignments will be 30% of your total course mark.

Assignment questions for the units in this course are contained in the Assignment File. You should be able to complete your assignments from the information and materials contained in your set textbooks, reading and study units. However, you may wish to use other references to broaden your viewpoint and provide a deeper understanding of the subject.

When you have completed each assignment, send it together with form to your tutor. Make sure that each assignment reaches your tutor on or before the deadline given. If, however, you cannot complete your work on time, contact your tutor before the assignment is done to discuss the possibility of an extension.

Examination and Grading

The final examination for the course will carry 70% percentage of the total marks available for this course. The examination will cover every aspect of the course, so you are advised to revise all your corrected assignments before the examination.

This course endows you with the status of a teacher and that of a learner. This means that you teach yourself and that you learn, as your learning capabilities would allow. It also means that you are in a better position to determine and to ascertain the what, the how, and the when of your language learning. No teacher imposes any method of learning on you.

The course units are similarly designed with the introduction following the table of contents, then a set of objectives and then the dialogue and so on.

The objectives guide you as you go through the units to ascertain your knowledge of the required terms and expressions.

Course Marking Scheme

This table shows how the actual course marking is broken down.

Assessment	**Marks**
Assignment 1- 4	Four assignments, best three marks of the four count at 30% of course marks
Final Examination	70% of overall course marks
Total	100% of course marks

Table 1: Course Marking Scheme

Course Overview

Unit	**Title of Work**	**Weeks Activity**	**Assessment (End of Unit)**
	Course Guide	Week 1	
	Module 1: Operating System Fundamentals		
1	Definition and Functions of Operating System (OS)	Week 1	Assignment 1
2	History and Evolution of Operating System	Week 1	Assignment 2
3	The Kernel	Week 2	Assignment 3
4	Types of Operating Systems	Week 2	Assignment 4
	Module 2: Types of Operating System		
1	Disk operating system	Week 3	Assignment 5
2	Real-time operating system	Week 3	Assignment 6

3	Time-Sharing and Object-oriented operating system	Week 3	Assignment 7
	Module 3: Process Management		
1	Processes	Week 4	Assignment 8
2	Co-operating Processes	Week 4	Assignment 9
3	Threads	Week 5	Assignment 10
4	CPU Scheduling	Week 6-7	Assignment 11
5	*Algorithm Evaluation*	Week 7	Assignment 12
	Module 4: Process Synchronization		
1	Race Condition	Week 8	Assignment 13
2	*Synchronization*	Week 8	Assignment 14
3	*Mutual Exclusion*	Week 9	Assignment 15
4	Critical Section Problem	Week 9	Assignment 16
5	*Classic Problems of Synchronization*	Week 10	Assignment 17
	Module 5: Deadlocks		
1	Deadlock Characterization	Week 11	Assignment 18
2	*Methods for Dealing with Deadlocks*	Week 12	Assignment 19
	Module 6: Memory Management		
1	Memory Management Fundamentals	Week 13	Assignment 20
2	Memory Management Algorithms I	Week 14	Assignment 21
3	*Memory Management Algorithms II*	Week 15	Assignment 22
	Revision	Week 16	
	Examination	Week 17	
Total		17 weeks	

How to get the best from this course

In distance learning the study units replace the university lecturer. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. In the same way that a lecturer might set you some reading to do, the study units tell you when to read your set books or other material. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other units and the course as a whole. Next is a set of learning objectives. These objectives enable you know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units you must go back and check whether you have achieved the objectives. If you make a habit of doing this you will significantly improve your chances of passing the course.

Remember that your tutor's job is to assist you. When you need help, don't hesitate to call and ask your tutor to provide it.

1. Read this Course Guide thoroughly.

2. Organize a study schedule. Refer to the 'Course Overview' for more details. Note the time you are expected to spend on each unit and how the assignments relate to the units. Whatever method you chose to use, you should decide on it and write in your own dates for working on each unit.

3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they lag behind in their course work.

4. Turn to Unit 1 and read the introduction and the objectives for the unit.

5. Assemble the study materials. Information about what you need for a unit is given in the 'Overview' at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time.

6. Work through the unit. The content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit you will be instructed to read sections from your set books or other articles. Use the unit to guide your reading.

7. Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.

8. When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.

9. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have any questions or problems.

10. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this Course Guide).

Tutors and Tutorials

There are 12 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials, together with the name and phone number of your tutor, as soon as you are allocated a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail or submit your tutor-marked assignments to your tutor well before the due 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 hesitate to contact your tutor by telephone, or e-mail if you need help. The following might be circumstances in which you would find help necessary. Contact your tutor if:

- you do not understand any part of the study units or the assigned readings,
- you have difficulty with the self-tests or exercises,
- you have a question or problem with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should try your best to attend the tutorials. This is the only chance to have face to face contact with your tutor and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussions actively.

Summary

Introduction to Operating systems introduces you to the concepts associated with operating system development and functions which is critical in understanding the various computing environments/hardware platforms. The content of the course material was planned and written to ensure that you acquire the proper knowledge and skills for the appropriate situations. Real-life situations have been created to enable you identify with and create some of your own. The essence is to help you in acquiring the necessary knowledge and competence by equipping you with the necessary tools to accomplish this.
We hope that by the end of this course you would have acquired the required knowledge to view operating systems and the computing environments in a new way.

We wish you success with the course and hope that you will find it both interesting and useful.

CIT 211

INTRODUCTION TO OPERATING SYSTEMS

Course Developer	Afolorunso, A. A. National Open University of Nigeria Lagos.
Course Co-ordinator	Afolorunso, A. A. National Open University of Nigeria Lagos.
Course Editor	Dr. A. S. Sodiya University of Agriculture, Abeokuta Ogun State
Programme Leader	Prof. Afolabi Adebanjo

Module 1: Operating System Fundamentals

- Unit 1: Definition and Functions of Operating System (OS)
- Unit 2: History and Evolution of Operating System
- Unit 3: The Kernel
- Unit 4: Types of Operating Systems

Module 2: Types of Operating System

- Unit 1: Disk operating system
- Unit 2: Real-time operating system
- Unit 3: Time-Sharing and Object-oriented operating system

Module 3: Process Management

- Unit 1: Processes
- Unit 2: Co-operating Processes
- Unit 3: Threads
- Unit 4: CPU Scheduling
- Unit 5: Algorithm Evaluation

Module 4: Process Synchronization

- Unit 1: Race Condition
- Unit 2: Synchronization
- Unit 3: Mutual Exclusion
- Unit 4: Critical Section Problem

Module 5: Deadlocks

- Unit 1: Deadlock Characterization
- Unit 2: Methods for Dealing with Deadlocks

Module 6: Memory Management

- Unit 1: Memory Management Fundamentals
- Unit 2: Memory Allocation Techniques
- Unit 3: Non-Contiguous Memory Allocation

Module 1: Operating System Fundamentals
Unit 1: Definition and Functions of Operating System (OS)

Table of Contents

1.0 Introduction

Having just read through the Course Guide, you are now to go through this first unit of the course which is very fundamental to the understanding of what an Operating system is and the role it plays in the whole computer system.

Now let us go through your study objectives for this unit.

2.0 Objectives

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

- Define an OS
- State the major functions of the OS
- State the importance of the OS in the computer system./
- Enumerate the various services rendered the user by the OS

4.0 Main Body

3.1 What is an Operating System?

An **Operating System** (**OS**) can be defined as a set of computer programs that manage the hardware and software resources of a computer. It is the core of computer programming that primarily deals with computer architecture. Operating system is basically an application program that serves as an interface to coordinate different resources of computer. An operating system processes raw system and user input and responds by allocating and managing tasks and internal system resources as a service to users and programs of the system.

But simply put, an (OS) can be defined as a suite (set) of programs implemented either in software or firmware (hardwired instructions on chips usually in ROM) or both that makes the hardware usable.

At the foundation of all system software, an operating system performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems. Most operating systems come with an application that provides an interface to the OS managed resources. These applications have had command line interpreters as a basic user interface, but more recently have been implemented as a graphical user interface (GUI) for ease of operation. Operating Systems themselves, have no user interfaces, and the user of an OS is an application, not a person. The operating system forms a platform for other system software and for application software. Windows, Linux, and Mac OS are some of the most popular OS's.

3.2 Goals and Functions of OS

OS can further be described by what they do i.e. by their functions, goals and objectives. Therefore, we will quickly run you through some of the goals of the OS which are:

3.2.1 Convenience for the User

When there is no OS, users of computer system will need to write machine-level program in order to manipulate the hardware. With OS, users can now easily and conveniently use the computer with no stress of directly programming the hardware. OS provide a convenient interface for using the computer system.

3.2.2 Efficiency

An OS allows computer system resources to be used in an efficient manner. This particularly important for large shared multi-user systems which are usually expensive. In the past, the efficiency (i.e. optimal use of the computer resources) considerations were often more important than convenience.

3.2.3 Evolutionary Capabilities

Ability to evolve also happens to be one of the goals of the OS. An OS should be constructed in such a way that it permits the effective development, testing and introduction of new system functions without interfering with its service.

3.3 Views of OS

OS can be viewed from the perspective of what they are. These views are diverse depending on the particular view point of a user. But some of these views are discussed below.

3.3.1 OS as a User/Computer Interface

A computer system can be viewed as a layered or hierarchical structure consisting of the hardware, operating system, utilities, application programs and users.

The users of application programs are called the end-users and are generally not concerned with the computer's architecture. The end-user views the computer system in terms of an application.

The application is developed by the application programmer who uses a programming language and a language translator. A set of programs called the utilities is provided to assist the programmer in program creation, file management and the control of Input/Output (I/O) devices.

The most important system program, operating system masks the details of the hardware from the programmer and provides a convenient interface for using the system. it acts as a mediator, making it easier for the programmer and for application programs to access and use the available services and facilities.

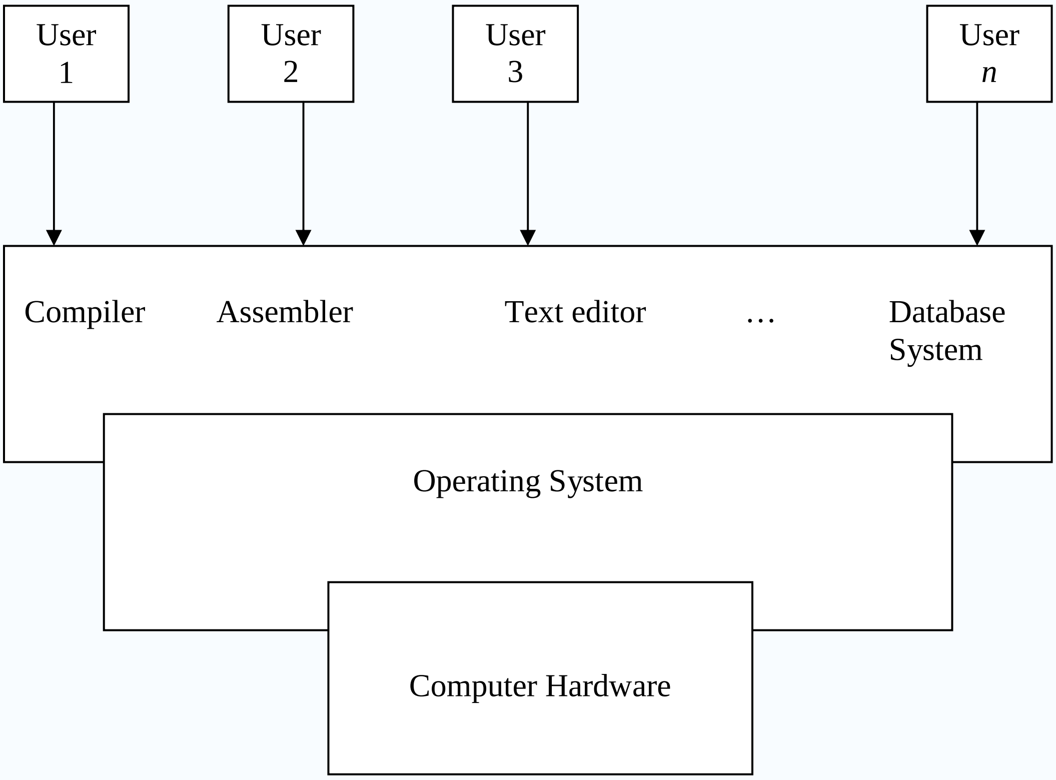


Figure 3.1: Abstract View of the Components of a Computer System

3.3.2 OS as a Resource Manager

A computer system has a set of resources for the movement, storage and processing of data. The OS is responsible for managing these resources. Note that resources include CPU, file storage space, data, programs, memory space, I/O devices, etc.

The OS is like any other computer program in that it provides instructions for the processor. The key difference is in the purpose of the program. The OS directs the processor in the use of the other system resources and in the timing of its execution of other programs. The processor, in order to do any of these things, must cease execution of the OS program to execute other programs. Thus, the OS relinquishes control long enough to prepare the processor to do the next piece of work.

The portion of the OS that is always in main memory is called the kernel or nucleus and it contains the most frequently used functions in the OS. The remainder of the main memory contains other user programs and data. The allocation of this resource (i.e. main memory) is controlled jointly by the OS and the memory management hardware in the processor.

3.3.2 Services Provided by the OS

The services provided by the OS can be categorised into two:

3.3.2.1 Convenience for the Programmer/User

The conveniences offered the user are in diverse and following ways:

i. Program Creation: Although editors and debuggers are not part of the OS, they are accessed through the OS to assist programmers in creating programs.
ii. Program Execution: OS ensures that programs are loaded into the main memory. I/O devices and files are initialised and other resources are prepared. The program must be able to end its execution either normally or abnormally. In case of abnormal end to a program, it must indicate error.
iii. Access to I/O devices: Each I/O device requires its own set of instructions or control signal for operation. The OS takes care of the details so that the programmer can think in terms of reads and writes.
iv. Controlled Access: In the case of files, control includes an understanding of the nature of the I/O device (e.g. diskette drive, CDROM drive, etc.) as well as the file format of the storage medium. The OS deals with these details. In the case of the multi-user system, the OS must provide protection mechanisms to control access to the files.
v. Communications: There are many instances in which a process needs to exchange information with another process. There are two major ways in which communication can occur:

    - It can take place between processes executing on the same computer.
    - It can take place between processes executing on different computer systems that are linked by a computer network.

Communications may be implemented via a shared memory or by a technique of message passing in which packets of information are moved between processes by the OS.

vi. Error Detection: A variety of errors can occur while a computer system is running. These errors include:
   - CPU and memory hardware error: This encompasses memory error, power failure, a device failure such as connection failure on a network, lack of paper in printer.
   - Software errors: Arithmetic overflow, attempt to access forbidden memory locations, inability of the OS to grant the request of an application.

   In each case, the OS must make a response that makes the less impact on running applications. The response may range from ending the program that caused the error, retrying the operation or simply reporting the error to the application.

3.3.2.2 Efficiency of System: Single and Multi-User

In the area of system efficiency, the OS offer the following services:

i. System Access or Protection: In the case of a shared or public system, the OS controls access to the system and to specific system resources by ensuring that each user authenticates him/herself to the system, usually by means of passwords to be allowed access to system resources. It extends to defending external I/O devices including modems, network adapters from invalid access attempts and to recording all such connections for detection of break-ins.

ii. Resources Allocation: In an environment where there multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the OS. Some (such as CPU cycles, main memory and file storage) may have general request and release codes. For instances, in determining how best to use the CPU, the OS have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available and other factors. These routines may also be used to allocate plotters, modems and other peripheral devices.

iii. Accounting: This helps to keep track of how much of and what types of computer resources are used by each user. Today, this record keeping is not for billing purposes but for simply accumulating usage statistics. This statistics may be available tool for researchers who want to reconfigure the system to improve computing services.

iv. Ease of Evolution of OS: A major OS will evolve over time for a number of reasons such as hardware upgrades and new types of hardware e.g. The use of graphics terminals may affect OS design. This is because such a terminal may allow the user to view several applications at the same time through ‘windows’ on the screen. This requires more sophisticated support in the OS.

v. New Services: In response to user demands or the need of system managers, the OS may expand to offer new services.
vi. Error corretion: The OS may have faults which may be discovered over the course of time and fixes will need to be made.

Other features provided by the OS includes:

- Defining the user interface
- Sharing hardware among users
- Allowing users to share data
- Scheduling resources among users
- Facilitating I/O
- Recovering from errors
- Etc.

The OS interfaces with, programs, hardware, users such as administrative personnel, computer operators, application programmers, system programmers, etc.

4.0 Conclusion

As you have learnt in this unit the OS is very important software in the computer system that provides a variety of services to the applications running on the system and the user. It also adds to the efficiency and performance of the computer system.

5.0 Summary

The OS forms the bedrock of the computer system and is the platform on which all other software run. But the OS has not always been nor come with the computer system. It evolved over time as you are going to learn in the next unit.

6.0 Tutor Marked Assignment

You are to attempt the following assignments and submit your answers to your tutor for this course. Here we go:

1. What do you understand by the term ‘Operating System’?
2. List and briefly explain the various services rendered to the users by the OS
3. Enumerate the goals and functions of the OS

7.0 References/Further Reading

1) T. Y. James (1999). *Introduction to Operating Systems*. 2nd Edition
2) Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5

Module 1: Operating System Fundamentals

Unit 2: History and Evolution of Operating System

Table of Contents

1.0 Introduction

Operating system was absent in the first commercial form of electronic computer launched in 1940's. Rows of mechanical switches were used to enter programs. At that time, programming languages were not in use. Naturally, there was hardly any idea about operating system. The user had sole use of the machine and would arrive armed with program and data, often on punched paper tape. The program would be loaded into the machine, and the machine would be set to work until the program completed or crashed. Programs could generally be debugged via a front panel using switches and lights. It is said that Alan Turing was a master of this on the early Manchester Mark I machine, and he was already deriving the primitive conception of an *operating system* from the principles of the Universal Turing Machine.

Later machines came with libraries of support code, which would be linked to the users' program to assist in operations such as input and output. This was the genesis of the modern-day operating system. However, machines still ran a single job at a time; at Cambridge University in England the job queue was at one time a washing line from which tapes were hung with different coloured clothes-pegs to indicate job-priority.

As machines became more powerful, the time needed for a run of a program diminished and the time to hand off the equipment became very large by comparison. Accounting for and paying for machine usage moved on from checking the wall clock to automatic logging by the computer

2.0 Objectives

By the end of this unit, you should be able to:

- Discuss the history and evolution of operating system
- State the basic functions of the operating system
- Differentiate the various features of each generation of the operating system

3.0 Main Body

3.1 History of Operating Systems

To see what operating systems are and what operating systems do, let us consider how they have developed over the last 30 years. By tracing that evolution we can identify the common elements of operating systems, and see how, and why they developed as they are now.

Operating systems and computer architecture have a great deal of influence on each other. To facilitate the use of the hardware, operating systems were developed. As operating system were designed and used, it became obvious that changes in the design of the hardware could simplify the operating system. In this short historical review, not ice how the introduction of new hardware features is the natural solution to many operating system problems.

Operating systems have been evolving over the years. let us briefly look at this development. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. This mapping of operating systems generations to computer generations is admittedly crude, but it does provide some structure where there would otherwise be none.

Since the history of computer operating systems parallels that of computer hardware, it can be generally divided into five distinct time periods, called generations that are characterized by hardware component technology, software development, and mode of delivery of computer services.

3.1.1 The Zeroth Generation

The term zeroth generation is used to refer to the period when there was no OS. This is the period before the commercial production and sale of computer equipment. The period might be dated as extending from the mid-1800s, and Charles Babbage’s Analytical Engine, to the development of the first commercial computer in 1951. In particular, this period witnessed the emergence of the first electronics digital computers on the ABC, designed by John Atanasoff in 1940; the Mark I, built by Howard Aiken and a group of IBM engineers at Harvard in 1944; and the ENIAC, designed and constructed at the University of Pennsylvania by Wallace Eckert and John Mauchly. Perhaps the most significant of these early computers was the EDVAC, developed in 1944-46 by John von Neumann, Arthur Burks, and Herman Goldstine, since it was the first to fully implement the idea of the stored program and serial execution of instructions. The development of EDVAC set the stage for the evolution of commercial computing and operating system software. The hardware component technology of this period was electronic vacuum tubes.

The actual operation of these early computers took place without the benefit of an operating system. Early programs were written in machine language and each contained code for initiating operation of the computer itself.

The mode of operation was called "open-shop" and this meant that users signed up for computer time and when a user’s time arrived, the entire (in those days quite large) computer system was turned over to the user. The individual user (programmer) was responsible for all machine set up and operation, and subsequent clean-up and preparation for the next user. This system was clearly inefficient and depended on the varying competencies of the individual programmer as operators.

3.1.2 The First Generation, 1951-1956

The first generation marked the beginning of commercial computing.

Operation continued without the benefit of an operating system for a time. The mode was called "closed shop" and was characterized by the appearance of hired operators who would select the job to be run, initial program load the system, run the user’s program, and then select another job, and so forth. Programs began to be written in higher level, procedure-oriented languages, and thus the operator’s routine expanded. The operator now selected a job, ran the translation program to assemble or compile the source program, and combined the translated object program along with any existing library programs that the program might need for input to the linking program, loaded and ran the composite linked program, and then handled the next job in a similar fashion.

Application programs were run one at a time, and were translated with absolute computer addresses that bound them to be loaded and run from these pre-assigned storage addresses set by the translator, obtaining their data from specific physical I/O device. There was no provision for moving a program to different location in storage for any reason. Similarly, a program bound to specific devices could not be run at all if any of these devices were busy or broken.

The inefficiencies inherent in the above methods of operation led to the development of the mono-programmed operating system, which eliminated some of the human intervention in running job and provided programmers with a number of desirable functions. The OS consisted of a permanently resident kernel in main storage, and a job scheduler and a number of utility programs kept in secondary storage. User application programs were preceded by control or specification cards (in those day, computer program were submitted on data cards) which informed the OS of what system resources (software resources such as compilers and loaders; and hardware resources such as tape drives and printer) were needed to run a particular application. The systems were designed to be operated as batch processing system.

These systems continued to operate under the control of a human operator who initiated operation by mounting a magnetic tape that contained the operating system executable code onto a "boot device", and then pushing the IPL (initial program load) or "boot" button to initiate the bootstrap loading of the operating system. Once the system was loaded, the operator entered the date and time, and then initiated the operation of the job scheduler program which read and interpreted the control statements, secured the needed resources, executed the first user program, recorded timing and accounting information, and then went back to begin processing of another user program, and so on, as long as there were programs waiting in the input queue to be executed.

The first generation saw the evolution from hands-on operation to closed shop operation to the development of mono-programmed operating systems. At the same time, the development of programming languages was moving away from the basic machine languages; first to assembly language, and later to procedure-oriented languages, the most significant being the development of FORTRAN (Formula Translator) by John W. Backus in 1956. Several problems remained, however. The most obvious was the inefficient use of system resources, which was most evident when the CPU waited while the relatively slower, mechanical I/O devices were reading or writing program data. In addition, system protection was a problem because the operating system kernel was not protected from being overwritten by an erroneous application program. Moreover, other user programs in the queue were not protected from destruction by executing programs.

3.1.3 The Second Generation, 1956-1964

The second generation of computer hardware was most not ably characterized by transistors replacing vacuum tubes as the hardware component technology. In addition, some very important changes in hardware and software architectures occurred during this period. For the most part, computer systems remained card and tape-oriented systems. Significant use of random access devices, that is, disks, did not appear until towards the end of the second generation. Program processing was, for the most part, provided by large centralized computers operated under mono-programmed batch processing operating systems.

The most significant innovations addressed the problem of excessive central processor delay due to waiting for input/output operations. Recall that programs were executed by processing the machine instructions in a strictly sequential order. As a result, the CPU, with its high speed electronic component, was often forced to wait for completion of I/O operations which involved mechanical devices (card readers and tape drives) that were order of magnitude slower. This problem led to the introduction of the data channel, an integral, special-purpose computer with its own instruction set, registers, and control unit designed to process input/output operations separate and asynchronously from the operation of the computer’s main CPU near the end of the first generation, and its widespread adoption in the second generation.

The data channel allowed some I/O to be buffered. That is, a program’s input data could be read "ahead" from data cards or tape into a special block of memory called a buffer. Then, when the user’s program came to an input statement, the data could be transferred from the buffer locations at the faster main memory access speed rather than the slower I/O device speed. Similarly, a program’s output could be written another buffer and later moved from the buffer to the printer, tape, or card punch. What made this all work was the data channel’s ability to work asynchronously and concurrently with the main processor. Thus, the slower mechanical I/O could be happening concurrently with main program processing. This process was called I/O overlap.

The data channel was controlled by a channel program set up by the operating system I/O control routines and initiated by a special instruction executed by the CPU. Then, the channel independently processed data to or from the buffer. This provided communication from the CPU to the data channel to initiate an I/O operation. It remained for the channel to communicate to the CPU such events as data errors and the completion of a transmission. At first, this communication was handled by polling. The CPU stopped its work periodically and polled the channel to determine if there were any message.

Polling was obviously inefficient (imagine stopping your work periodically to go to the post office to see if an expected letter has arrived) and led to another significant innovation of the second generation - the interrupt. The data. channel was now able to interrupt the CPU with a message- usually "I/O complete." In fact, the interrupt idea was later extended from I/O to allow signalling of number of exceptional conditions such as arithmetic overflow, division by zero and time-run-out. Of course, interval clocks were added in conjunction with the latter, and thus operating system came to have a way of regaining control from an exceptionally long or indefinitely looping program.

These hardware developments led to enhancements of the operating system. I/O and data channel communication and control became functions of the operating system, both to relieve the application programmer from the difficult details of I/O programming and to protect the integrity of the system to provide improved service to users by segmenting jobs and running shorter jobs first (during "prime time") and relegating longer jobs to lower priority or night time runs. System libraries became more widely available and more comprehensive as new utilities and application software components were available to programmers.

In order to further mitigate the I/O wait problem, system were set up to spool the input batch from slower I/O devices such as the card reader to the much higher speed tape drive and similarly, the output from the higher speed tape to the slower printer. Initially, this was accomplished by means of one or more physically separate small satellite computers. In this scenario, the user submitted a job at a window, a batch of jobs was accumulated and spooled from cards to tape "off line," the tape was moved to the main computer, the jobs were run, and their output was collected on another tape that later was taken to a satellite computer for offline tape-to-printer output used then picked up their output at the submission windows.

Toward the end of this period, as random access devices became available, tape-oriented operating system began to be replaced by disk-oriented systems. With the more sophisticated disk hardware and the operating system supporting a greater portion of the programmer’s work, the computer system that users saw was more and more removed from the actual hardware - users saw a virtual machine.

The second generation was a period of intense operating system development. Also it was the period for sequential batch processing. But the sequential processing of one job at a time remained a significant limitation. Thus, there continued to be low CPU utilization for I/O bound jobs and low I/O device utilization for CPU bound jobs. This was a major concern, since computers were still very large (room-size) and expensive machines. Researchers began to experiment with multiprogramming and multiprocessing in their computing services called the time-sharing system. A noteworthy example is the compatible Time Sharing System (CTSS), developed at MIT during the early 1960s.

3.1.4 The Third Generation, 1964-1979

The third generation officially began in April 1964 with IBM’s announcement of its System/360 family of computers. Hardware technology began to use integrated circuits (ICs) which yielded significant advantages in both speed and economy.

Operating system development continued with the introduction and widespread adoption of multiprogramming. This marked first by the appearance of more sophisticated I/O buffering in the form of spooling operating systems, such as the HASP (Houston Automatic Spooling) system that accompanied the IBM OS/360 system. These systems worked by introducing two new systems programs, a system reader to move input jobs from cards to disk, and a system writer to move job output from disk to printer, tape, or cards. Operation of spooling system was, as before, transparent to the computer user who perceived input as coming directly from the cards and output going directly to the printer.

The idea of taking fuller advantage of the computer’s data channel I/O capabilities continued to develop. That is, designers recognized that I/O needed only to be initiated by a CPU instruction - the actual I/O data transmission could take place under control of separate and asynchronously operating channel program. Thus, by switching control of the CPU between the currently executing user program, the system reader program, and the system writer program, it was possible to keep the slower mechanical I/O device running and minimize the amount of time the CPU spent waiting for I/O completion. The net result was an increase in system throughput and resource utilization, to the benefit of both user and providers of computer services.

This concurrent operation of three programs (more properly, apparent concurrent operation, since systems had only one CPU, and could, therefore execute just one instruction at time) required that additional features and complexity be added to the operating system. First, the fact that the input queue was now on disk, a direct access device, freed the system scheduler from the first-come-first-served policy so that it could select the "best" next job to enter the system (looking for either the shortest job or the highest priority job in the queue). Second, since the CPU was to be shared by the user program, the system reader, and the system writer, some processor allocation rule or policy was needed. Since the goal of spooling was increase resource utilization by enabling the slower I/O devices to run asynchronously with user program processing, and since I/O processing required the CPU only for short periods to initiate data channel instructions, the CPU was dispatched to the reader the writer, and the program in that order. Moreover, if the writer or the user program was executing when something became available to read, the reader program would preempt the currently executing program to regain control of the CPU for its initiation instruction, and the writer program would preempt the user program for the same purpose. This rule, called the static priority rule with preemption, was implemented in the operating system as a system dispatcher program.

The spooling operating system, in fact, had multiprogramming since more than one program was resident in main storage at the same time. Later this basic idea of multiprogramming was extended to include more than one active user program in memory at time. To accommodate this extension, both the scheduler and the dispatcher were enhanced. The scheduler became able to manage the diverse resource needs of the several concurrently active use programs, and the dispatcher included policies for allocating processor resources among the competing user programs. In addition, memory management became more sophisticated in order to assure that the program code for each job or at least that part of the code being executed, was resident in main storage.

The advent of large scale multiprogramming was made possible by several important hardware innovations. The first was the widespread availability of large capacity, high speed disk units to accommodate the spooled input streams and the memory overflow together with the maintenance of several concurrently active program in execution. The second was relocation hardware which facilitated the moving of blocks of code within memory without an undue overhead penalty. Third was the availability of storage protecting hardware to ensure that user jobs are protected from one another and that the operating system itself is protected from user programs. Some of these hardware innovations involved extensions to the interrupt system in order to handle a variety of external conditions such as program malfunctions, storage protection violations, and machine checks in addition to I/O interrupts. In addition, the interrupt system became the technique for the user program to request services from the operating system kernel. Finally, the advent of privileged instructions allowed the operating system to maintain coordination and control over the multiple activities now going on within the system.

Successful implementation of multiprogramming opened the way for the development of a new way of delivering computing services-time-sharing. In this environment, several terminals, sometimes up to 200 of them, were attached (hard wired or via telephone lines) to a central computer. Users at their terminals, "logged in" to the central system, and worked interactively with the system. The system’s apparent concurrency was enabled by the multiprogramming operating system. Users shared not only the system’ hardware but also its software resources and file system disk space.

The third generation was an exciting time, indeed, for the development of both computer hardware and the accompanying operating system. During this period, the topic of operating systems became, in reality, a major element of the discipline of computing.

3.1.5 The Fourth Generation, 1979 - Present

The fourth generation is characterized by the appearance of the personal computer and the workstation. Miniaturization of electronic circuits and components continued and large scale integration (LSI), the component technology of the third generation, was replaced by very large scale integration (VLSI), which characterizes the fourth generation. VLSI with its capacity for containing thousands of transistors on a small chip, made possible the development of desktop computers with capabilities exceeding those that filled entire rooms and floors of building just twenty years earlier.

The operating system that control these desktop machines have brought us back in a full circle, to the open shop type of environment where each user occupies an entire computer for the duration of a job's execution. This works better now, not only because the progress made over the years has made the virtual computer resulting from the operating system/hardware combination so much easier to use or, in the words of the popular press "user-friendly."

However, improvements in hardware miniaturization and technology have evolved so fast that we now have inexpensive workstation-class computer capable of supporting multiprogramming and time-sharing. Hence the operating systems that supports today's personal computers and workstations look much like those which were available for the minicomputers of the third generation. Examples are Microsoft's DOS for IBM-compatible personal computers and UNIX for workstation. However, many of these desktop computers are now connected as networked or distributed systems. Computers in a networked system each have their operating system augmented with communication capabilities that enable users to remotely log into any system on the network and transfer information among machines that are connected to the network. The machines that make up distributed system operate as a virtual single processor system from the user's point of view; a central operating system controls and makes transparent the location in the system of the particular processor or processors and file systems that are handling any given program.

4.0 Conclusion

As you have learnt in this unit, first computers did not have operating systems, but as technology advances through the 1960s, several major concepts were developed, driving the development of operating systems. In this unit, you have been introduced to the brief history and evolution of operating system. The knowledge of OS, being an important system software without which today's computers would not function, is crucial to your being able to work with the computer system.

5.0 Summary

Since you are going to be interacting with the computer machine in your day-to-day activities as a computer user or professional, it is necessary to have the basic knowledge of OS. In the next unit, you are going to be introduced to the various types of OS in the market today based on several criteria.

6.0. Tutor Marked Assignment

You are to attempt the following assignments and submit your answers to your tutor for this course. Here we go:

1. What is an OS?
2. What led to the invention of the OS?
3. Describe the characteristic features of the second generation OS
4. What distinguishes the fourth generation OS from the third generation OS an what improvement in the computer architecture led to this?

7.0 References/Further Reading

1. Per Brinch Hansen (2001), *Classic operating systems: from batch processing to distributed systems*. New York,: Springer-Verlag, 1-36. ISBN 0-387-95113-X.
2. Deitel, Harvey M.; Deitel, Paul; Choffnes, David (2004). *Operating Systems*. Upper Saddle River, NJ: Pearson/Prentice Hall. ISBN 0-13-182827-4.
3. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5.
4. Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems. Design and Implementation*. Upper Saddle River, N.J.: Pearson/Prentice Hall. ISBN 0-13-142938-8.
5. Tanenbaum, Andrew S. (2001). *Modern Operating Systems*. Upper Saddle River, N.J.: Prentice Hall. ISBN 0-13-092641-8.

Module 1: Operating System Fundamentals

Unit 3: The Kernel

Table of Contents

1.0 Introduction

In the previous unit we discussed the history and evolution of the operating system. In this unit you will be taken through the core component of the operating system which is the kernel.

2.0 Objectives

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

- Define the kernel
- Describe the functions/responsibilities of the kernel
- Explain its design philosophies/decisions

- Describe the various kernel-wide design approaches

3.0 Main Body

3.1 Kernel Overview

A kernel connects the application software to the hardware of a computer.

In computer science, the **kernel** is the central component of most computer operating systems (OS). Its responsibilities include managing the system's resources and the communication between hardware and software components. As a basic component of an operating system, a kernel provides the lowest-level abstraction layer for the resources (especially memory, processors and I/O devices) that applications must control to perform their function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls.

These tasks are done differently by different kernels, depending on their design and implementation. While monolithic kernels will try to achieve these goals by executing all the code in the same address space to increase the performance of the system, microkernels run most of their services in user space, aiming to improve maintainability and modularity of the codebase. A range of possibilities exists between these two extremes

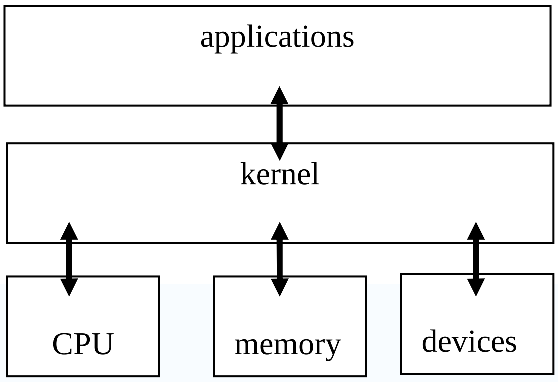


Figure 3.1: The kernel connecting the application software to the hardware of a computer.

Most operating systems rely on the kernel concept. The existence of a kernel is a natural consequence of designing a computer system as a series of abstraction layers, each relying on the functions of layers beneath itself. The kernel, from this viewpoint, is simply the name given to the lowest level of abstraction that is implemented in software. In order to avoid having a kernel, one would have to design all the software on the system not to use abstraction layers; this would increase the complexity of the design to such a point that only the simplest systems could feasibly be implemented.

While it is today mostly called the *kernel*, the same part of the operating system has also in the past been known as the ***nucleus*** or ***core***. (You should note, however, that the term *core* has also been used to refer to the primary memory of a computer system, typically because some early computers used a form of memory called Core memory.)

In most cases, the boot loader starts executing the kernel in supervisor mode, The kernel then initializes itself and starts the first process. After this, the kernel does not typically execute directly, only in response to external events (e.g. via system calls used by applications to request services from the kernel, or via interrupts used by the hardware to notify the kernel of events). Additionally, the kernel typically provides a loop that is executed whenever no processes are available to run; this is often called the *idle process*.

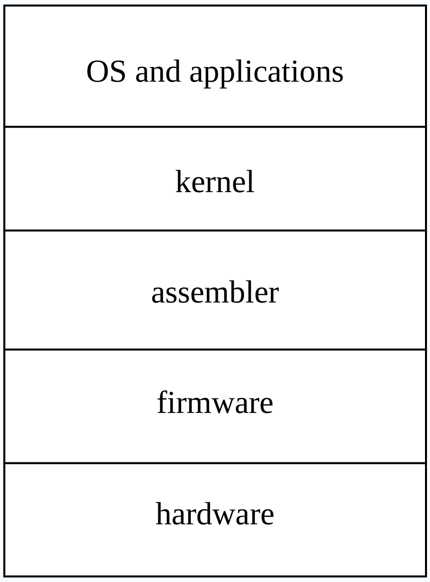


Figure 3.2: A typical vision of a computer architecture as a series of abstraction layers: hardware, firmware, assembler, kernel, operating system and applications.

Kernel development is considered one of the most complex and difficult tasks in programming. Its central position in an operating system implies the necessity for good performance, which defines the kernel as a critical piece of software and makes its correct design and implementation difficult. For various reasons, a kernel might not even be able to use the abstraction mechanisms it provides to other software. Such reasons include memory management concerns (for example, a user-mode function might rely on memory being subject to demand paging, but as the kernel itself provides that facility it cannot use it, because then it might not remain in memory to provide that facility) and lack of reentrancy, thus making its development even more difficult for software engineers.

A kernel will usually provide features for low-level scheduling of processes (dispatching), Inter-process communication, process synchronization, context switch, manipulation of process control blocks, interrupt handling, process creation and destruction, process suspension and resumption (see process states in the next module).

3.2 Kernel Basic Responsibilities

The kernel's primary purpose is to manage the computer's resources and allow other programs to run and use these resources. Typically, the resources consist of:

- The CPU (frequently called the processor). This is the most central part of a computer system, responsible for *running* or *executing* programs on it. The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors (each of which can usually run only one program at once)
- The computer's memory. Memory is used to store both program instructions and data. Typically, both need to be present in memory in order for a program to execute. Often multiple programs will want access to memory, frequently demanding more memory than the computer has available. The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough is available.
- Any Input/Output (I/O) devices present in the computer, such as disk drives, printers, displays, etc. The kernel allocates requests from applications to perform I/O to an appropriate device (or subsection of a device, in the case of files on a disk or windows on a display) and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device)

Kernels also usually provide methods for synchronization and communication between processes (called *inter-process communication* or IPC). This is discussed in module 3.

A kernel may implement these features itself, or rely on some of the processes it runs to provide the facilities to other processes, although in this case it must provide some means of IPC to allow processes to access the facilities provided by each other.

Finally, a kernel must provide running programs with a method to make requests to access these facilities.

3.2.1 Process Management

The main task of a kernel is to allow the execution of applications and support them with features such as hardware abstractions. To run an application, a kernel typically sets up an address space for the application, loads the file containing the application's code into memory (perhaps via demand paging), sets up a stack for the program and branches to a given location inside the program, thus starting its execution.

Multi-tasking kernels are able to give the user the illusion that the number of processes being run simultaneously on the computer is higher than the maximum number of processes the computer is physically able to run simultaneously. Typically, the number of processes a system may run simultaneously is equal to the number of CPUs installed (however this may not be the case if the processors support simultaneous multithreading).

In a pre-emptive multitasking system, the kernel will give every program a slice of time and switch from process to process so quickly that it will appear to the user as if these processes were being executed simultaneously. The kernel uses scheduling algorithms to determine which process is running next and how much time it will be given. The algorithm chosen may allow for some processes to have higher priority than others. The kernel generally also provides these processes a way to communicate; this is known as inter-process communication (IPC) and the main approaches are shared memory, message passing and remote procedure calls (see module 3).

Other systems (particularly on smaller, less powerful computers) may provide co-operative multitasking, where each process is allowed to run uninterrupted until it makes a special request that tells the kernel it may switch to another process. Such requests are known as "yielding", and typically occur in response to requests for interprocess communication, or for waiting for an event to occur. Older versions of Windows and Mac OS both used co-operative multitasking but switched to pre-emptive schemes as the power of the computers to which they were targeted grew.

The operating system might also support multiprocessing (SMP or Non-Uniform Memory Access); in that case, different programs and threads may run on different processors. A kernel for such a system must be designed to be re-entrant, meaning that it may safely run two different parts of its code simultaneously. This typically means providing synchronization mechanisms (such as spinlocks) to ensure that no two processors attempt to modify the same data at the same time.

3.2.2 Memory Management

The kernel has full access to the system's memory and must allow processes to access this memory safely as they require it. Often the first step in doing this is virtual addressing, usually achieved by paging and/or segmentation. Virtual addressing allows the kernel to make a given physical address appear to be another address, the virtual address. Virtual address spaces may be different for different processes; the memory that one process accesses at a particular (virtual) address may be different memory from what another process accesses at the same address. This allows every program to behave as if it is the only one (apart from the kernel) running and thus prevents applications from crashing each other.

On many systems, a program's virtual address may refer to data which is not currently in memory. The layer of indirection provided by virtual addressing allows the operating system to use other data stores, like a hard drive, to store what would otherwise have to remain in main memory (RAM). As a result, operating systems can allow programs to use more memory than the system has physically available. When a program needs data which is not currently in RAM, the CPU signals to the kernel that this has happened, and the kernel responds by writing the contents of an inactive memory block to disk (if necessary) and replacing it with the data requested by the program. The program can then be resumed from the point where it was stopped. This scheme is generally known as demand paging.

Virtual addressing also allows creation of virtual partitions of memory in two disjointed areas, one being reserved for the kernel (kernel space) and the other for the applications (user space). The applications are not permitted by the processor to address kernel memory, thus preventing an application from damaging the running kernel. This fundamental partition of memory space has contributed much to current designs of actual general-purpose kernels and is almost universal in such systems, although some research kernels (e.g. Singularity) take other approaches.

3.2.3 Device Management

To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. For example, to show the user something on the screen, an application would make a request to the kernel, which would forward the request to its display driver, which is then responsible for actually plotting the character/pixel.

A kernel must maintain a list of available devices. This list may be known in advance (e.g. on an embedded system where the kernel will be rewritten if the available hardware changes), configured by the user (typical on older PCs and on systems that are not designed for personal use) or detected by the operating system at run time (normally called Plug and Play).

In a plug and play system, a device manager first performs a scan on different hardware buses, such as Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB), to detect installed devices, then searches for the appropriate drivers.

As device management is a very OS-specific topic, these drivers are handled differently by each kind of kernel design, but in every case, the kernel has to provide the I/O to allow drivers to physically access their devices through some port or memory location. Very important decisions have to be made when designing the device management system, as in some designs accesses may involve context switches, making the operation very CPU-intensive and easily causing a significant performance overhead.

3.2.4 System Calls

To actually perform useful work, a process must be able to access the services provided by the kernel. This is implemented differently by each kernel, but most provide a C library or an API, which in turn invoke the related kernel functions.

The method of invoking the kernel function varies from kernel to kernel. If memory isolation is in use, it is impossible for a user process to call the kernel directly, because that would be a violation of the processor's access control rules. A few possibilities are:

- Using a software-simulated interrupt. This method is available on most hardware, and is therefore very common.
- Using a call gate. A call gate is a special address which the kernel has added to a list stored in kernel memory and which the processor knows the location of.

When the processor detects a call to that location, it instead redirects to the target location without causing an access violation. Requires hardware support, but the hardware for it is quite common.

- Using a special system call instruction. This technique requires special hardware support, which common architectures (not ably, x86) may lack. System call instructions have been added to recent models of x86 processors, however, and some (but not all) operating systems for PCs make use of them when available.
- Using a memory-based queue. An application that makes large numbers of requests but does not need to wait for the result of each may add details of requests to an area of memory that the kernel periodically scans to find requests.

3.3 Kernel Design Decisions

3.3.1 Fault Tolerance

An important consideration in the design of a kernel is fault tolerance; specifically, in cases where multiple programs are running on a single computer, it is usually important to prevent a fault in one of the programs from negatively affecting the other. Extended to malicious design rather than a fault, this also applies to security, and is necessary to prevent processes from accessing information without being granted permission.

Two main approaches to the protection of sensitive information are assigning privileges to hierarchical protection domains, for example by using a processor's supervisor mode, or distributing privileges differently for each process and resource, for example by using capabilities or access control lists.

Hierarchical protection domains are much less flexible, as it is not possible to assign different privileges to processes that are at the same privileged level, and ca not therefore satisfy Denning's four principles for fault tolerance (particularly the Principle of least privilege). Hierarchical protection domains also have a major performance drawback, since interaction between different levels of protection, when a process has to manipulate a data structure both in 'user mode' and 'supervisor mode', always requires message copying (transmission by value). A kernel based on capabilities, however, is more flexible in assigning privileges, can satisfy Denning's fault tolerance principles, and typically does not suffer from the performance issues of copy by value.

Both approaches typically require some hardware or firmware support to be operable and efficient. The hardware support for hierarchical protection domains is typically that of "CPU modes." An efficient and simple way to provide hardware support of capabilities is to delegate the MMU the responsibility of checking access-rights for every memory access, a mechanism called capability-based addressing. Most commercial computer architectures lack MMU support for capabilities. An alternative approach is to simulate capabilities using commonly-support hierarchical domains; in this approach, each protected object must reside in an address space that the application does not have access to; the kernel also maintains a list of capabilities in such memory. When an application needs to access an object protected by a capability, it performs a system call and the kernel performs the access for it. The performance cost of address space switching limits the practicality of this approach in systems with complex interactions between objects, but it is used in current operating systems for objects that are not accessed frequently or which are not expected to perform quickly. Approaches where protection mechanism are not firmware supported but are instead simulated at higher levels (e.g. simulating capabilities by manipulating page tables on hardware that does not have direct support), are possible, but there are performance implications. Lack of hardware support may not be an issue, however, for systems that choose to use language-based protection.

3.3.2 Security

An important kernel design decision is the choice of the abstraction levels where the security mechanisms and policies should be implemented. One approach is to use firmware and kernel support for fault tolerance (see above), and build the security policy for malicious behaviour on top of that (adding features such as cryptography mechanisms where necessary), delegating some responsibility to the compiler. Approaches that delegate enforcement of security policy to the compiler and/or the application level are often called *language-based security*.

3.3.3 Hardware-Based Protection or Language-Based Protection

Typical computer systems today use hardware-enforced rules about what programs are allowed to access what data. The processor monitors the execution and stops a program that violates a rule (e.g., a user process that is about to read or write to kernel memory, and so on). In systems that lack support for capabilities, processes are isolated from each other by using separate address spaces. Calls from user processes into the kernel are regulated by requiring them to use one of the above-described system call methods.

An alternative approach is to use language-based protection. In a language-based protection system, the kernel will only allow code to execute that has been produced by a trusted language compiler. The language may then be designed such that it is impossible for the programmer to instruct it to do something that will violate a security requirement.

Advantages of this approach include:

- Lack of need for separate address spaces. Switching between address spaces is a slow operation that causes a great deal of overhead, and a lot of optimization work is currently performed in order to prevent unnecessary switches in current operating systems. Switching is completely unnecessary in a language-based protection system, as all code can safely operate in the same address space.
- Flexibility. Any protection scheme that can be designed to be expressed via a programming language can be implemented using this method. Changes to the protection scheme (e.g. from a hierarchical system to a capability-based one) do not require new hardware.

Disadvantages include:

- Longer application start up time. Applications must be verified when they are started to ensure they have been compiled by the correct compiler, or may need recompiling either from source code or from bytecode.
- Inflexible type systems. On traditional systems, applications frequently perform operations that are not type safe. Such operations cannot be permitted in a language-based protection system, which means that applications may need to be rewritten and may, in some cases, lose performance.

Examples of systems with language-based protection include JX and Microsoft's Singularity.

3.3.4 Process cooperation

Edsger Dijkstra proved that from a logical point of view, atomic lock and unlock operations operating on binary semaphores are sufficient primitives to express any functionality of process cooperation. However this approach is generally held to be lacking in terms of safety and efficiency, whereas a message passing approach is more flexible.

3.3.5 I/O devices management

The idea of a kernel where I/O devices are handled uniformly with other processes, as parallel co-operating processes, was first proposed and implemented by Brinch Hansen (although similar ideas were suggested in 1967). In Hansen's description of this, the "common" processes are called *internal processes*, while the I/O devices are called *external processes*.

3.4 Kernel-Wide Design Approaches

Naturally, the above listed tasks and features can be provided in many ways that differ from each other in design and implementation. While monolithic kernels execute all of their code in the same address space (kernel space) to increase the performance of the system, microkernels try to run most of their services in user space, aiming to improve maintainability and modularity of the codebase. Most kernels do not fit exactly into one of these categories, but are rather found in between these two designs. These are called hybrid kernels. More exotic designs such as nanokernels and exokernels are available, but are seldom used for production systems. The Xen hypervisor, for example, is an exokernel.

The principle of *separation of mechanism and policy* is the substantial difference between the philosophy of micro and monolithic kernels. Here a *mechanism* is the support that allows the implementation of many different policies, while a *policy* is a particular "mode of operation". In minimal microkernel just some very basic policies are included, and its mechanisms allows what is running on top of the kernel (the remaining part of the operating system and the other applications) to decide which policies to adopt (as memory management, high level process scheduling, file system management, etc.). A monolithic kernel instead tends to include many policies, therefore restricting the rest of the system to rely on them.

3.4.1 Monolithic kernels

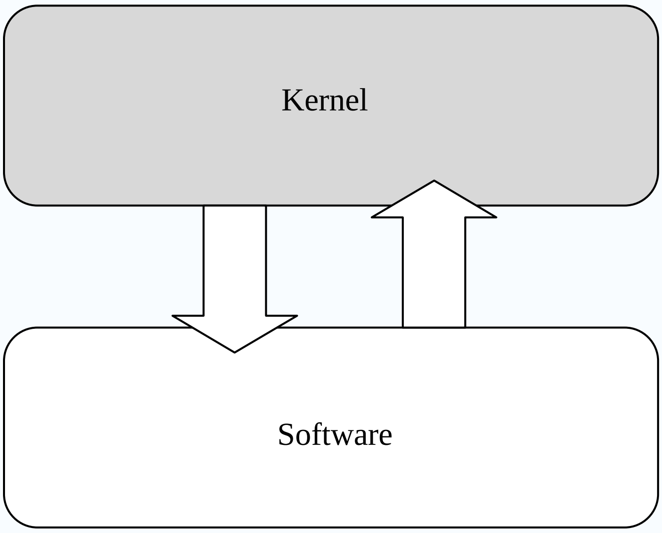


Figure 3.3: Graphical overview of a monolithic kernel

In a monolithic kernel, all OS services run along with the main kernel thread, thus also residing in the same memory area. This approach provides rich and powerful hardware access. Some developers maintain that monolithic systems are easier to design and implement than other solutions, and are extremely efficient if well-written. The main disadvantages of monolithic kernels are the dependencies between system components - a bug in a device driver might crash the entire system - and the fact that large kernels can become very difficult to maintain.

3.4.2 Microkernels

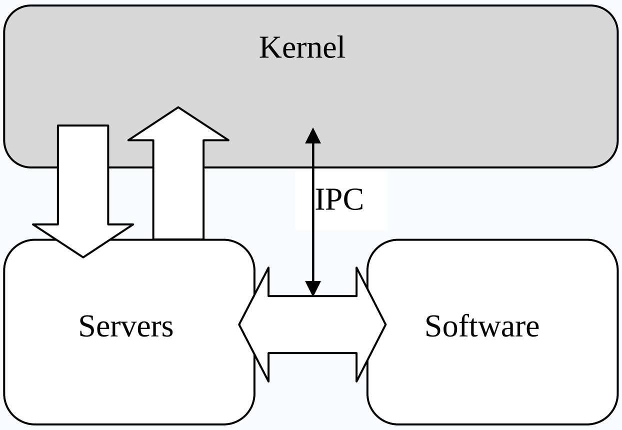


Figure 3.4: Diagram of Microkernels Approach

In the microkernel approach, the kernel itself only provides basic functionality that allows the execution of servers, separate programs that assume former kernel functions, such as device drivers, GUI servers, etc.

The microkernel approach consists of defining a simple abstraction over the hardware, with a set of primitives or system calls to implement minimal OS services such as memory management, multitasking, and inter-process communication. Other services, including those normally provided by the kernel such as networking, are implemented in user-space programs, referred to as *servers*. Microkernels are easier to maintain than monolithic kernels, but the large number of system calls and context switches might slow down the system because they typically generate more overhead than plain function calls.

Microkernels generally underperform traditional designs, sometimes dramatically. This is due in large part to the overhead of moving in and out of the kernel, a context switch, to move data between the various applications and servers. By the mid-1990s, most researchers had abandoned the belief that careful tuning could reduce this overhead dramatically, but recently, newer microkernels, optimized for performance, such as L4 and K42 have addressed these problems.

A microkernel allows the implementation of the remaining part of the operating system as a normal application program written in a high-level language, and the use of different operating systems on top of the same unchanged kernel. It is also possible to dynamically switch among operating systems and to have more than one active simultaneously.

3.4.3 Monolithic kernels Vs. Microkernels

As the computer kernel grows, a number of problems become evident. One of the most obvious is that the memory footprint increases. This is mitigated to some degree by perfecting the virtual memory system, but not all computer architectures have virtual memory support. To reduce the kernel's footprint, extensive editing has to be performed to carefully remove unneeded code, which can be very difficult with non-obvious interdependencies between parts of a kernel with millions of lines of code.

Due to the problems that monolithic kernels pose, they were considered obsolete by the early 1990s. As a result, the design of Linux using a monolithic kernel rather than a microkernel was the topic of a famous flame war between Linus Torvalds and Andrew Tanenbaum. There is merit on both sides of the argument presented in the Tanenbaum/Torvalds debate.

Some, including early UNIX developer Ken Thompson, argued that while microkernel designs were more aesthetically appealing, monolithic kernels were easier to implement. However, a bug in a monolithic system usually crashes the entire system, while this does not happen in a microkernel with servers running apart from the main thread. Monolithic kernel proponents reason that incorrect code does not belong in a kernel, and that microkernels offer little advantage over correct code. Microkernels are often used in embedded robotic or medical computers where crash tolerance is important and most of the OS components reside in their own private, protected memory space. This is impossible with monolithic kernels, even with modern module-loading ones. However, the monolithic model tends to be more efficient through the use of shared kernel memory, rather than the slower IPC system of microkernel designs, which is typically based on message passing.

3.4.4 Hybrid kernels

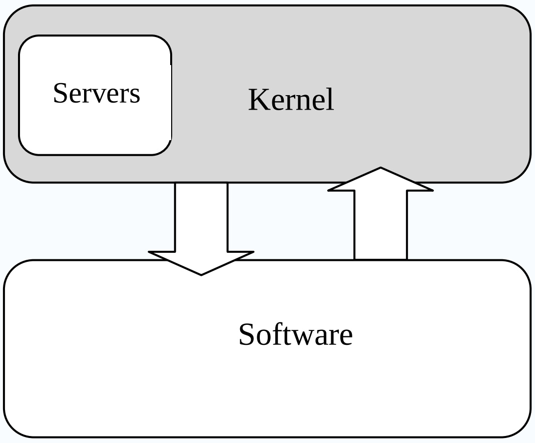


Figure 3.5: Diagram of Hybrid kernels Approach

Hybrid kernel is a kernel architecture based on combining aspects of microkernel and monolithic kernel architectures used in computer operating systems. The category is controversial due to the similarity to monolithic kernel; the term has been dismissed by some as just marketing. The usually accepted categories are monolithic kernels and microkernels (with nanokernels and exokernels seen as more extreme versions of microkernels).

The hybrid kernel approach tries to combine the speed and simpler design of a monolithic kernel with the modularity and execution safety of a microkernel.

Hybrid kernels are essentially a compromise between the monolithic kernel approach and the microkernel system. This implies running some services (such as the network stack or the file system) in kernel space to reduce the performance overhead of a traditional microkernel, but still running kernel code (such as device drivers) as servers in user space.

The idea behind this quasi-category is to have a kernel structure similar to a microkernel, but implemented as a monolithic kernel. In contrast to a microkernel, all (or nearly all) services are in kernel space. As in a monolithic kernel, there is no performance overhead associated with microkernel message passing and context switching between kernel and user mode. Also, as with monolithic kernels, there are none of the benefits of having services in user space.

3.4.5 Nanokernels

The term is sometimes used informally to refer to a very light-weight microkernel, such as L4.

A **nanokernel** or **picokernel** is a very minimalist operating system kernel. The nanokernel represents the closest hardware abstraction layer of the operating system by interfacing the CPU, managing interrupts and interacting with the MMU. The interrupt management and MMU interface are not necessarily part of a nanokernel; however, on most architectures these components are directly connected to the CPU, therefore, it often makes sense to integrate these interfaces into the kernel.

A nanokernel delegates virtually all services – including even the most basic ones like interrupt controllers or the timer – to device drivers to make the kernel memory requirement even smaller than a traditional microkernel.

software.

Advantages and Disadvantages

Nanokernels Versus Monolithic kernels

A nanokernel is considered to be slower than a typical monolithic kernel due to the management and communication complexity caused by the separation of its components. Contrariwise this abstraction potentiates considerably faster development, simpler modules and higher code quality. Additionally the management effort of such code is not ably decreased because monolithic implementations tend to be more complex and intradependent. As a result of its lower module complexity nanokernel modules tend to be more accurate and maintainable.

Furthermore APIs of monolithic kernels (as present in for example the Linux kernel) are often considered to be very unstable and quite mutable. It is often argued that this applies only to some implementations, but in reality monolithic drivers use more internal structures than separated modules.

Another key aspect is the isolation of the nanokernel modules by architecture. Monolithic kernels generally suffer from a considerably bad security architecture because an inaccurate and insecure part directly affects the whole operating system.

Nanokernels Versus microkernels

Generally microkernels have integrated IPC, memory-, thread- and process management and elementary drivers. A nanokernel in contrast has essentially none of those, therefore nanokernels are not independently executable operating systems, which is why they are not an operating system kernel in the traditional sense.

Nevertheless this significant difference potentiates extremely powerful techniques like multi scheduling or operating system emulation. The simultaneous execution of a realtime- and pre-emptive scheduler on multi processor machines or the emulation of an entire operating system like UNIX in heterogeneous environments are some application areas of this technique. But in general this superiority applies and occurs only significantly in parallel- or distributed computing environments.

3.4.6 Exokernels

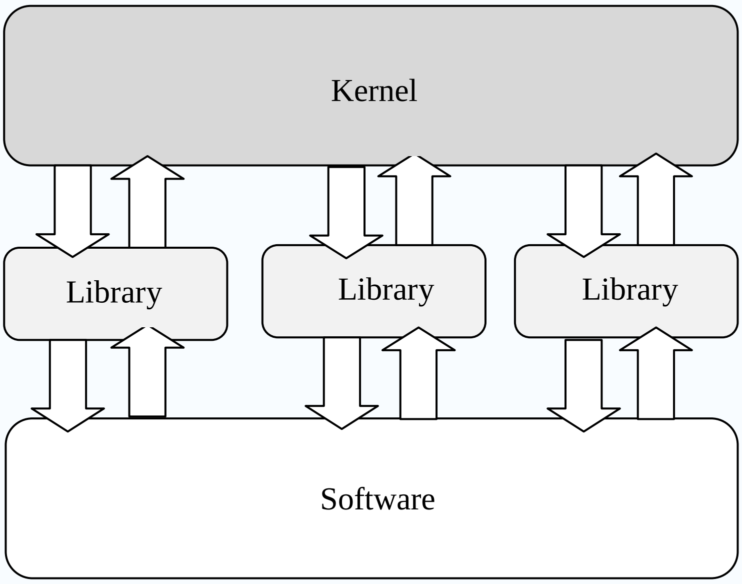


Figure 3.6: Graphical overview of Exokernel

An exokernel is a type of kernel that does not abstract hardware into theoretical models. Instead it allocates physical hardware resources, such as processor time, memory pages, and disk blocks, to different programs. A program running on an exokernel can link to a *library operating system* that uses the exokernel to simulate the abstractions of a well-known OS, or it can develop application-specific abstractions for better performance.

4.0 Conclusion

In this unit you have been taken through the concept of kernel and its importance in operating systems design. Also, its various responsibilities in the computer environment had been deeply discussed not leaving behind the kernel design issues and trade-offs.

5.0 Summary

Strictly speaking, an operating system (and thus, a kernel) is not *required* to run a computer. Programs can be directly loaded and executed on the "bare metal" machine, provided that the authors of those programs are willing to work without any hardware abstraction or operating system support. Most early computers operated this way during the 1950s and early 1960s, which were reset and reloaded between the execution of different programs. Eventually, small ancillary programs such as program loaders and debuggers were left in memory between runs, or loaded from ROM. As these were developed, they formed the basis of what became early operating system kernels. The "bare metal" approach is still used today on some video game consoles and embedded systems, but in general, newer computers use modern operating systems and kernels.

6.0 Tutor Marked Assignments

1. In the context of Kernel design decisions, distinguish between harware-based and language-based protection.
2. Differentiate between monolithic kernels and microkernels.
3. Briefly describe the hybrid kernel concept.
4. Itemize and briefly explain the various issues in kernel design.
5. Enumerate the various responsibilities of the kernel.

7.0 References/Further Reading

1. Roch, Benjamin (2004). Monolithic kernel vs. Microkenrel (pdf). Retrieved on 2006-10-12.
2. Silberschatz, Abraham; James L. Peterson, Peter B. Galvin (1991). *Operating system concepts*. Boston, Massachusetts: Addison-Wesley, 696. ISBN 0-201-51379-X.
3. Hansen, Per Brinch (April 1970). "The nucleus of a Multiprogramming System". *Communications of the ACM* **13** (4): 238-241. ISSN 0001-0782.
4. Deitel, Harvey M. [1982] (1984). *An introduction to operating systems*, revisited first edition, Addison-Wesley, 673. ISBN 0-201-14502-2.
5. Denning, Peter J. (April 1980). "Why not innovations in computer architecture?". *ACM SIGARCH Computer Architecture News* **8** (2): 4-7. ISSN 0163-5964.
6. Hansen, Per Brinch [1973]. *Operating System Principles*. Englewood Cliffs: Prentice Hall, 496. ISBN 0-13-637843-9.
7. Per Brinch Hansen (2001). "*The evolution of operating systems*" (pdf). Retrieved on 2006-10-24. included in book: [2001] "1", in Per Brinch Hansen: *Classic operating systems: from batch processing to distributed systems*. New York,: Springer-Verlag, 1-36. ISBN 0-387-95113-X.
8. Levin, R.; E. Cohen, W. Corwin, F. Pollack, W. Wulf (1975). "Policy/mechanism separation in Hydra". *ACM Symposium on Operating Systems Principles / Proceedings of the fifth ACM symposium on Operating systems principles*: 132-140.
9. Linden, Theodore A. (December 1976). "Operating System Structures to Support Security and Reliable Software". *ACM Computing Surveys (CSUR)* **8** (4): 409 - 445. ISSN 0360-0300.
10. Lorin, Harold (1981). *Operating systems*. Boston, Massachusetts: Addison-Wesley, pp.161-186. ISBN 0-201-14464-6.
11. Schroeder, Michael D.; Jerome H. Saltzer (March 1972). "A hardware architecture for implementing protection rings". *Communications of the ACM* **15** (3): 157 - 170. ISSN 0001-0782.

12. Shaw, Alan C. (1974). *The logical design of Operating systems*. Prentice-Hall, 304. ISBN 0-13-540112-7.
13. Tanenbaum, Andrew S. (1979). *Structured Computer Organization* (in English). Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-148521-0.
14. Wulf, W.; E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack (June 1974). "HYDRA: the kernel of a multiprocessor operating system". *Communications of the ACM* **17** (6): 337 - 345. ISSN 0001-0782.
15. Andrew Tanenbaum, *Operating Systems - Design and Implementation (Third edition)*;
16. Andrew Tanenbaum, *Modern Operating Systems (Second edition)*;
17. Baiardi, F.; A. Tomasi, M. Vanneschi (1988). *Architettura dei Sistemi di Elaborazione, volume 1* (in Italian). Franco Angeli. ISBN 88-204-2746-X.
18. Daniel P. Bovet, Marco Cesati, *The Linux Kernel*;
19. David A. Peterson, Nitin Indurkhya, Patterson, *Computer Organization and Design*, Morgan Koffman (ISBN 1-55860-428-6);
20. B.S. Chalk, *Computer Organisation and Architecture*, Macmillan P.(ISBN 0-333-64551-0).

Module 1: Operating System Fundamentals

Unit 4: Types of Operating Systems

Table of Contents

1.0 Introduction

In the last unit you have been introduced to the concept and history of operating system and how it evolved with each discovery and improvement in the technology of computer architecture. In this unit, you are presented with types of operating system based on:

(i) The types of computer they control and the sort of applications they support

(ii) The nature of interaction that takes place between the computer user and his/her program during its processing.

2.0 Objectives

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

- Categorise operating systems based on various criteria

- List the basic features of each type of operating system
- Distinguish between one type of operating system and another

3.0 Types of OS

OS can be categorised in different ways based on perspectives. Some of the major ways in which the OS can be classified are explored and introduced in this unit.

3.1 Types of Operating Systems Based on the Types of Computer they Control and the Sort of Applications they Support

Based on the types of computers they control and the sort of applications they support, there are generally four types within the broad family of operating systems. The broad categories are as follows:

3.1.1 Real-Time Operating Systems (RTOS):

They are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-interface capability, and no end-user utilities, since the system will be a sealed box when delivered for use. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time every time it occurs. In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not move at all because the system is busy. RTOS can be hard or soft. A hard RTOS guarantees that critical tasks are performed on time. However, soft RTOS is less restrictive. Here, a critical real-time task gets priority over other tasks and retains that priority until it completes.

3.1.2 Single-User, Single-Tasking Operating System:

As the name implies, this operating system is designed to manage the computer so that one user can effectively do one thing at a time. The Palm OS for Palm handheld computers is a good example of a modern single-user, single-task operating system.

3.1.3 Single-User, Multi-Tasking Operating System:

This is the type of operating system most people use on their desktop and laptop computers today. **Windows 98** and the **Mac O.S.** are both examples of an operating system that will let a single user have several programs in operation at the same time. For example, it is entirely possible for you as a Windows user to be writing a note in a word processor while downloading a file from the Internet and at the same time be printing the text of an e-mail message.

3.1.4 Multi-User Operating Systems:

A multi-user operating system allows many different users to take advantage of the computer's resources simultaneously. The operating system must make sure that the requirements of the various users are balanced, and that each of the programs they are using has sufficient and separate resources so that a problem with one user does not affect the entire community of users. **Unix, VMS**, and mainframe operating systems, such as **MVS**, are examples of multi-user operating systems. It's important to differentiate here between multi-user operating systems and single-user operating systems that support networking.

Windows 2000 and **Novell Netware** can each support hundreds or thousands of networked users, but the operating systems themselves are not true multi-user operating systems. The system administrator is the only user for Windows 2000 or Netware. The network support and the entire remote user logins the network enables are, in the overall plan of the operating system, a program being run by the administrative user.

3.2 Types of OS based on the Nature of Interaction that takes place between the Computer User and His/Her Program during its Processing

Modern computer operating systems may be classified into three groups, which are distinguished by the nature of interaction that takes place between the computer user and his or her program during its processing. The three groups are: called batch, time-shared and real time operating systems.

3.2.1 Batch Processing OS

In a batch processing operating system environment, users submit jobs to a central place where these jobs are collected into a batch, and subsequently placed on an input queue at the computer where they will be run. In this case, the user has no interaction with the job during its processing, and the computer’s response time is the turnaround time (i.e. the time from submission of the job until execution is complete, and the results are ready for return to the person who submitted the job).

3.2.2 Time Sharing OS

Another mode for delivering computing services is provided by time sharing operating systems. In this environment a computer provides computing services to several or many users concurrently on-line. Here, the various users are sharing the central processor, the memory, and other resources of the computer system in a manner facilitated, controlled, and monitored by the operating system. The user, in this environment, has nearly full interaction with the program during its execution, and the computer’s response time may be expected to be no more than a few second.

3.2.3 Real Time OS

The third class of operating systems, real time operating systems, are designed to service those applications where response time is of the essence in order to prevent error, misrepresentation or even disaster. Examples of real time operating systems are those which handle airlines reservations, machine tool control, and monitoring of a nuclear power station. The systems, in this case, are designed to be interrupted by external signal that require the immediate attention of the computer system.

In fact, many computer operating systems are hybrids, providing for more than one of these types of computing service simultaneously. It is especially common to have a background batch system running in conjunction with one of the other two on the same computer.

3.3 Other Types of OS based on the Definition of the System/Environment

A number of other definitions are important to gaining a better understanding and subsequently classifying operating systems:

3.3.1 Multiprogramming Operating System

A multiprogramming operating system is a system that allows more than one active user program (or part of user program) to be stored in main memory simultaneously.

Thus, it is evident that a time-sharing system is a multiprogramming system, but note that a multiprogramming system is not necessarily a time-sharing system. A batch or real time operating system could, and indeed usually does, have more than one active user program simultaneously in main storage. Another important, and all too similar, term is ‘multiprocessing’.

A multiprocessing system is a computer hardware configuration that includes more than one independent processing unit. The term multiprocessing is generally used to refer to large computer hardware complexes found in major scientific or commercial applications.

3.3.2 Network Operating Systems

A networked computing system is a collection of physical interconnected computers. The operating system of each of the interconnected computers must contain, in addition to its own stand-alone functionality, provisions for handling communication and transfer of programs and data among the other computers with which it is connected.

In a network operating system, the users are aware of the existence of multiple computers, and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own user (or users). Network operating systems are designed with more complex functional capabilities.

Network operating systems are not fundamentally different from single processor operating systems. They obviously need a network interface controller and some low-level software to drive it, as well as programs to achieve remote login and remote files access, but these additions do not change the essential structure of the operating systems.

3.3.3 Distributed Operating Systems

A distributed computing system consists of a number of computers that are connected and managed so that they automatically share the job processing load among the constituent computers, or separate the job load as appropriate particularly configured processors. Such a system requires an operating system which, in addition to the typical stand-alone functionality, provides coordination of the operations and information flow among the component computers.

The distributed computing environment and its operating systems, like networking environment, are designed with more complex functional capabilities. However, a distributed operating system, in contrast to a network operating system, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. In a true distributed system, users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in critical ways. Distributed systems, for example, often allow program to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism achieved.

4.0 Conclusion

The earliest operating systems were developed for mainframe computer architectures in the 1960s and they were mostly batch processing operating systems. The enormous investment in software for these systems caused most of the original computer manufacturers to continue to develop hardware and operating systems that are compatible with those early operating systems. Those early systems pioneered many of the features of modern operating systems..

5.0 Summary

This unit has taken you through some of the various classifications of OS we have based on different criteria. You will need this knowledge as you work in different computer environment. In the next unit we will be discussing the disk operating system.

6.0 Tutor Marked Assignment

You are to attempt the following assignments and submit your answers to your tutor for this course. Here we go:

1. Network operating systems are not fundamentally different from single processor operating systems. Discuss

2. Distinguish between Network OS and Distributed OS.
3. How is a soft RTOS different from hard RTOS
4. List the major features of a multi-user OS.

7.0 References/Further Reading

1. Deitel, Harvey M.; Deitel, Paul; Choffnes, David (2004). *Operating Systems*. Upper Saddle River, NJ: Pearson/Prentice Hall. ISBN 0-13-182827-4.
2. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5.
3. Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems. Design and Implementation*. Upper Saddle River, N.J.: Pearson/Prentice Hall. ISBN 0-13-142938-8.
4. Tanenbaum, Andrew S. (2001). *Modern Operating Systems*. Upper Saddle River, N.J.: Prentice Hall. ISBN 0-13-092641-8.

Module 2: Types of Operating System

Unit 1: Disk operating system

Table of Contents

1.0 Introduction

The previous unit introduced the various types of OS based on different criteria. In this unit, you will be taken through the disk operating system and its various characteristics and examples.

2.0 Objectives

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

- Describe the disk operating system (DOS)
- List the classes of DOS we have
- State what distinguishes the different classes of DOS

3.0 Disk operating system

Disk Operating System (specifically) and **disk operating system** (generically), most often abbreviated as **DOS** (not to be confused with the DOS family of disk operating systems for the IBM PC compatible platform), refer to operating system software used in most computers that provides the abstraction and management of secondary storage devices and the information on them (e.g., file systems for organizing files of all sorts). Such software is referred to as a disk operating system when the storage devices it manages are made of rotating platters (such as hard disks or floppy disks).

In the early days of microcomputing, memory space was often limited, so the disk operating system was an extension of the operating system. This component was only loaded if needed. Otherwise, disk-access would be limited to low-level operations such as reading and writing disks at the sector-level.

In some cases, the disk operating system component (or even the operating system) was known as DOS.

Sometimes, a disk operating system can refer to the entire operating system if it is loaded off a disk and supports the abstraction and management of disk devices. Examples include DOS/360 and FreeDOS. On the PC compatible platform, an entire family of operating systems was called DOS.

In the early days of computers, there were no disk drives; delay lines, punched cards, paper tape, magnetic tape, magnetic drums, were used instead. And in the early days of microcomputers, paper tape or audio cassette tape (see Kansas City standard) or nothing were used instead. In the latter case, program and data entry was done at front panel switches directly into memory or through a computer terminal/keyboard, sometimes controlled by a ROM BASIC interpreter; when power was turned off after running the program, the information so entered vanished.

Both hard disks and floppy disk drives require software to manage rapid access to block storage of sequential and other data. When microcomputers rarely had expensive disk drives of any kind, the necessity to have software to manage such devices (i.e. the 'disks’) carried much status. To have one or the other was a mark of distinction and prestige, and so was having the Disk sort of an Operating System. As prices for both disk hardware and operating system software decreased, there were many such microcomputer systems.

Mature versions of the Commodore, SWTPC, Atari and Apple home computer systems all featured a disk operating system (actually called 'DOS' in the case of the Commodore 64 (CBM DOS), Atari 800 (Atari DOS), and Apple II machines (Apple DOS)), as did (at the other end of the hardware spectrum, and much earlier) IBM's System/360, 370 and (later) 390 series of mainframes (e.g., DOS/360: Disk Operating System / 360 and DOS/VSE: Disk Operating System / Virtual Storage Extended). Most home computer DOS'es were stored on a floppy disk always to be booted at start-up, with the not able exception of Commodore, whose DOS resided on ROM chips in the disk drives themselves, available at power-on.

In large machines there were other disk operating systems, such as IBM's VM, DEC's RSTS / RT-11 / VMS / TOPS-10 / TWENEX, MIT's ITS / CTSS, Control Data's assorted NOS variants, Harris's Vulcan, Bell Labs' Unix, and so on. In microcomputers, SWTPC's 6800 and 6809 machines used TSC's FLEX disk operating system, Radio Shack's TRS-80 machines used TRS-DOS, their Color Computer used OS-9, and most of the Intel 8080 based machines from IMSAI, MITS (makers of the legendary Altair 8800), Cromemco, North Star, etc used the CP/M-80 disk operating system. See list of operating systems.

Usually, a disk operating system was loaded from a disk. Only a very few comparable DOSes were stored elsewhere than floppy disks; among these exceptions were the British BBC Micro's optional Disc Filing System, DFS, offered as a kit with a disk controller chip, a ROM chip, and a handful of logic chips, to be installed inside the computer; and Commodore's CBM DOS, located in a ROM chip in each disk drive.

3.1 Brief History of MS-DOS

The history of Microsoft disk operating system (MS-DOS) is closely linked to the IBM PC and compatibles. Towards the end of the 1970's, a number of PCs appeared on the market, based on 8-bit microprocessor chips such as Intel 8080. IBM decided to enter this market and wisely opted for a 16-bit microprocessor, the Intel 8088. IBM wanted to introduced the PC to the market as quickly as possible and released it without having enough time to develop its own OS.

At that time, CP/M (by Digital Research) dominated the market. In 1979, a small company Seattle the Computer Products developed its own OS, 86-DOS to test some of its Intel based products (86-DOS was designed to be similar to CP/M). IBM purchased 86-DOS and in collaboration with Microsoft developed a commercial product. MS-DOS Version 1.0 was also referred to as PC-DOS MS-DOS had some similarities to CP/M, (such as the one level file storage system for floppy disks) which was important in terms of market acceptance in those days although MS-DOS did offer several improvements over CP/M such as:

- A larger disk sector size (512 bytes as opposed to 128 bytes)
- A memory-based file allocation table.

Both of which improved disk file performance.

Here is a summary of the most significant features of versions of MS-DOS:

Version	Date	Features
1.0	1981	• Based on IBM PC • Designed to cater for floppy disks and therefore used a simple file storage system, simlar to CP/M. • A memory-based file allocation table • A larger disk sector 512 bytes
2.0	1983	• Based on IBM PC/XT with 10MB hard disk • A hierarchical file directory to simplify the vast storage • Installable device drivers.
3.0	1984	• Based on IBM PC/AT with 20MB hard disk • Supported RAM disks • Read-only files
3.1	1984	• Some support for networks (file sharing, locking, etc.)
3.2	1986	• 3.5 inch disks • Support for IBM Token Ring Network

3.3	1987	• Support for new IBM PS/2 computers • 1.44 MB floppies • Multiple 32MB disk partitions • Support for Expanded Memory system
4.0	1988	• Simple window-based command shell • Up to 2 gigabyte disk partitions
5.0	1991	• Improved memory management • Doubling the disk space by compressing files for floppies and hard disks • Interlink a program that transfers files between computers • Improved data protection facility • Antivirus facility that can remove more than 800 viruses from your system. • Improved extended commands.

3.2 Examples of disk operating systems that were extensions to the OS

- The DOS operating system for the Apple Computer's Apple II family of computers. This was the primary operating system for this family from 1979 with the introduction of the floppy disk drive until 1983 with the introduction of ProDOS; many people continued using it long after that date. Usually it was called Apple DOS to distinguish it from MS-DOS.
- Commodore DOS, which was used by 8-bit Commodore computers. Unlike most other DOS systems, it was integrated into the disk drives, not loaded into the computer's own memory.
- Atari DOS: which was used by the Atari 8-bit family of computers. The Atari OS only offered low-level disk-access, so an extra layer called DOS was booted off a floppy that offered higher level functions such as filesystems.
- MSX-DOS, for the MSX computer standard. Initial version, released in 1984, was nothing but MS-DOS 1.0 ported to Z80; but in 1988 it evolved to version 2, offering facilities such as subdirectories, memory management and environment strings. The MSX-DOS kernel resided in ROM (built-in on the disk controller) so basic file access capacity was available even without the command interpreter, by using BASIC extended commands.
- Disc Filing System (DFS) This was an optional component for the BBC Micro, offered as a kit with a disk controller chip, a ROM chip, and a handful of logic chips, to be installed inside the computer. See also Advanced Disc Filing System.
- AMSDOS, for the Amstrad CPC computers.

- GDOS and G+DOS, for the +D and DISCiPLE disk interfaces for the ZX Spectrum.

3.2 Examples of Disk Operating Systems that were the OS itself

- The DOS/360 initial/simple operating system for the IBM System/360 family of mainframe computers (it later became DOS/VSE, and was eventually just called VSE).
- The DOS operating system for DEC PDP-11 minicomputers (this OS and the computers it ran on were nearly obsolete by the time PCs became common, with various descendants and other replacements).
- DOS for the IBM PC compatible platform

The best known family of operating systems named "DOS" is that running on IBM PCs type hardware using the Intel CPUs or their compatible cousins from other makers. Any DOS in this family is usually just referred to as DOS. The original was licensed to IBM by Microsoft, and marketed by them as "PC-DOS". When Microsoft licenced it to other hardware manufacturers, it was called MS-DOS. Digital Research produced a compatible variant known as "DR-DOS", which was eventually taken over (after a buyout of Digital Research) by Novell. This became "OpenDOS" for a while after the relevant division of Novell was sold to Caldera International, now called SCO. There is also a free version named "FreeDOS".

4.0 Conclusion

The earliest operating systems were developed for mainframe computer architectures in the 1960s and they were mostly batch processing operating systems. The enormous investment in software for these systems caused most of the original computer manufacturers to continue to develop hardware and operating systems that are compatible with those early operating systems. Those early systems pioneered many of the features of modern operating systems.

5.0 Summary

This unit has taken you through some of the various sample of early OS. The DOS is no longer popular because of the advanced GUI packages in the market now. But you still find it in some computing environment.

6.0 Tutor Marked Assignment

You are to attempt the following assignments and submit your answers to your tutor for this course. Here we go:

1. Disk operating system can be the operating system itself or not . Discuss.

2. Distinguish between DOS that is the OS itself and the one that is not .

3. Give two examples each of DOS that are the OS itself and DOS that are the extension of the OS.

7.0 References/Further Reading

1. Deitel, Harvey M.; Deitel, Paul; Choffnes, David (2004). *Operating Systems*. Upper Saddle River, NJ: Pearson/Prentice Hall. ISBN 0-13-182827-4.
2. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5.
3. Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems. Design and Implementation*. Upper Saddle River, N.J.: Pearson/Prentice Hall. ISBN 0-13-142938-8.
4. Tanenbaum, Andrew S. (2001). *Modern Operating Systems*. Upper Saddle River, N.J.: Prentice Hall. ISBN 0-13-092641-8.

Module 2: Types of Operating System

Unit 2: Real-time operating system

Table of Contents

1.0 Introduction

The previous unit discussed the disk operating system and its various characteristics and examples. In this unit you will be exposed to the real-time operating system (RTOS), its design philosophies and some of its other characteristic features like scheduling, interrupt handling, etc.

2.0 Objectives

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

- Define and describe the real-time OS
- Explain its design philosophies
- Describe how its handles tasks such as memory allocation, scheduling, interrupt handling, intertask communication, etc.
- State how it is different from the disk OS
- Give examples of RTOS

3.0 **Real-Time Operating System** (**RTOS**)

A **real-time operating system** (**RTOS**) is a multitasking operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers, mobile telephones), industrial robots, spacecraft, industrial control (see SCADA), and scientific research equipment.

An RTOS facilitates the creation of a real-time system, but does not guarantee the final result will be real-time; this requires correct development of the software. An RTOS does not necessarily have high throughput; rather, an RTOS provides facilities which, if used properly, guarantee deadlines can be met generally (soft real-time) or deterministically (hard real-time). An RTOS will typically use specialized scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behavior in the final system. An RTOS is valued more for how quickly and/or predictably it can respond to a particular event than for the given amount of work it can perform over time. Key factors in an RTOS are therefore at minimal interrupt latency and a minimal thread switching latency.

An early example of a large-scale real-time operating system was the so-called "control program" developed by American Airlines and IBM for the Sabre Airline Reservations System.

Debate exists about what actually constitutes real-time computing.

3.1 Design philosophies

Two basic designs exist:

- Event-driven (priority scheduling) designs switch tasks only when an event of higher priority needs service, called preemptive priority.
- Time-sharing designs switch tasks on a clock interrupt, and on events, called round-robin.

Time-sharing designs switch tasks more often than is strictly needed, but give smoother, more deterministic multitasking, the illusion that a process or user has sole use of a machine.

Early CPU designs needed many cycles to switch tasks, during which the CPU could do nothing useful. So early OSes tried to minimize wasting CPU time by maximally avoiding unnecessary task-switches.

More recent CPUs take far less time to switch from one task to another; the extreme case is *barrel processors* that switch from one task to the next in zero cycles. Newer RTOSes almost invariably implement time-sharing scheduling with priority driven pre-emptive scheduling.

3.2 Scheduling

In typical designs, a task has three states:

1) Running

2) Ready

3) Blocked.

Most tasks are blocked, most of the time. Only one task per CPU is running. In simpler systems, the ready list is usually short, two or three tasks at most.

The real key is designing the scheduler. Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled. But, the choice of data structure depends also on the maximum number of tasks that can be on the ready list (or ready queue).

If there are never more than a few tasks on the ready list, then a simple unsorted bidirectional linked list of ready tasks is likely optimal. If the ready list usually contains only a few tasks but occasionally contains more, then the list should be sorted by priority, so that finding the highest priority task to run does not require iterating through the entire list. Inserting a task then requires walking the ready list until reaching either the end of the list, or a task of lower priority than that of the task being inserted. Care must be taken not to inhibit preemption during this entire search; the otherwise-long critical section should probably be divided into small pieces, so that if, during the insertion of a low priority task, an interrupt occurs that makes a high priority task ready, that high priority task can be inserted and run immediately (before the low priority task is inserted).

The critical response time, sometimes called the **flyback time**, is the time it takes to queue a new ready task and restore the state of the highest priority task. In a well-designed RTOS, readying a new task will take 3-20 instructions per ready queue entry, and restoration of the highest-priority ready task will take 5-30 instructions. On a 20MHz 68000 processor, task switch times run about 20 microseconds with two tasks ready. 100 MHz ARM CPUs switch in a few microseconds.

In more advanced real-time systems, real-time tasks share computing resources with many non-real-time tasks, and the ready list can be arbitrarily long. In such systems, a scheduler ready list implemented as a linked list would be inadequate.

3.3 Intertask communication and resource sharing

A significant problem that multitasking systems must address is sharing data and hardware resources among multiple tasks. It is usually "unsafe" for two tasks to access the same specific data or hardware resource simultaneously. ("Unsafe" means the results are inconsistent or unpredictable, particularly when one task is in the midst of changing a data collection. The view by another task is best done either before any change begins, or after changes are completely finished.) There are three common approaches to resolve this problem:

- Temporarily masking/disabling interrupts

- Binary semaphores
- Message passing

General-purpose operating systems usually do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wished. Modern CPUs make the interrupt disable control bit (or instruction) inaccessible in user mode to allow operating systems to prevent user tasks from doing this. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater system call efficiency and also to permit the application to have greater control of the operating environment without requiring OS intervention.

On single-processor systems, if the application runs in kernel mode and can mask interrupts, often that is the best (lowest overhead) solution to preventing simultaneous access to a shared resource. While interrupts are masked, the current task has *exclusive* use of the CPU; no other task or interrupt can take control, so the critical section is effectively protected. When the task exits its critical section, it must unmask interrupts; pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the longest path through the critical section is shorter than the desired maximum interrupt latency, or else this method will increase the system's maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few source code lines long and contains no loops. This method is ideal for protecting hardware bitmapped registers when the bits are controlled by different tasks.

When the critical section is longer than a few source code lines or involves lengthy looping, an embedded/real-time programmer must resort to using mechanisms identical or similar to those available on general-purpose operating systems, such as semaphores and OS-supervised interprocess messaging. Such mechanisms involve system calls, and usually invoke the OS's dispatcher code on exit, so they can take many hundreds of CPU instructions to execute, while masking interrupts may take as few as three instructions on some processors. But for longer critical sections, there may be no choice; interrupts cannot be masked for long periods without increasing the system's interrupt latency.

A binary **semaphore** is either locked or unlocked. When it is locked, a queue of tasks can wait for the semaphore. Typically a task can set a timeout on its wait for a semaphore. Problems with semaphore based designs are well known: priority inversion and deadlocks.

In **priority inversion**, a high priority task waits because a low priority task has a semaphore. A typical solution is to have the task that has a semaphore run at (inherit) the priority of the highest waiting task. But this simplistic approach fails when there are multiple levels of waiting (A waits for a binary semaphore locked by B, which waits for a binary semaphore locked by C). Handling multiple levels of inheritance without introducing instability in cycles is not straightforward.

In a **deadlock**, two or more tasks lock a number of binary semaphores and then wait forever (no timeout) for other binary semaphores, creating a cyclic dependency graph.

The simplest deadlock scenario occurs when two tasks lock two semaphores in lockstep, but in the opposite order. Deadlock is usually prevented by careful design, or by having floored semaphores (which pass control of a semaphore to the higher priority task on defined conditions).

The other approach to resource sharing is for tasks to send **messages**. In this paradigm, the resource is managed directly by only one task; when another task wants to interrogate or manipulate the resource, it sends a message to the managing task. This paradigm suffers from similar problems as binary semaphores: Priority inversion occurs when a task is working on a low-priority message, and ignores a higher-priority message (or a message originating indirectly from a high priority task) in its in-box. Protocol deadlocks occur when two or more tasks wait for each other to send response messages.

Although their real-time behavior is less crisp than semaphore systems, simple message-based systems usually do not have protocol deadlock hazards, and are generally better-behaved than semaphore systems.

3.4 Interrupt handlers and the scheduler

Since an interrupt handler blocks the highest priority task from running, and since real time operating systems are designed to keep thread latency to a minimum, interrupt handlers are typically kept as short as possible. The interrupt handler defers all interaction with the hardware as long as possible; typically all that is necessary is to acknowledge or disable the interrupt (so that it wo not occur again when the interrupt handler returns). The interrupt handler then queues work to be done at a lower priority level, often by unblocking a driver task (through releasing a semaphore or sending a message). The scheduler often provides the ability to unblock a task from interrupt handler

3.5 Memory allocation

Memory allocation is even more critical in an RTOS than in other operating systems.

Firstly, speed of allocation is important. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable free memory block; however, this is unacceptable as memory allocation has to occur in a fixed time in an RTOS.

Secondly, memory can become fragmented as free regions become separated by regions that are in use. This can cause a program to stall, unable to get memory, even though there is theoretically enough available. Memory allocation algorithms that slowly accumulate fragmentation may work fine for desktop machines—when rebooted every month or so—but are unacceptable for embedded systems that often run for years without rebooting.

The simple fixed-size-blocks algorithm works astonishingly well for simple embedded systems

4.0 Conclusion

As you would have seen or discovered while reading through this unit the design of the RTOS is more complex and involving than that of the DOS. The RTOS is not found in most computer environment because of its complexity. But it is most suitable for use in life and time critical environment.

5.0 Summary

This unit has extensively describe and discussed the RTOS and the way it carries out various OS functions such as memory allocation, scheduling, interrupt handling and interprocess communication (You will learn more about some of these functions of the OS in later units in these course). You are to compare all these with what you have learnt about DOS in the previous unit.

6.0 Tutor Marked Assignment

You are to attempt the following assignments and submit your answers to your tutor for this course. Here we go:

1. Memory allocation is even more critical in an RTOS than in other operating systems. Discuss
2. Name some of the environment in which the RTOS can be found
3. List and explain the two basic design philosophies for the RTOS
4. Describe how interprocess communication and resource sharing are implemented in the RTOS.

7.0 Further Reading

1. Deitel, Harvey M.; Deitel, Paul; Choffnes, David (2004). *Operating Systems*. Upper Saddle River, NJ: Pearson/Prentice Hall. ISBN 0-13-182827-4.
2. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5.
3. Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems. Design and Implementation*. Upper Saddle River, N.J.: Pearson/Prentice Hall. ISBN 0-13-142938-8.
4. Tanenbaum, Andrew S. (2001). *Modern Operating Systems*. Upper Saddle River, N.J.: Prentice Hall. ISBN 0-13-092641-8.

Module 2: Types of Operating System

Unit 3: Object-oriented and Time sharing Operating Systems

Table of Contents

1.0 Introduction

In the previous unit, you been exposed to the real-time operating system, its history and application areas. This unit will extensively discuss the time-sharing operating system as well as the Object-oriented operating systems and the various attempts (citing examples where necessary) that had been made to develop them.

2.0 Objectives

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

- Explain what is meant by object-oriented OS
- Compare with examples the various attempts that had been made to develop an object-oriented OS

3.0 Main Body

3.1 Object-oriented operating system

An **object-oriented operating system** is an operating system which internally uses object-oriented methodologies.

An object-oriented operating system is in contrast to an object-oriented user interface or programming framework, which can be placed above a non-object-oriented operating system like DOS, Microsoft Windows or Unix.

It can be argued, however, that there are already object-oriented concepts involved in the design of a more typical operating system such as Unix. While a more traditional language like C does not support object orientation as fluidly as more recent languages, the not ion, for example, of a file, stream, or device driver (in Unix, each represented as a file descriptor) can be considered a good example of object orientation: they are, after all, abstract data types, with various methods in the form of system calls, whose behavior varies based on the type of object, whose implementation details are hidden from the caller, and might even use inheritance in their underlying code.

3.1.1 Examples

3.1.1.1 NeXTSTEP

During the late 1980s, Steve Jobs formed the computer company NeXT. One of NeXT's first tasks was to design an object-oriented operating system, NEXTSTEP. They did this by adding an object-oriented framework on top of Mach and BSD using the Objective-C language as a basis.

NEXTSTEP's basis, Mach and BSD, are not object-oriented. Instead, the object-oriented portions of the system live in userland. Thus, NEXTSTEP cannot be considered an object-oriented operating system in the strictest terms.

The NeXT hardware and operating system were not successful, and, in search of a new strategy, the company re-branded its object-oriented technology as a cross-platform development platform.

Though NeXT's efforts were innovative and novel, they gained only a relatively small acceptance in the marketplace. NeXT was later acquired by Apple Computer and its operating system became the basis for Mac OS X most visibly in the form of the "Cocoa" frameworks.

3.1.1.2 Choices

Choices is an object-oriented operating system that was developed at the University of Illinois at Urbana-Champaign. It is written in C++ and uses objects to represent core kernel components like the CPU, Process and so on. Inheritance is used to separate the kernel into portable machine independent classes and small non-portable dependent classes. Choices has been ported to and runs on SPARC, x86 and ARM.

3.1.1.3 Athene

Athene is an object based operating system first released in 2000 by Rocklyte Systems. The user environment is constructed entirely from objects that are linked together at runtime. Applications for Athene can also be created using this methodology and are commonly scripted using the object scripting language 'DML' (Dynamic Markup Language). Objects can be shared between processes by creating them in shared memory and locking them as required for access. Athene's object framework is multi-platform, allowing it to be used in Windows and Linux environments for the development of object oriented programs.

3.1.1.4 BeOS

One attempt at creating a truly object-oriented operating system was the BeOS of the mid 1990s, which used objects and the C++ language for the application programming interface (API). But the kernel itself was written in C with C++ wrappers in user space. The system did not become mainstream though even today it has its fans and benefits from ongoing development.

3.1.1.5 Syllable

Syllable makes heavy use of C++ and for that reason is often compared to BeOS.

3.1.1.6 TAJ

TAJ is India's first object oriented operating system. It is made in C++ with some part in assembly. The source code of TAJ OS is highly modularized and is divided into different modules, each module is implemented as class. Many object oriented features like inheritance, polymorphism, virtual functions etc are extensively used in developing TAJ Operating System. TAJ OS is a multitasking, multithreading and multiuser operating system.

The kernel of TAJ Operating System is of monolithic type. i.e. all the device drivers and other important OS modules are embedded into kernel itself. This increases the speed of execution by reducing context switching time (time taken to execute a system call).

TAJ OS is developed by Viral Patel. You can download the image file for TAJ OS at http://www.viralpatel.net or http://www.geocities.com/taj_os

Features of TAJ Operating System:

- 32-bit Protected mode Operating System
- Paging enable
- Secure Exception handling
- Interrupt management system
- Work with different kinds of CPU (80386 onwards).
- Fully functional built in keyboard driver
- Total DMA control
- Floppy driver
- Mouse driver
- Fat file system driver

- Multitasking
- Multithreading
- Multiuser

3.1.1.7 Java-based operating systems

Given that Sun Microsystems' Java is today one of the most dominant object-oriented languages, it is no surprise that Java-based operating systems have been attempted. In this area, ideally, the kernel would consist of the bare minimum required to support a JVM. This is the only component of such an operating system that would have to be written in a language other than Java. Built upon that JVM and basic hardware support, it would be possible to write the rest of the operating system in Java; even parts of the system that are more traditionally written in a lower-level language such as C, for example device drivers, can be written in Java.

Examples of attempts at such an operating system include JNode and JOS

3.2 Time-sharing

Time-sharing refers to sharing a computing resource among many users by multitasking.

Because early mainframes and minicomputers were extremely expensive, it was rarely possible to allow a single user exclusive access to the machine for interactive use. But because computers in interactive use often spend much of their time idly waiting for user input, it was suggested that multiple users could share a machine by using one user's idle time to service other users. Similarly, small slices of time spent waiting for disk, tape, or network input could be granted to other users.

Throughout the late 1960s and the 1970s computer terminals were multiplexed onto large institutional mainframe computers (central computer systems), which in many implementations sequentially polled the terminals to see if there was any additional data or action requested by the computer user. Later technology in interconnections were interrupt driven, and some of these used parallel data transfer technologies like, for example, the IEEE 488 standard. Generally, computer terminals were utilized on College properties in much the same places as *desktop computers* or *personal computers* are found today. In the earliest days of personal computers, many were in fact used as particularly smart terminals for time-sharing systems.

With the rise of microcomputing in the early 1980's, time-sharing faded into the background because the individual microprocessors were sufficiently inexpensive that a single person could have all the CPU time dedicated solely to their needs, even when idle.

The Internet has brought the general concept of time-sharing back into popularity. Expensive corporate server farms costing millions can host thousands of customers all sharing the same common resources. As with the early serial terminals, websites operate primarily in bursts of activity followed by periods of idle time. The bursty nature permits the service to be used by many website customers at once, and none of them not ice any delays in communications until the servers start to get very busy.

3.2.1 The Time-Sharing Business

In the 1960s, several companies started providing time-sharing services as service bureaus. Early systems used Teletype K/ASR-33s or K/ASR-35s in ASCII environments, and an IBM teleprinter in EBCDIC environments. They would connect to the central computer by dial-up acoustically coupled modems operating at 10-15 characters per second. Later terminals and modems supported 30-120 characters per second. The time-sharing system would provide a complete operating environment, including a variety of programming language processors, various software packages, file storage, bulk printing, and off-line storage. Users were charged rent for the terminal, a charge for hours of connect time, a charge for seconds of CPU time, and a charge for kilobyte-months of disk storage.

Common systems used for time-sharing included the SDS 940, the PDP-10, and the IBM 360. Companies providing this service included Tymshare (founded in 1966), Dial Data (bought by Tymshare in 1968), and Bolt, Beranek, and Newman. By 1968, there were 32 such service bureaus serving the NIH alone.

3.2.2 History

The concept was first described publicly in early 1957 by Bob Bemer as part of an article in *Automatic Control Magazine*. The first project to implement a time-sharing system was initiated by John McCarthy in late 1957, on a modified IBM 704, and later an additionally modified IBM 7090 computer. Although he left to work on Project MAC and other projects, one of the results of the project, known as the *Compatible Time Sharing System* or CTSS, was demonstrated in November, 1961. CTSS has a good claim to be the first time-sharing system and remained in use until 1973. The first commercially successful time-sharing system was the *Dartmouth Time-Sharing System* (DTSS) which was first implemented at Dartmouth College in 1964 and subsequently formed the basis of General Electric's computer bureau services. DTSS influenced the design of other early timesharing systems developed by Hewlett Packard, Control Data Corporation, UNIVAC and others (in addition to introducing the BASIC programming language).

Other historical timesharing systems, some of them still in widespread use, include:

- IBM CMS (part of VM/CMS)
- IBM TSS/360 (never finished; see OS/360)
- IBM Time Sharing Option (TSO)
- KRONOS (and later NOS) on the CDC 6000 series
- Michigan Terminal System
- Multics
- MUSIC/SP
- ORVYL
- RSTS/E

- RSX-11
- TENEX
- TOPS-10
- TOPS-20

4.0 Conclusion

As you have learnt in this unit, several attempts had been made at developing a truly object-oriented operating system. The unit has also discussed the basic features of some the object-oriented OS in existence today.

5.0 Summary

This unit has briefly discussed the time-sharing operating system as well as object-oriented OS and the various attempts that have been made to develop them. In the next unit you will be learning about some of the basic functions of the OS and how they are achieved.

6.0 Tutor Marked Assignment

You are to attempt the following assignments and submit your answers to your tutor for this course. Here we go:

1. What do you under by object-oriented OS
2. Discuss at least three of the attempts that have been made to develop object-oriented OS stating the characteristic features of each of the examples of these attempts.

7.0 References/Further Reading

1. Deitel, Harvey M.; Deitel, Paul; Choffnes, David (2004). *Operating Systems*. Upper Saddle River, NJ: Pearson/Prentice Hall. ISBN 0-13-182827-4.
2. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5.
3. Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems. Design and Implementation*. Upper Saddle River, N.J.: Pearson/Prentice Hall. ISBN 0-13-142938-8.
4. Tanenbaum, Andrew S. (2001). *Modern Operating Systems*. Upper Saddle River, N.J.: Prentice Hall. ISBN 0-13-092641-8.

Module 3: Process Management

Unit 1: Processes

Table of Contents

1.0 Introduction

Early computer systems allowed one program to be executed at a time. This program has complete control of the system, and had access to all the system's resources. Current-day computer systems allow multiple programs to be loaded into memory and to be executed concurrently. This evolution requires firmer control and more compartmentalization of the various programs. These needs resulted in the not ion of a process, which is a program in execution. A process is the unit of work in a modern time-sharing system.

Although, the main concern of the OS is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: Operating system processes executing system code, and user processes executing user code. All these processes can potentially execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive.

2.0 Objectives

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

- Define a process
- List the possible states of a process
- Describe a process control block (PCB)
- Describe process creation and process termination

3.0 Main Body

3.1 The Process

Informally, a **process** is a program in execution or simply an instance of a computer program that is being executed. It is more than the program code, which is sometimes called **text section**. While a program itself is just a passive collection of instructions, a process is the actual execution of those instructions. It includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, return addresses, and local variables), and a **data section**, which contains global variables.

You should note the emphasis that a program by itself is not a process; a program is a *passive* entity such as the content of the file stored on disk, whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources.

Several processes may be associated with the same program - each would execute independently (multithreading - where each thread represents a process), either synchronously (sequentially) or asynchronously (in *parallel*). Although, two processes may be associated with the same program, they nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the editor program. Each of these is a separate process, and although the text sections are equivalent, the data sections vary. It is also common to have a process that spawns many processes as it runs.

Modern computer systems allow multiple programs and processes to be loaded into memory at the same time and, through time-sharing (or multitasking), give an appearance that they are being executed at the same time (*concurrently*) even if there is just one processor. Similarly, using a multithreading OS and/or computer architecture, *parallel* processes of the same program may actually execute *simultaneously* (on different CPUs) on a multiple CPU machine or network.

In general, a computer system process consists of (or is said to 'own') the following resources:

- An *image* of the executable machine code associated with a program.
- Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep

track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during runtime.

- Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers, physical memory addressing, etc. The *state* is typically stored in computer registers when the process is executing, and in memory otherwise.

3.2 Process states

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur (such as I/O completion or reception of a signal)
- **Ready:** the process is waiting to be assigned to a processor.
- **Terminated:** The processor has finished execution.

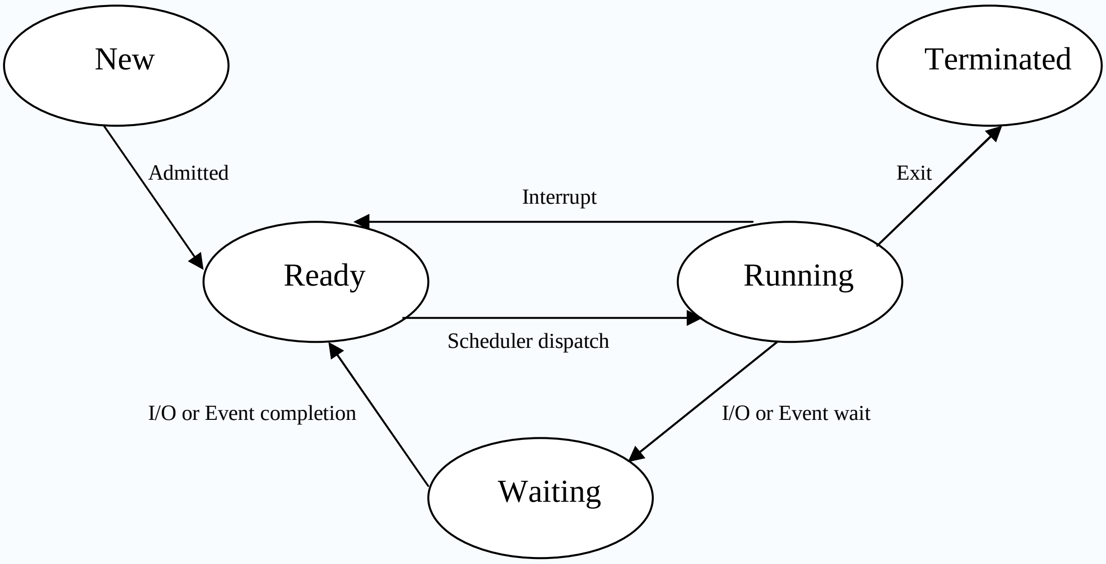


Figure 3.1: Process State

These state names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems more finely delineate process states. Only one process can be *running* on any processor at any instant, although many processes may be *ready* and *waiting*. The state diagram corresponding to these states is presented in Figure 3.1. The various process states, are displayed in the figure, with arrows indicating possible transitions between states.

3.3 Process Control Block (PCB)

Each process is represented in the operating system by a process control block (PCB) – also called a task control block. A PCB contains many pieces of information associated with a specific process as shown in Figure 3.2 below.

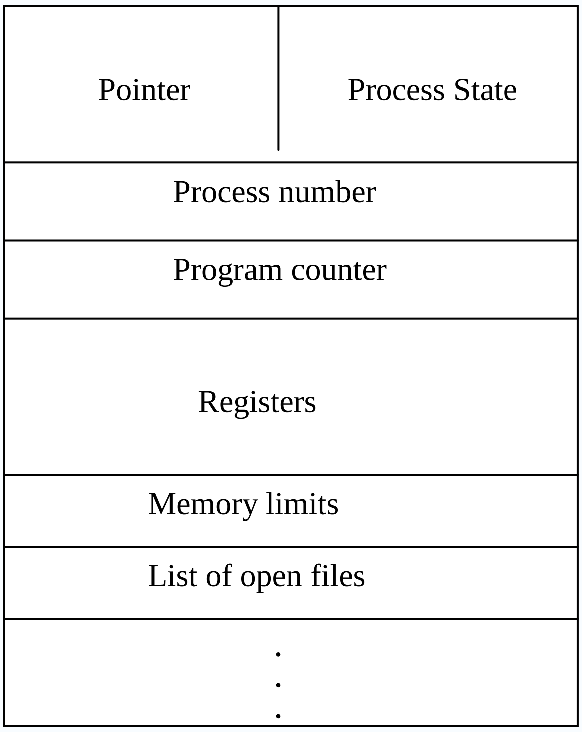


Figure 3.2: Process Control Block (PCB)

The content of the PCB include:

- **Process State:** As you have learnt in the previous section, the state may be new, ready, running, waiting, halted, etc.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** the register vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward. (Figure 3.3)
- **CPU-Scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** this information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** this information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, etc.

- **I/O status information:** the information includes the list of I/O devices allocated to this process, a list of open files, etc.

The PCB simply serves as the repository for any information that may vary from process to process.

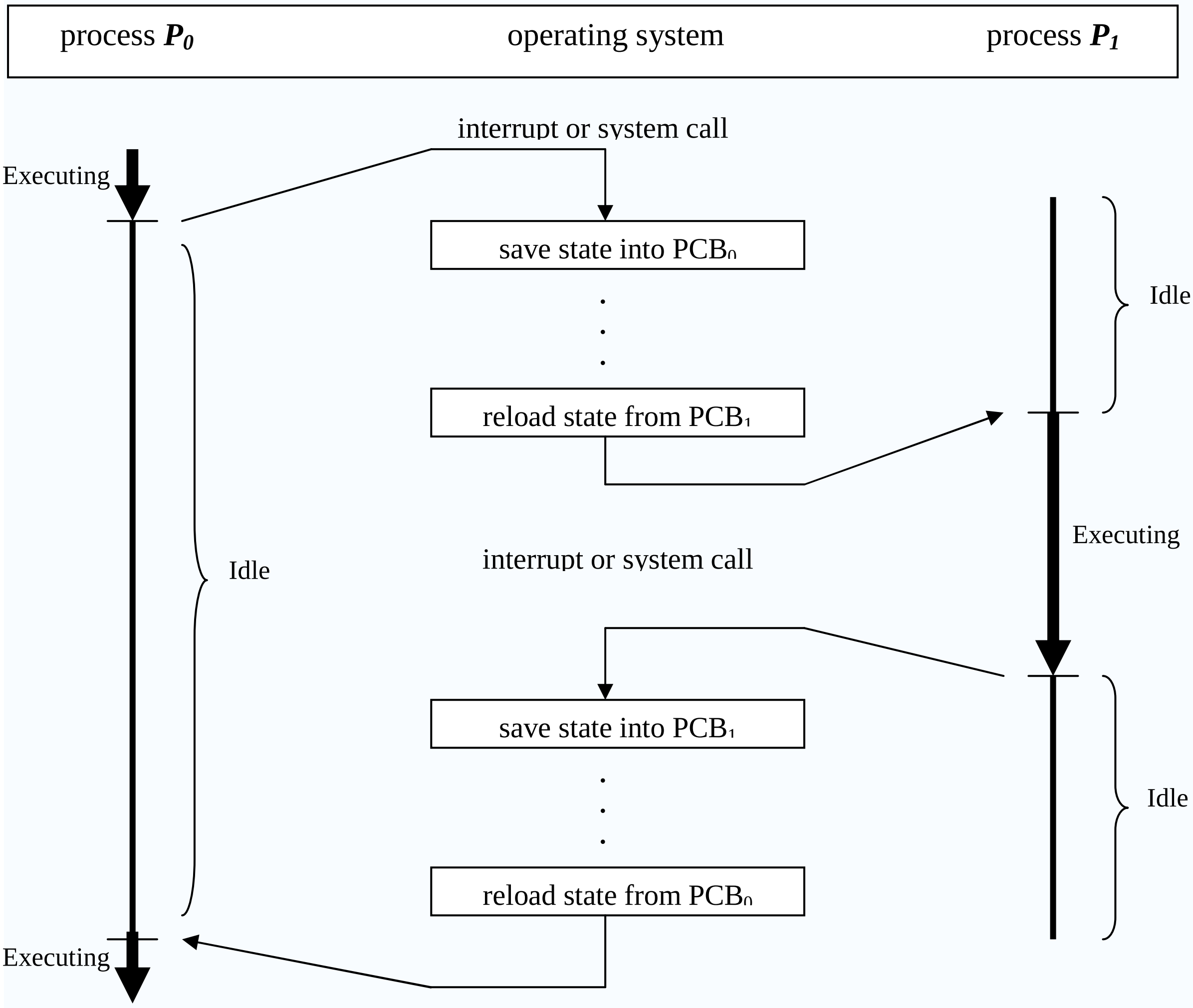


Figure 3.3: Diagram showing CPU switch from process to process

3.4 Process Scheduling

The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. A uniprocessor system can have only one running process. If more processes exist, the rest must wait until the CPU is free and can be rescheduled.

3.4.1 Scheduling Queues

As processes enter the system, they are put into a **job queue**. This queue consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCB in the list. We extend each PCB to include a pointer field that points to the next PCB in the ready queue.

The operating system also has other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. In the case of I/O request, such a request may be to a dedicated tape drive, or to a shared device, such as a disk. Since the system has many processes, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own queue.

A common way of representating process scheduling is by using a **queueing** diagram, such as that in Figure 3.4. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of several events could occur:

- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new sub process and wait for its termination.
- The process could be removed forcibly from the CPU, as result of an interrupt, and be put back in the ready queue.
- In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

3.4.2 Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes for these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

In a batch system, often more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device, where they a re kept for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute, and allocates the CPU to one of them.

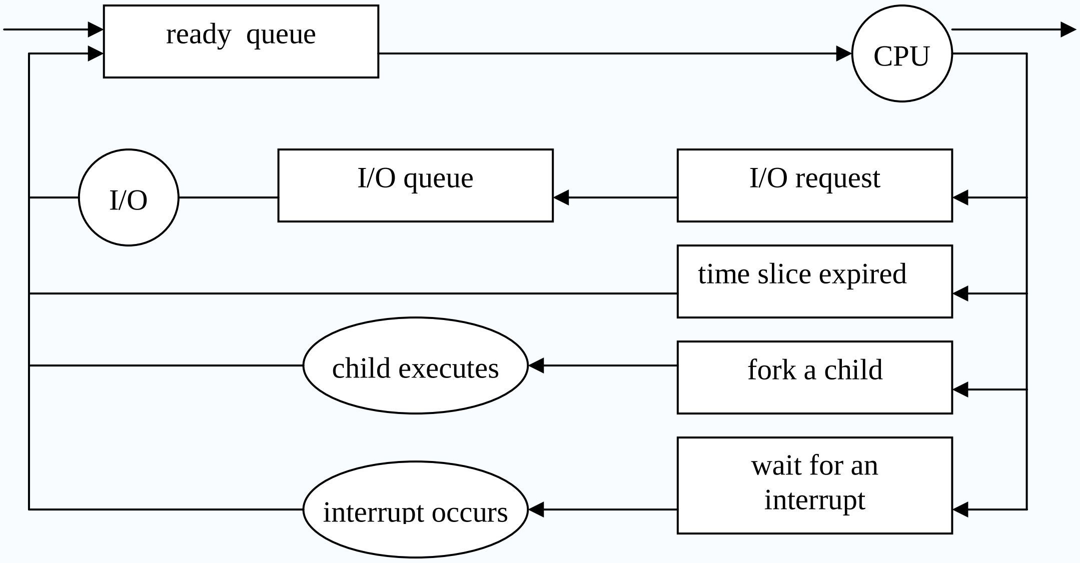


Figure 3.4: Queuing-diagram representation of process scheduling

The primary distinction between these two schedulers is the frequency of their execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Due to the brief time between executions, the short-term scheduler must be fast.

The long-term scheduler, on the other hand, executes much less frequently. There may be minutes between the creation of new processes in the system. The long-term scheduler controls the degree of multiprogramming - the number of processes in memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average rate of processes leaving the system.. therefore, the long-term scheduler may need to be invoked only when process leaves the system.

3.4.3 Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as context switch. The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state (Figure 3.1), and memory-management information. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions. Typical speeds range from 1 to

1000 micro seconds. Also, context-switch times are highly dependent on h/which support.

3.5 Operations on Processes

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Therefore, the operating system must provide a mechanism (or facility) for process creation and termination.

3.5.1 Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, whereas the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a tree of processes (Figure 3.5).

In general, a process will need certain resources (such as CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

When a process is created it obtains initialization data (or input) that may be passed along from the parent process to the child process in addition to the various physical and logical resources. For instance, consider a process whose function is to display the status of a file, say F_1, on the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file F_1, and it will execute using that datum to obtain the desired information. It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, F_1 and the terminal device, and may just need to transfer the datum between the two.

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

In UNIX, every process except process 0 (the swapper) is created when another process executes the fork system call. The process that invoked fork is the *parent process* and the newly-created process is the *child process*. Every process (except process 0) has one parent process, but can have many child processes.

In UNIX, a child process is in fact created (using fork) as a copy of the parent. The child process can then overlay itself with a different program (using `exec`) as required.

Each process may create many child processes but will have only one parent process, except for the very first process which has no parent. The first process, called **init** in UNIX, is started by the kernel at booting time and never terminates.

The kernel identifies each process by its process identifier (PID). Process 0 is a special process that is created when the system boots; after forking a child process (process 1), process 0 becomes the swapper process. Process 1, known as `init`, is the ancestor of every other process in the system.

When a child process terminates execution, either by calling the *exit* system call, causing a fatal execution error, or receiving a terminating signal, an exit status is returned to the operating system. The parent process is informed of its child's termination through a SIGCHLD signal. A parent will typically retrieve its child's exit status by calling the *wait* system call. However, if a parent does not do so, the child process becomes a **zombie** process.

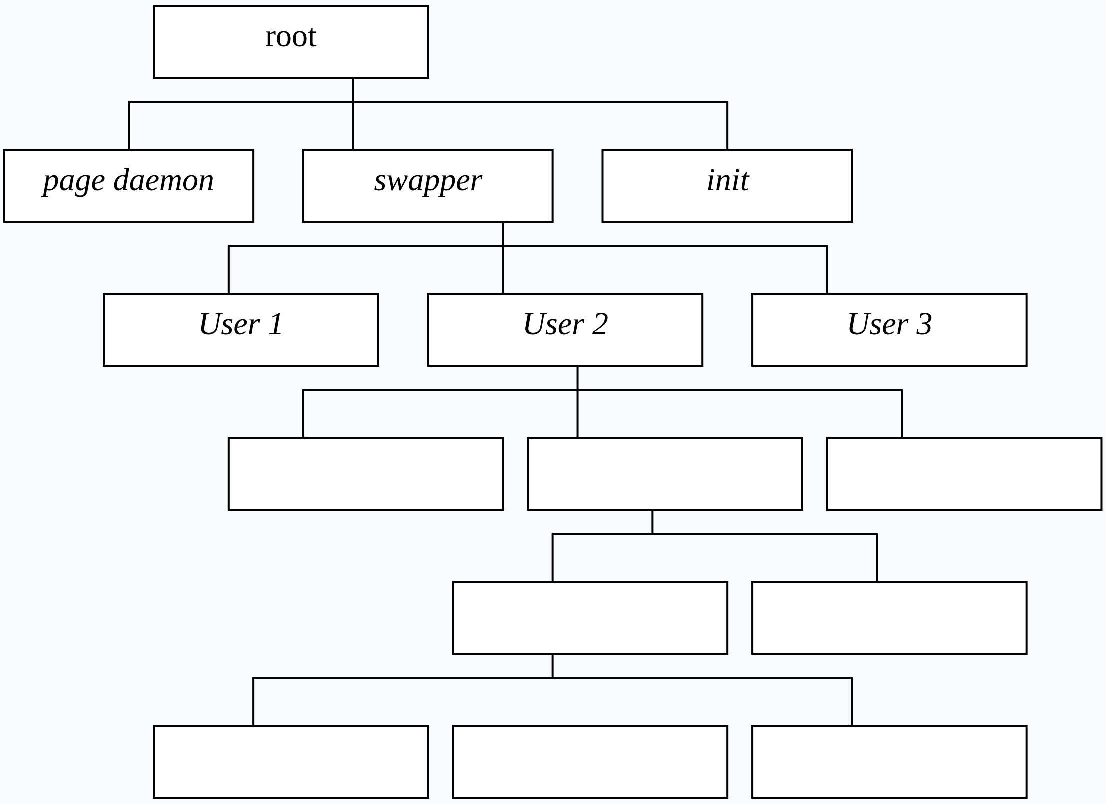


Figure 3.5: A tree on a typical UNIX system

On Unix and Unix-like operating systems, a **zombie process** or **defunct process** is a process that has completed execution but still has an entry in the process table, this entry being still needed to allow the process that started the zombie process to read its exit status.

Example

Here is some sample C programming language code to illustrate the idea of forking. The code that is in the "Child process" and "Parent process" sections are executed simultaneously.

```
pid_t pid;

pid = fork();

if(pid == 0)
{
   /* Child process:
    * When fork() returns 0, we are in
    * the child process.
    * Here we count up to ten, one each second.
    */
   int j;
   for(j=0; j < 10; j++)
   {
      printf("child: %d\n", j);
      sleep(1);
   }
   _exit(0); /* Note that we do not  use exit() */
}
else if(pid > 0)
{
   /* Parent process:
    * Otherwise, we are in the parent process.
    * Again we count up to ten.
    */
   int i;
   for(i=0; i < 10; i++)
   {
      printf("parent: %d\n", i);
      sleep(1);
   }
}
else
{
   /* Error handling. */
   fprintf(stderr, "could not  fork");
```

```
      exit(1);
   }
```

This code will print out the following:

```
parent: 0
child: 0
child: 1
parent: 1
parent: 2
child: 2
child: 3
parent: 3
parent: 4
child: 4
child: 5
parent: 5
parent: 6
child: 6
child: 7
parent: 7
parent: 8
child: 8
child: 9
parent: 9
```

Figure 3.6: C program forking a separate process

The order of each output is determined by the kernel.

Windows NT operating system supports both models: The parent's address space may be duplicated, or the parent may specify the name of a program for the operating system to load into the address space of the new process.

3.5.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using exit system call. At that point, the process may return data (output) 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.

Termination occurs under additional circumstances. A process can cause the termination of another process via an appropriate system call e.g. abort. Usually, only the parent of the process that is to be terminated can invoke such a system call. Otherwise, users could arbitrarily kill each other's jobs. A parent, therefore, need to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

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. This requires the parent to 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 allow a child to continue if its parent terminate. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

To illustrate process execution and termination, consider that in UNIX, we can terminate a process by using the exit system call; its parent process may wait for the termination of a child process by using wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. If the parent terminates, however, all its children have assigned as their new parent the init process. Thus, the children still have a parent to collect their status.

4.0 Conclusion

This unit has introduced you to the concept of processes. It has extensively discussed process control block, process scheduling and the various operations that can be carried out on processes.

In the next unit, you will be taken through co-operating processes and the means through which these co-operating processes communicate with one another.

5.0 Summary

As you have learnt in this unit, a process is a program in execution. As a process executes, it changes state. The state of a process is defined by that process current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated. Each process is represented in the operating system by its own process-control block (PCB).

A process, when it is not executing is placed in some waiting queue. The two major classes of queues in an operating system are I/O request queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB, and the PCB can be linked together to form a ready queue. Long-term (or job) scheduling is selection of processes to be allowed to contend for the CPU. Long-term scheduling is normally influenced by resource –

allocation considerations, especially memory management. Short-term (or CPU) scheduling is the selection of one process from the ready queue.

6.0 Tutor-Marked Assignment

1. What do you understand by the term ‘Process’?
2. Distinguish between a process and a program
3. List and explain the possible states of a process
4. Describe a process control block (PCB).
5. What are possible information that a PCB will contain?
6. Describe the actions taken by the kernel to switch context between processes.
7. Describe process creation and process termination.
8. What do you understand by process scheduling?
9. Explain the various types of scheduling we have.

7.0 References/Further Reading

Silberschatz, Abraham; Cagne, Greg, Galvin, Peter Baer (2004). *Operating system concepts with Java*, Sixth Edition, John Wiley & Sons, Inc.

Stallings, William (2005). *Operating Systems: internals and design principles (5th edition)*. Prentice Hall. ISBN 0-13-127837-1.

http://en.wikipedia.org/wiki/Process_states

Module 3: Process Management

Unit 2: Co-operating Processes

Table of Contents

1.0 Introduction

The previous unit has introduced you to the concept of processes and the various operations that can be carried out on processes. Sometimes, when you have more than one process running on the computer system, there may be need for them to interact with one another. This unit takes you through the different ways that these various processes that may be running on the computer system at the same time interacts with one another.

2.0 Objectives

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

- Describe the concept of co-operating processes
- State reasons for allowing process co-operation
- Explain interprocess communication
- Describe message passing
- Describe some methods for logically implementing a link and the send/receive operations
- Describe means of ensuring synchronization communicating processes
- Describe the concept of buffering and the various ways it can be implemented.

3.0 Main Body

3.1 Co-operating processes

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

We may want to provide an environment for process co-operation for several reasons:

- **Information sharing:** since several users may be interested in the same piece of information, e.g. 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 sub-tasks, each of which will be executing in parallel with the others. 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.
- **Convenience:** even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

Concurrent execution of co-operating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions.

To illustrate the concept of co-operating processes, let us consider the producer-consumer problem, which is a common paradigm for co-operating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a print program produces characters that are consumed by the printer driver.

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 consuming another item. The producer and consumer must be synchronised, so that it 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.

The unbounded-buffer producer-consumer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded-buffer producer-consumer problem assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

The buffer may either be provided by the operating system through the use of an interprocess-communication (IPC) facility (this will be discussed fully in the next section), or explicitly coded by the application programmer with the use of shared memory.

3.2 Interprocess Communication (IPC)

Processes can communicate with each other via Inter-process communication (IPC). This is possible for both processes running on the same machine and on different machines.

IPC provides a mechanism to allow processes to communicate and to synchronise 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 of this is a chat program used on the World Wide Web (WWW).

IPC is best provided by message-passing system and message-passing system can be defined in many ways. We are now going to look at different issues when designing message-passing systems.

3.2.1 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. On the other hand, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

If processes *P* and *Q* want to communicate, they must send messages to and receive message from each other, 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
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-sized or variable-sized messages

We are going to look at each of these types of messages in the following section.

3.2.2 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

3.2.2.1 Direct Communication

With 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 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 want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

This scheme exhibits symmetry in addressing; that is, both the sender and the receiver processes must name the other 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 send and receive primitives are defined as follows:

- send (P, message) – Send 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 symmetric and asymmetric schemes is the limited modularity of the resulting process definitions. Changing the name of a process may necessitate examining all other process definitions. All refrences to the old name must be found, so that they can be modified to the new name. this situations is not desirable from the viewpoint of separate compilation.

3.2.2.2 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 process via a number of different mailboxes. Two processes can communicate only if they share a mailbox. 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.
- A number different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Now suppose that processes P_1, P_2 and P_3 all share mailbox *A*. Process P_1 sends a message to *A*, while P_2 and P_3 each a receive from *A*. Which process will receive the message sent by P_1? 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 P_2 or P_3, but not both, will receive the message). The system may identify the receiver to the sender.

A mailbox may be owned by either a process or by the operating system. if the mailbox is owned by a process (i.e. that 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 (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 not ified that the mailbox no longer exists.

On the other hand, a mailbox owned by the operating system 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 the mailbox. However, the ownership and receive privilege mat be passed to other processes through appropriate system calls. Of course, the provision would result in multiple receiver for each mailbox.

3.2.3 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.

3.2.4 Buffering

Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such a queue can be implenmented in three ways:

- **Zero capacity:** The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, te sender must block until the recipient receives the message.
- **Bounded capacity:** The queue has finite length *n*; thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue the execution without waiting. The link has a finite capacity, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity:** The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks.

The zero capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as automatic buffering.

4.0 Conclusion

In this unit, you have been taken through the concept of co-operating processes, the means through which they communicate, and the various means to ensure synchronization between communicate processes.

5.0 Summary

As you have learnt in this unit, the processes in the system can execute concurrently. There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity,, and convenience. Concurrent execution requires mechanism for process creation and deletion.

The processes executing in the operating system may be either independent processes or co-operating processes. Co-operating processes must have the means to communicate with each other. Principally, two complementary communication schemes exist: shared memory and message systems. The shared-memory method requires communicating processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared memory system, the responsibility for providing communication rests with the application programmers, the operating system needs to provide only the shared memory. The message-system method allows the processes to exchange messages. The responsibility for providing communication may resy with the operating system itself. These two schemes are not mutually exclusive, and can be used simultaneously within a single operating system.

6.0 Tutor-Marked Assignment

1. What do you understand by co-operating processes
2. State reasons for allowing process co-operation
3. What do you understand by interprocess communication (IPC)
4. What are the benefits and detriments of each of the following? Consider both the system and the programmers' levels.
    a. Direct and indirect communication
    b. Symmetric and asymmetric communication
    c. Automatic and explicit buffering
    d. Send by copy and send by reference
    e. Fixed-sized and variable-sized messages
5. Consider the IPC scheme where mailboxes are used.
    a. Suppose a process *P* wants to wait for two messages, one from mailbox *A* and one from mailbox *B*. What sequence of send and receive should it execute?
    b. What sequence of send and receive should *P* execute if *P* wants to wait for one message from mailbox *A* or from mailbox *B* (or from both)
    c. A receive operation makes a process wait until the mailbox is nonempty. Devise a scheme that allows a process to wait until a mailbox is empty, or explain why such a scheme cannot exist.
6. Briefly explain buffering and the various ways it can be implemented.

7.0 References/Further Reading

1. Silberschatz, Abraham; Cagne, Greg, Galvin, Peter Baer (2004). *Operating system concepts with Java*, Sixth Edition, John Wiley & Sons, Inc.. ISBN 0-471-48905-0.
2. Stallings, William (2005). *Operating Systems: internals and design principles (5th edition)*. Prentice Hall. ISBN 0-13-127837-1.

Module 3: Process Management

Unit 3: Threads

Table of Contents

1.0 Introduction

A thread, sometimes called a **lightweight process** (LWP), is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating system resources, such as open files and signals. A traditional (or **heavyweight**) process has a single thread of control. If the process has multiple threads of control, it can do more than one task at a time. Figure 3.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

2.0 Objectives

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

- Distinguish between a thread and a process
- Enumerate the advantages of threads over processes
- Distinguish between user and kernel threads

- Describe various multithreading models and their advantages and disadvantages.
- State the advantages of thread pools and the motivation for thread pools.

3.0 Main Body

3.1 Threads

In modern operating systems, each process can have several *threads of execution* (or *threads* for short). Multiple threads share the same program code, operating system resources (such as memory and file access) and operating system permissions (for file access as the process they belong to). A process that has only one thread is referred to as a *single-threaded* process, while a process with multiple threads is referred to as a *multi-threaded* process. Multi-threaded processes have the advantage that they can perform several tasks concurrently without the extra overhead needed to create a new process and handle synchronised communication between these processes. For example a word processor could perform a spell check as the user types, without freezing the application - one thread could handle user input, while another runs the spell checking utility.

3.1.1 Motivation

Many software packages that run on modern desktop PCs are **multithread**. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network. A word processor may have athread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

In certain situations a single application may be required to perform several similar tasks. For instance, a web server accepts client requests for web pages, images, sound, and so on. A bus web server may have several (perhaps hundreds of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time. The amount of time that a client might have to wait for its request to be serviced could be enormous.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation, as you have seen in the previous unit is very heavyweight. If the new process will perform the same tasks as the existing process, why incur all that overhead?

It is generally more efficient for one process that contains multiple threads to serve the same purpose. This approach would multithread the web-server process. The server would create a separate thread that would listen for client requests; when a request was made; rather than creating another process, it would create another thread to service the request.

3.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness:** Multithreading an interactive application may allow program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread.

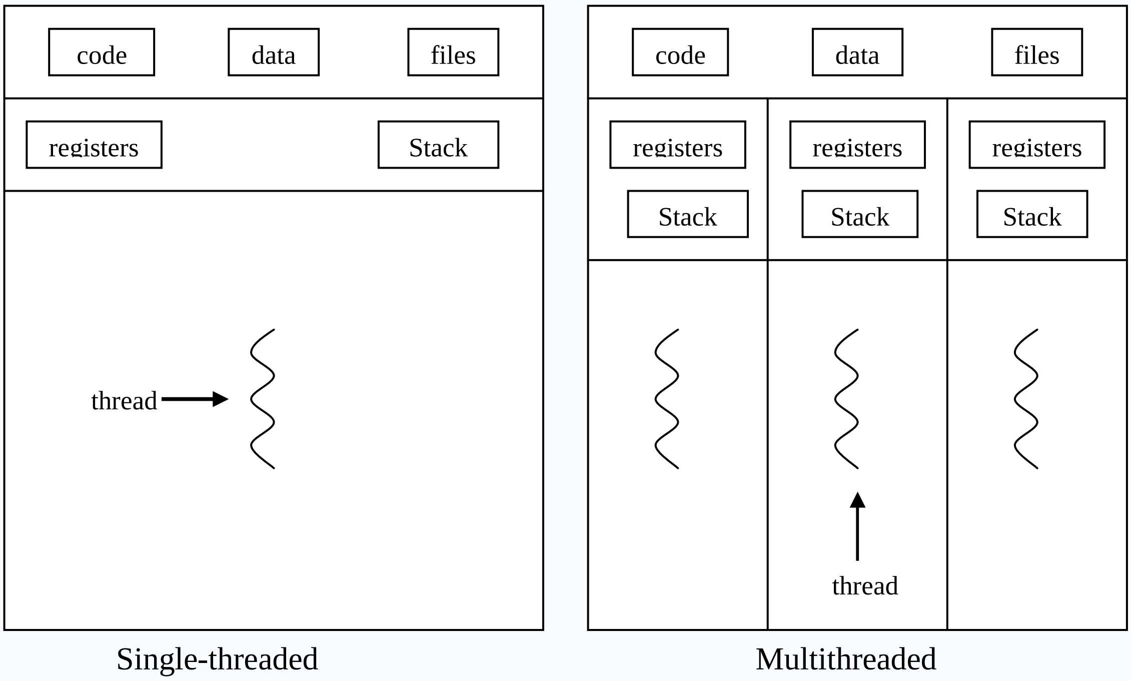


Figure 3.1: Single- and multithreaded processes

2. **Resource sharing:** By default, threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space.
3. **Economy:** Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads. It can be difficult to gauge empirically he difference in overhead for creating and maintaining a process rather than a thread, but in general it is much more time consuming to create and mange processes than threads.
4. **Utilization of multiprocessor architectures:** Multithreading is a popular programming and execution model that allows multiple threads to exist within the context of a single process, sharing the process' resources but able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be

running in parallel on a different processor. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency. In single-processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time.

3.1.3 Types of Threads

Threads can be classified into two different types viz: user threads and kernel threads, depending on the level at which support is provided for threads.

Support for threads may be provided at either the user level in which case the thread is referred to as **user threads** or **fibers**, or by the kernel, in which case it is referred to **kernel threads**.

- **User threads (Fibers):** These are supported above the kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. Since the kernel is not aware of user-level threads, fiber creation and scheduling are done in user space without the need for kernel intervention. Therefore, fibers are generally fast to create and manage.

  However, the use of blocking system calls in fibers can be problematic. If a fiber performs a system call that blocks, the other fibers in the process are unable to run until the system call returns. A typical example of this problem is when performing I/O: most programs are written to perform I/O synchronously. When an I/O operation is initiated, a system call is made, and does not return until the I/O operation has been completed. In the intervening period, the entire process is "blocked" by the kernel and cannot run, which starves other fibers in the same process from executing.

  As mentioned earlier, fibers are implemented entirely in **userspace**. As a result, context switching between fibers in a process does not require any interaction with the kernel at all and is therefore extremely efficient: a context switch can be performed by locally saving the CPU registers used by the currently executing fiber and loading the registers required by the fiber to be executed. Since scheduling occurs in userspace, the scheduling policy can be more easily tailored to the requirements of the program's workload.

  User-thread libraries include POSIX **Pthreads**, Mach **C-threads**, and Solaris 2 **UI-threads.**

- **Kernel threads:** These are supported directly by the operating system. The kernel performs thread creation, scheduling and management in kernel space. Due to the fact that thread management is done by the operating system, kernel threads are generally slower to create and manage than are user threads. However, since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. Also in multiprocessor environment, the kernel can schedule threads on different processors. Most contemporary operating systems – including Windows NT, Windows 200, Solaris 2, BeOS, and Tru64 UNIX support kernel threads. Most contemporary operating systems – including Windows NT, Windows 200, Solaris 2, BeOS, and Tru64 UNIX support kernel threads.

The use of kernel threads simplifies user code by moving some of the most complex aspects of threading into the kernel. The program does not need to schedule threads or explicitly yield the processor. User code can be written in a familiar procedural style, including calls to blocking APIs, without starving other threads. However, since the kernel may switch between threads at any time, kernel threading usually requires locking that would not be necessary otherwise. Bugs caused by incorrect locking can be very subtle and hard to reproduce. Kernel threading also has performance limits. Each time a thread starts, blocks, or exits, the process must switch into kernel mode, and then back into user mode. This context switch is fairly quick, but programs that create many short-lived threads can suffer a performance hit. Hybrid threading schemes are available which provide a balance between kernel threads and fibers.

3.2 Multithreading Implementation

Operating systems generally implement threads in one of two ways: preemptive multithreading, or cooperative multithreading. Preemptive multithreading is generally considered the superior implementation, as it allows the operating system to determine when a context switch should occur. Cooperative multithreading, on the other hand, relies on the threads themselves to relinquish control once they are at a stopping point. This can create problems if a thread is waiting for a resource to become available. The disadvantage to preemptive multithreading is that the system may make a context switch at an inappropriate time, causing priority inversion or other bad effects which may be avoided by cooperative multithreading.

Traditional mainstream computing hardware did not have much support for multithreading as switching between threads was generally already quicker than full process context switches. Processors in embedded systems, which have higher requirements for real-time behaviors, might support multithreading by decreasing the thread switch time, perhaps by allocating a dedicated register file for each thread instead of saving/restoring a common register file. In the late 1990s, the idea of executing instructions from multiple threads simultaneously has become known as simultaneous multithreading. This feature was introduced in Intel's Pentium 4 processor, with the name *Hyper-threading*.

3.2.1 Multithreading Implementation Models

Many systems provide support for both fibers and kernel threads, resulting in many different and incompatible implementations of threading. In this section we will look at the three common types of multithreading implementation.

3.2.1.1 Many-to-One Model

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done in user space, so it is efficient, but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. **Green threads**, a thread library available for Solaris 2, use this model. In addition, fiber libraries implemented on operating systems that do not support kernel threads use the many-to-one model.

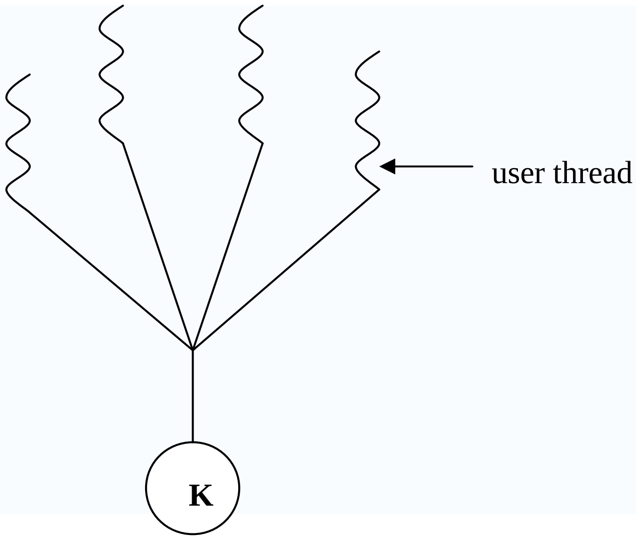


Figure 3.2: Many-to-One Model

3.2.1.2 One-to-One Model

The one-to-one model maps each user thread (fiber) to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a fiber requires creating the corresponding kernel thread. Most implementations of this model restrict the number of threads supported by the system because the overhead of creating kernel threads can burden the performance of an application.

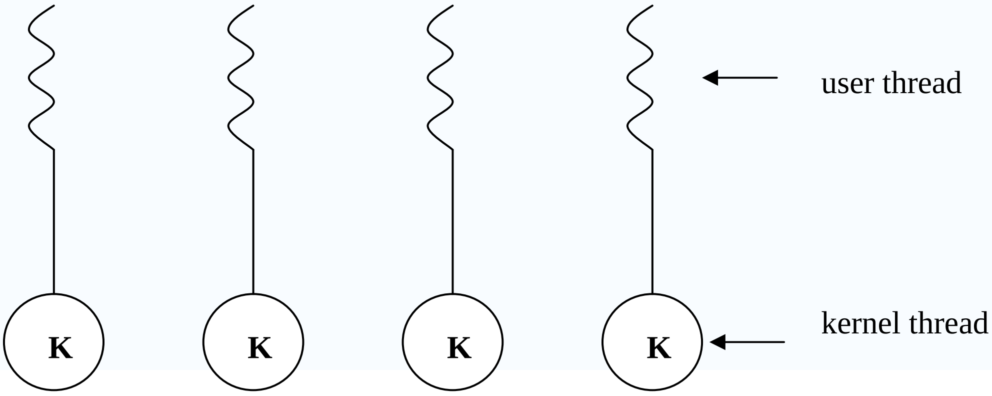


Figure 3.3: One-to-One Model

3.2.1.3 Many-to-Many Model

The many-to-many model multiplexes many fibers to a smaller or equal number of kernel threads. The number of kernel threads may be specified to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a miniprocessor. whereas the many-to-one model allows the developer to create as many fibers as she wishes, true concurrency is not gained because kernel can schedule only one thread at a time. The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create) the many-to-many model suffers from neither of these shortcomings. Developers can create as many fibers as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also when a thread performs a blocking system call, the kernel can schedule another thread for execution. Solaris 2, IRX, HP-UX, and Tru64 UNIX support this model.

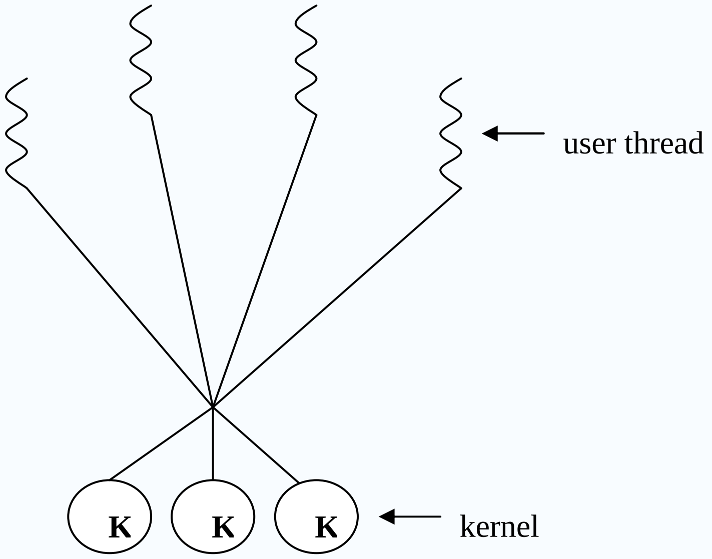


Figure 3.4: Many-to-Many Model

3.3 Threading Issues

3.3.1 Thread Cancellation

This is the task of terminating a thread before it has completed. For instance, if multiple threads are running concurrently searching through a database and one returns the result, the remaining threads might be cancelled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often a web page is loaded in a separate thread. When a user presses the stop button, the thread loading the page is cancelled.

A thread that is to be cancelled is often referred as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation: one thread immediately terminates the target thread.
2. Deferred cancellation: the target thread can periodically check if it should terminate, allowing the target an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a cancelled thread or if a thread was cancelled while in the middle of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. The operating system will often reclaim system resources from a cancelled thread, but often will not claim all resources. Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.

Alternatively, deferred cancellation works by one thread indicating that a target thread is to be cancelled. However, cancellation will occur only when the target thread checks to determine if it should be cancelled or not . This allows a thread to check if it should be cancelled at a point when it can safely be cancelled. Pthreads refers to such as cancellation point.

3.3.2 Signal Handling

In UNIX systems, a **signal** is used to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source and the reason for the event being signalled. Whether a signal is synchronous or asynchronous, all signals follow the following pattern:

a) A signal is generated by the occurrence of a particular event.
b) A generated signal is delivered to a process.
c) Once delivered, the signal must be handled.

An example of a synchronous signal includes an illegal memory access or division by zero. Synchronous signals are delivered to the same process that performed the operation causing the signal, hence the name synchronous.

When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes or having a timer expire. Typically an asynchronous signal is sent to another process.

Every signal may be handled by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler.

Every signal has a **default signal handler** that is run by the kernel when handling the signal. This default action may be overridden by a **user-defined signal handler** function. In this instance, the user-defined function is called to handle the signal rather than the default action. Both synchronous and asynchronous signals may be handled in different ways. Some signals may be simply ignored (such ads changing the size of a window); others may be handled by terminating the program (such as an illegal memory access.

Handling signals in single-threaded programs is straightforward. Signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, as a process may have several threads. Generally, a signal can be delivered in any of the following ways:

- Deliver the signal to the thread to which the signal applies.
- Deliver the signal to every thread in the process.
- Deliver the signal to certain threads in the process.
- Assign a specific thread to receive all signals for the process.

The method of delivering a signal depends on the type of signal generated.

3.3.3 Thread Pools

In the scenario of multithreading a web server, whenever the server receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly faster than creating a separate process, a multithreaded server nonetheless has potential problems. The first concerns the amount of time required to create the thread prior to servicing the request, compounded with the fact that this thread will be discarded once it has completed its work. The second issue is more problematic: if we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this issue is to thread pools.

The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool (if one is available) passing it the request to service. Once the thread completes its service, it returns to the pool awaiting more work. If the pool contains no available thread, the server waits until one becomes free.

3.3.3.1 Advantages of Thread Pools

In particular, thread pools have the following advantages:

1. It is usually faster to service a request with an existing thread than waiting to create a thread.

2. A thread pool limits the number of threads that exist at any point in time. This is particularly important on systems that cannot support a large number of concurrent threads.

The number of threads in the pool can be set heuristically based upon factors such as the number of CPUs in the system, the amount of physical memory and the expected number of concurrent client requests. More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns. Such architectures provide the further advantage of having a smaller pool, thereby consuming less memory, when the load on the system is low.

3.3.4 Thread-Specific Data

Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, each thread might need its own copy of certain data in some circumstances. Let us call such data **thread-specific data**. For instance, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction may be assigned a unique identifier. To associate each thread with its unique identifier we could use thread-specific data. Most libraries, including Win32 and Pthreads, provide some form of support for thread-specific data. Java provides support as well.

4.0 Conclusion

In this unit you have been introduced to the main concept of light-weight processes popularly known as threads. It is believed that having gone through this unit you are now conversant with threads, the main issues concerning threads, the motivation for threads, etc.

5.0 Summary

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and the ability to take advantage of multiprocessor architectures.

Fibers are threads that are visible to the programmer and are unknown to the kernel. A thread library in user space typically manages fibers. The operating system kernel supports and manages kernel-level threads. In general, fibers are faster to create and manage than are kernel threads. Three different types of models relate fibers and kernel-level threads: The many-to-one maps many fibers to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads. Other issues include thread cancellation, signal handling and thread-specific data.

6.0 Tutor-Marked Assignment

1. What are the two major differences between fibers and kernel-level threads? Under what circumstances is one type better than the other?
2. What resources are used when a thread is created? How do they differ from those used when a process is created?
3. Describe the action taken by a kernel to context switch between kernel threads.
4. Describe the action taken by a thread library to context switch between fibers.
5. Provide two programming examples of multithreading that improves performance over a single-threaded solution
6. Provide two programming examples of multithreading that do not improve performance over a single-threaded solution

7.0 References/Further Reading

1. David R. Butenhof: *Programming with POSIX Threads*, Addison-Wesley, ISBN 0-201-63392-2
2. Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farell: *Pthreads Programming*, O'Reilly & Associates, ISBN 1-56592-115-1
3. Charles J. Northrup: *Programming with UNIX Threads*, John Wiley & Sons, ISBN 0-471-13751-0
4. Mark Walmsley: *Multi-Threaded Programming in C++*, Springer, ISBN 1-85233-146-1
5. Paul Hyde: *Java Thread Programming*, Sams, ISBN 0-672-31585-8
6. Bill Lewis: *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, ISBN 0-13-443698-9
7. Steve Kleiman, Devang Shah, Bart Smaalders: *Programming With Threads*, SunSoft Press, ISBN 0-13-172389-8
8. Pat Villani: *Advanced WIN32 Programming: Files, Threads, and Process Synchronization*, Harpercollins Publishers, ISBN 0-87930-563-0
9. Jim Beveridge, Robert Wiener: *Multithreading Applications in Win32*, Addison-Wesley, ISBN 0-201-44234-5
10. Thuan Q. Pham, Pankaj K. Garg: *Multithreaded Programming with Windows NT*, Prentice Hall, ISBN 0-13-120643-5
11. Len Dorfman, Marc J. Neuberger: *Effective Multithreading in OS/2*, McGraw-Hill Osborne Media, ISBN 0-07-017841-0
12. Alan Burns, Andy Wellings: *Concurrency in ADA*, Cambridge University Press, ISBN 0-521-62911-X
13. Silberschatz, Abraham; Cagne, Greg, Galvin, Peter Baer (2004). "Chapter 4", *Operating system concepts with Java*, Sixth Edition, John Wiley & Sons, Inc.. ISBN 0-471-48905-0.
14. Stallings, William (2005). *Operating Systems: internals and design principles (5th edition)*. Prentice Hall. ISBN 0-13-127837-1.

Module 3: Process Management

Unit 4: CPU Scheduling

Table of Contents

1.0 Introduction

CPU scheduling is the basis of multi-programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this unit, you are going to be introduced to the basic scheduling concepts and be presented with several different CPU-scheduling algorithms. The problem of selecting an algorithm for a particular system will also be considered.

2.0 Objectives:

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

- Distinguish between preemptive and non-preemptive scheduling
- State the goals for CPU scheduling
- Give comparative analysis of the following scheduling algorithms:
  - FCFS
  - SJF
  - Priority Scheduling
  - Round-Robin Scheduling

    - Multilevel Queue Scheduling
    - Multilevel Feedback Queue Scheduling
- Select a CPU scheduling algorithm for a particular system.

3.0 Main Body

3.1 Ontology: Basic Concepts

The objective of multiprogramming is to have some process running at times, in order to maximize CPU utilization. In a uniprocessor system, only one process may run at a time; any other processes must wait until the CPU is free and can be rescheduled.

The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then sit idle. All this waiting time is wasted. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.

Scheduling is fundamental to operating system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating system design.

3.1.1 CPU-I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes. Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, then another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with system request to terminate execution, rather than with another I/O burst (see Figure 3.1)

The duration of these CPU burst have been extensively measured. Although they vary greatly by process and by computer, they tend to have a frequency curve similar to that shown in Figure 3.2. The curve is generally characterized as exponential or hyper-exponential, with many short CPU bursts and a few long CPU bursts. An I/O-bound program would typically have many very short CPU bursts while a CPU-bound program might have a few very long CPU bursts. This distribution can help us select an appropriate CPU-scheduling algorithm.

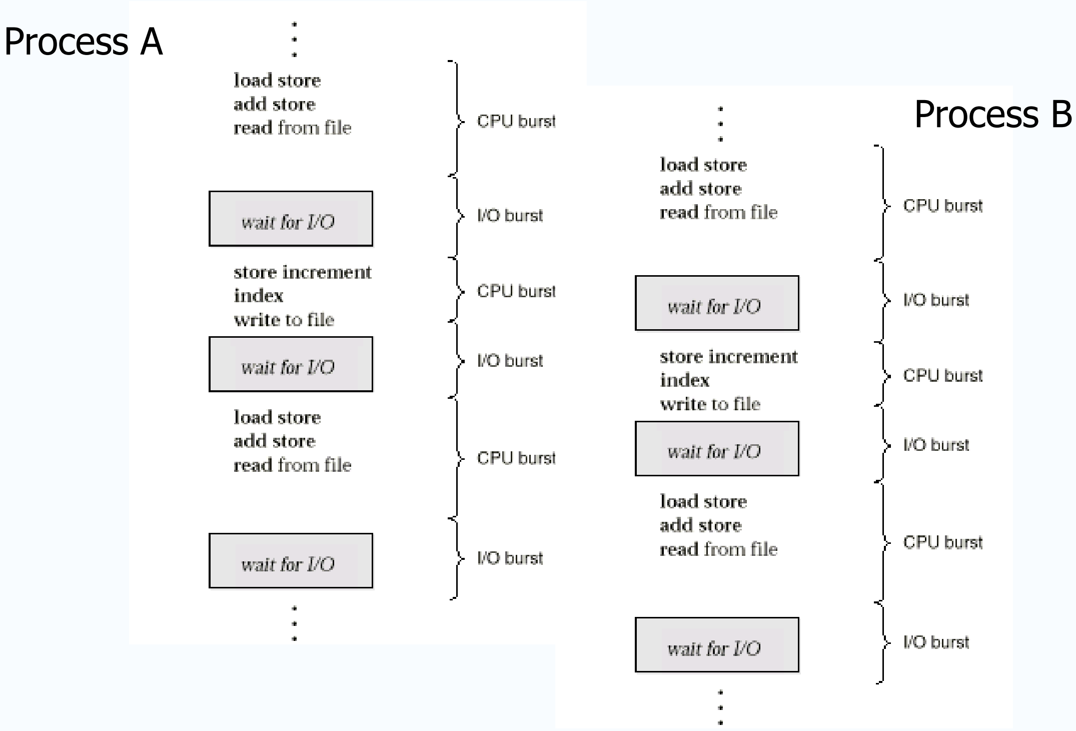


Figure 3.1 Alternating Sequence of CPU and I/O Bursts of Two Processes.

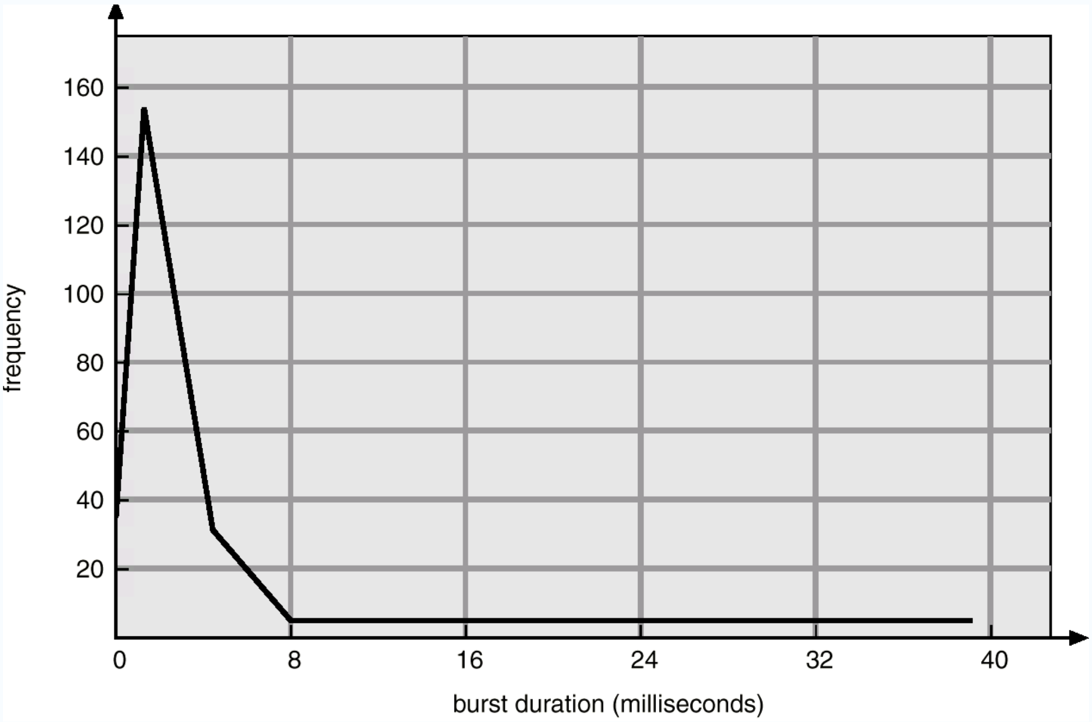


Figure 3.2: Histogram of CPU-Bursts Times.

3.3.1 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select on the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

The ready queue is not necessarily a first-in, first-out (FIFO) queue. As you shall see, when we consider the various scheduling algorithms, a ready queue may be implemented as a FIFO queue, a priority queue, a tree, or simply an ordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

3.3.2 Preemptive Scheduling and Non-preemptive Scheduling

CPU scheduling decisions may take place under the following circumstances:

1. When a process switches from the running state to the waiting state (for example, I/O request, or invocation of wait for the termination of one of the child processes)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, completion of I/O)
4. When a process terminates

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, in circumstances 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **nonpreemptive**. Otherwise, the scheduling scheme is **preemptive**. Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method is used by Microsoft Windows 3.1 operating system. It is the only method that can be used on certain hardware platforms, because it does not require the special hardware needed for preemptive scheduling.

Some of the disadvantages of preemptive scheduling are that:

- It incurs a cost
- It also has an effect on the design of the operating system kernel.

3.1.4 Dispatcher

Another component involved in the CPU scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function includes:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

3.2 Scheduling Criteria

Different CPU-scheduling algorithms have different properties and may favour one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. The characteristics used for comparison can make a substantial difference in the determination of the best algorithm. The criteria include the following:

CPU Utilization: We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

Throughput: if the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called throughput. For long processes, this rate may be 1 process per hour or 10 processes per second for short transactions.

Turnaround Time: This is the interval from the time of submission of a process to the time of completion. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU and doing I/O.

Waiting Time: The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is, therefore, the sum of the periods spent waiting in the ready queue.

Response Time: This is the amount of time it takes to start responding but not the time it takes to output the response. i.e. the time from the submission of a request until the first response is produced.

We usually want to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, in some circumstances we want to optimize the minimum or maximum values, rather than the average. For instance, to guarantee that all users get good service, we may want to minimize the maximum response time.

3.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. In this section, we describe many CPU-scheduling algorithms that exist.

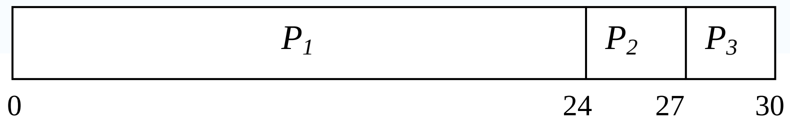
3.3.1 First-Come, First Served (FCFS) Scheduling

This is the simplest CPU-scheduling algorithm. In this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is the removed from the queue. The code for FCFS scheduling is simple to write.

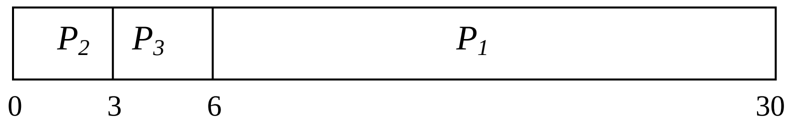
The average waiting time under FCFS policy is often quite long.

Process	**Burst time**
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the Gantt Chart below:



The waiting time is 0 milliseconds for process P_1, 24 milliseconds for process P_2, and 27 milliseconds for process P_3. Hence the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P_2, P_3, P_1, however, the result will be as shown in the Gantt chart below:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Therefore, the average waiting time under FCFS policy is generally not minimal, and may vary substantially if the process CPU-burst times vary greatly.

FCFS scheduling algorithm may lead to convoy effect whereby all other processes wait for one big process to get off the CPU. This results in lower CPU and device utilization. FCFS scheduling algorithm is non-preemptive.

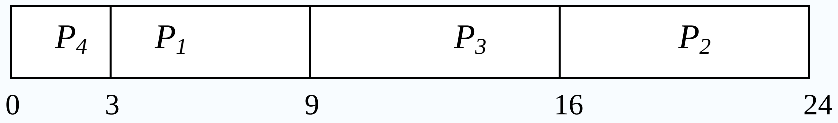
3.3.2 Shortest-Job-First (SJF) Scheduling

This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

Example 3.2 Consider the following set of processes that arrive at time 0, wit the length of the CPU-burst time given in milliseconds:

Process	**Burst time**
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we could schedule these processes according to the Gantt chart below:



The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Hence the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. If we were using FCFS scheduling scheme, then the average waiting time would be 10.25 milliseconds.

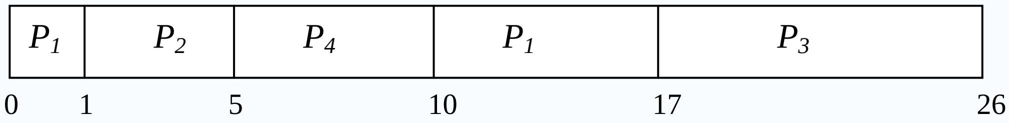
The SJF scheduling algorithm is provably optimal because it gives the minimum average waiting time for a given set of processes. However, it cannot be implemented at the level of short-term CPU scheduling because there is no way to know the length of the next CPU burst.

SJF algorithm may be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new process may have a shorter next CPU burst than what is left of the current executing process. A preemptive SJF algorithm will pre-empt the current executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometime called shortest-remaining-time-first scheduling.

Example 3.3: Consider the following four processes, with length of the CPU-burst given in milliseconds:

Process	**Arrival time**	**Burst time**
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in Gantt chart below:



Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is pre-empted, and process P_2 is scheduled. The average waiting time for this example is ((10 – 1) + (1 – 1) + (17 – 2) + (5 – 3)/4 = 26/4 = 6.5 milliseconds.

A nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

3.3.3 Priority Scheduling

A priority is associated with each process and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS.

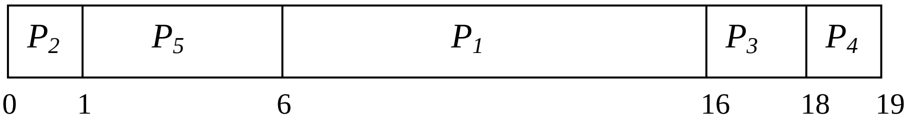
An SJF algorithm is therefore simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority and vice versa.

Priority is expressed in terms of fixed range number such as 0 to 10. however there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority while others use low numbers for high priority. But in this course, we will use low numbers to represent high priority.

Example 3.4: Consider the following set of processes, assumed to have arrived at time 0, in the order P_1, P_2, …, P_5, with the length of the CPU-burst time given in milliseconds:

Process	Burst time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the Gantt chart below:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use measurable quantity(ies) such as time limits, memory requirements, etc. to compute the priority of a process. External priorities are set by criteria that are external to the operating system such as importance of the process, amount being paid for use of the compute, the owner of the process, and other (political) factors.

Priority scheduling may be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with that of the currently running process. A preemptive priority-scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than that of the currently running process. A nonpreemptive priority-scheduling algorithm will simply put the new process at the head of the ready queue.

The major disadvantage of priority-scheduling algorithms is indefinite blocking or starvation. A situation whereby low priority processes indefinitely wait for the CPU because of a steady stream of higher-priority processes.

A solution to indefinite blocking of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

3.3.4 Round-Robin (RR) Scheduling

Round-robin (RR) is one of the simplest scheduling algorithms for processes in an operating system. It assigns time slices to each process in equal portions and in circular order, handling all processes without priority. Round-robin scheduling is both simple and easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks.

The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum (or time slice), is defined. A time quantum is generally from 10 – 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement the RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the queue. The CPU scheduler picks from the head of the queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

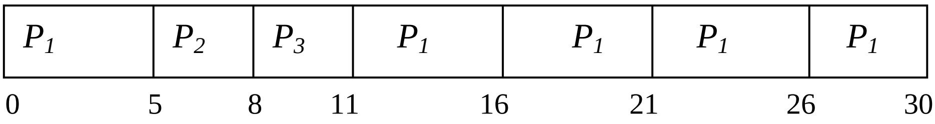
One of two things will then happen. The process may have a CPU burst of less than one time quantum. In which case the process itself releases the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The scheduler then selects the next process in the ready queue.

The average waiting time under RR policy is often quite long.

Example 3.5: Consider the following set of processes, assumed to have arrived at time 0, in the order *P_1*, *P_2*, *P_3*, with the length of the CPU-burst time given in milliseconds:

Process	**Burst time**
P_1	24
P_2	3
P_3	3

If we use a time quantum of 5 milliseconds, we would schedule these processes according to the Gantt chart below:



The average waiting time is 17/3 = 5.66 milliseconds.

In RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process' CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is preemptive.

3.3.5 Multilevel Queue (MLQ) Scheduling

In this scheduling algorithm, processes are classified into different groups. For instance, a common division is made between foreground (or interactive) processes and background (or batch) processes. Therefore the ready queue is partitioned into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, etc. Each queue has its own scheduling algorithm. E.g. foreground might use RR while background might use FCFS.

In addition, there must be scheduling among the queues which is usually implemented as fixed priority preemptive scheduling. For example foreground queue may have absolute priority over background queue. Therefore no process in the background queue could run except the foreground queues are empty. If a process entered the foreground queue while a prcesss from the background queue is running, the background queue process will be preempted.

This will lead to possible starvation for the background queue process. To address this problem, time slice can be used between the queue. Each queue gets a certain portion of the CPU time which it an then schedule among the various processes in its queue. For instance, in the background – foreground queue example, the foreground queue can be given 80% of the CPU time for RR scheduling among its processes, whereas the background queue receives 20% of the CPU to give to its processes in FCFS manner.

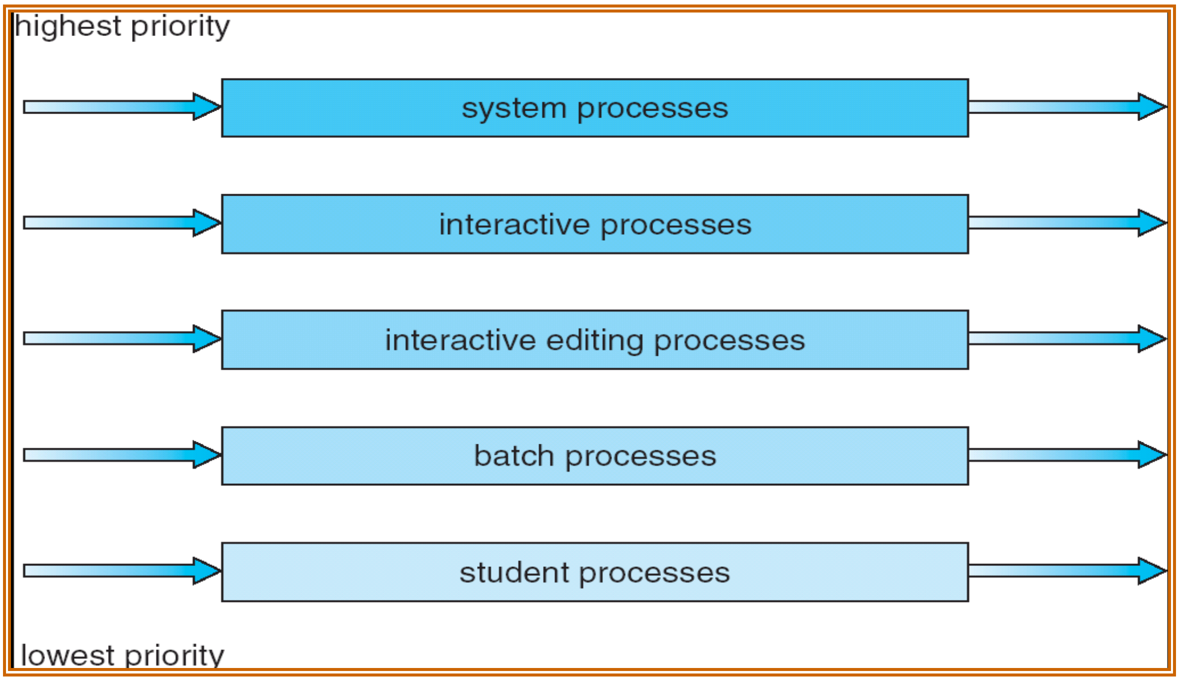


Figure 3.3: Multilevel Queue Scheduling

3.3.6 Multilevel Feedback Queue (MLFQ) Scheduling

Similar to MLQ, but here processes can move between the queues. The idea is to separate processes with different CPU-burst characteristics (i.e., based on their “behavior”). A process that uses too much CPU time is degraded to a lower-priority queue, a process that waits too long is upgraded to a higher-priority queue. This is a kind of aging that prevents starvation.

In general, MLFQ scheduler is defined by the following parameters:

- Number of queues
- Scheduling algorithms for each queue
- Criteria for determining when to upgrade a process to a higher-priority queue
- Criteria for determining when to demote a process to a lower-priority queue
- The criteria for determining which queue a process will enter when that process needs service.

MLFQ is the most general scheme, and also the most complex.

Example 3.6: Consider a MLFQ scheduler with three queues, *Q*0 with time quantum 8 milliseconds, *Q*1 with time quantum 16 milliseconds and *Q*2 on FCFS basis only when queues *Q*0 and *Q*1 are empty.

In this scheduling algorithm a new job enters queue *Q0* served by FCFS. Then job receives 8 milliseconds. If not finished in 8 milliseconds, it is moved to *Q*1. At *Q*1 job served by FCFS. It then receives 16 milliseconds. If not completed, it is preempted and moved to *Q*2 where it is served in FCFS order with any CPU cycles left over from queues *Q*0 and *Q*1.

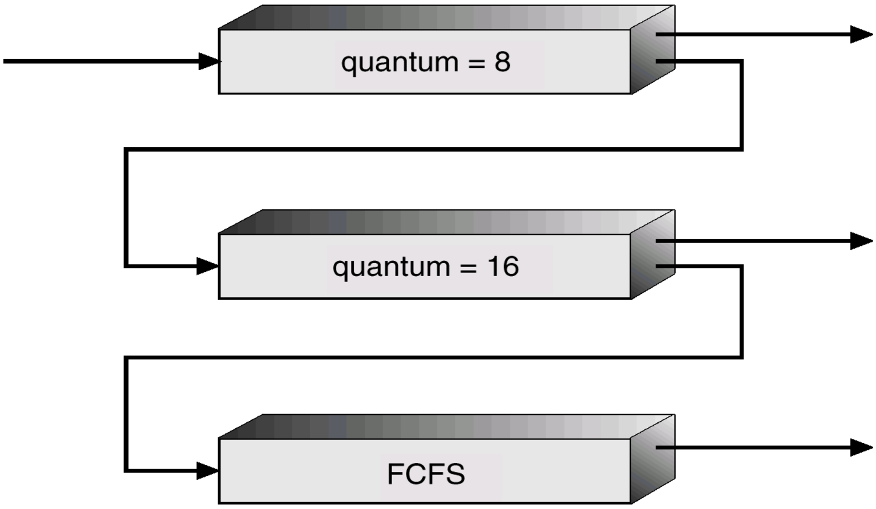


Figure 3.4: Multilevel Feedback Queues

3.4 Multiple-Processor Scheduling

Our discussion in this unit so far has focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, the scheduling problem is correspondingly more complex. Many possibilities have been tried but as you saw with single processor CPU scheduling, there is no one best solution. In his section we will briefly discuss some of the issues concerning multiple-processor scheduling.

If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case however, one processor could be idle, with an empty queue, while another processor could be very busy. To avoid such situation, we can use a common ready queue. All processes go into one queue and are scheduled onto any available processor.

In such a scheme, one of two scheduling approaches may be used. In one approach, each processor is self-scheduling. Each processor examines the common ready queue and selects a process to execute. However, we must ensure that no two processors choose the same process and that processes are not lost from the queue. The second approach avoids this problem by appointing one processor as scheduler for the other processors, thereby creating a master-slave structure.

Some systems go a step further by having all scheduling decisions, I/O processing, and other system activities handled by one single processor – the master server. The other processors only execute user codes.

SELF ASSESSMENT TEST

Suppose that the following processes arrive for execution at the times indicated. Each process will run the listed amount of time. In answering the questions, use nonpreemptive scheduling and base all decisions on the information you have at the time the decision must be made.

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

c. The SJF is supposed to improve performance, but not ice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future knowledge scheduling.

Process	**Arrival time**	**Burst time**
P_1	0	8
P_2	0	4
P_3	1	1

4.0 Conclusion

In this unit, you have been taken through the various CPU scheduling algorithms. It is our belief that by now you can select a scheduling algorithm that will be optimal for a particular situation/system. However, there are formal techniques for determining the best scheduling algorithm for a particular situation and this we will discuss this in this next unit.

5.0 Summary

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

FCFS scheduling is the simplest scheduling algorithm but it can cause short processes to wait for very long ones. SJF scheduling is provably optimal, producing the shortest waiting time. SJF is a special case of the general priority scheduling algorithm, which simply allocates the CPU to the highest priority process. But both SJF and priority scheduling may suffer from starvation. Aging is a technique to prevent starvation.

RR scheduling is more appropriate for a time shared (interactive) system. The major problem is the selection of the time quantum. Too large quantum will make the RR scheduling to degenerate to FCFS scheduling while too small quantum results in scheduling overhead in the form of context-switch time becoming excessive.

FCFS algorithm is non-preemptive; RR algorithm is preemptive. SJF and priority algorithms may be either preemptive or non-preemptive. MLQ algorithms allow different algorithms to be used for various classes of processes. MLF queues allow processes to move from one queue to another.

6.0 Tutor Marked Assignments

1. Explain the differences in the degree to which the following scheduling algorithms discriminate in favour of short processes:

   a) FCFS

   b) RR

   c) Multilevel Feedback queues

2. Define the differences between pre-emptive and non-pre-emptive scheduling. State why strict non-preemptive scheduling is unlikely to be used in a computer centre.
3. Consider the following set of processes, with the length of the CPU-burst time given in milliseconds as below:

Process	Burst time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order $P_{1,}$ $P_{2,}$ $P_{3,}$ $P_{4,}$ $P_{5,}$ all at time 0.

a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.

b. What is the turnaround time of each process for each of the scheduling algorithms in (a) above?

c. What is the waiting time of each process for each of the scheduling algorithms in (a)?

d. Which of the schedules in (a) results in the minimal average waiting time (over all processes)?

7.0 Reference/Further Reading

1. Deitel, Harvey M.; Deitel, Paul; Choffnes, David (2004). *Operating Systems*. Upper Saddle River, NJ: Pearson/Prentice Hall. ISBN 0-13-182827-4.
2. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5.
3. Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems. Design and Implementation*. Upper Saddle River, N.J.: Pearson/Prentice Hall. ISBN 0-13-142938-8.
4. Tanenbaum, Andrew S. (2001). *Modern Operating Systems*. Upper Saddle River, N.J.: Prentice Hall. ISBN 0-13-092641-8.

Module 3: Process Management

Unit 5: Algorithm Evaluation

Table of Contents

1.0 Introduction

How do we select a CPU-scheduling algorithm for a particular system? As you saw in the previous unit, there are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult. The first problem is defining the criteria to be used in selecting an algorithm. As you saw in the previous unit, criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, you must first define the relative importance of these measures. Your criteria may include several measures, such as:

- Maximize CPU utilization under the constraint that the maximum response time is 1 second.
- Maximize throughput such that turnaround is (on average) linearly proportional to total execution time.

Once the selection criteria have been defined, we are then going to evaluate the various algorithms under consideration. We describe the different evaluation methods in the rest of this unit.

2.0 Objectives

At the end of this unit you should be able to:

- Describe the various CPU scheduling evaluation algorithms.
- Enumerate the advantages and disadvantages of each evaluation algorithms
- Based on your knowledge, select the best scheduling algorithm for a particular a particular system.

3.0 Main Body

3.1 Deterministic Modelling

Deterministic modelling is a type of analytical evaluation. Analytical evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.

Deterministic modelling takes a particular predetermined workload and defines the performance of each algorithm for that workload.

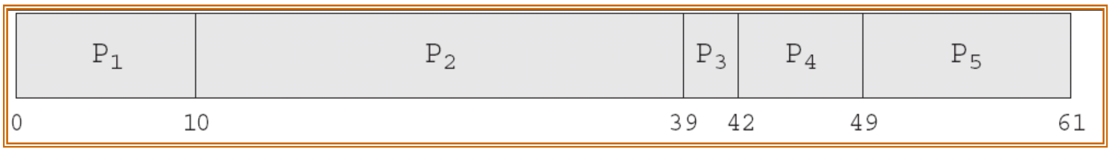
Example 1:

Assume we have the workload shown. All five processes arrive at time 0, in the order given, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

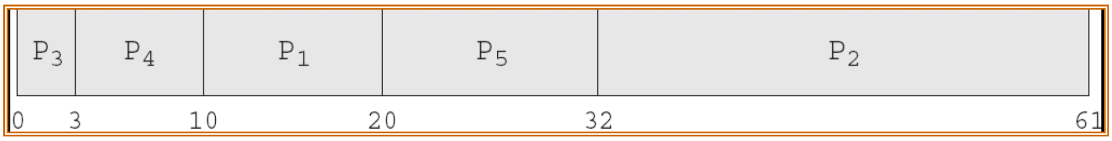
Consider the FCFS, SJF, and RR (quantum = 10milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as:



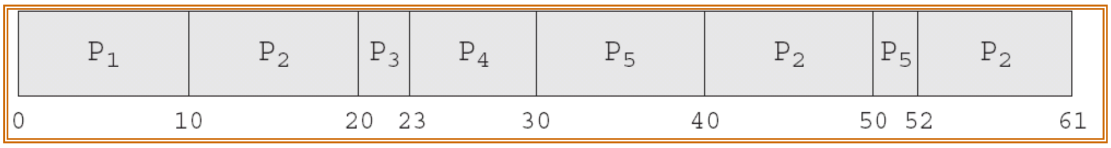
The waiting time is 0 milliseconds for process P_1, 10 milliseconds for process P_2, 39 milliseconds for P_3, 42 milliseconds for process P_4 and 49 milliseconds for process P_5. Therefore, the average waiting time is (0 + 10 + 39 + 42 + 49)/5 = 28 milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as:



The waiting time is 10 milliseconds for process *P_1*, 32 milliseconds for process *P_2*, 0 milliseconds for *P_3*, 3 milliseconds for process *P_4* and 20 milliseconds for process *P_5*. Therefore, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With RR algorithm, we execute the processes as:



The waiting time is 0 milliseconds for process *P_1*, 32 milliseconds for process *P_2*, 20 milliseconds for *P_3*, 23 milliseconds for process *P_4* and 40 milliseconds for process *P_5*. Therefore, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 13 milliseconds.

You can see that in this case, the SJF results in less than one-half the average waiting time obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Advantages of Deterministic Modelling:

- It is simple and fast.
- It gives exact numbers, allowing the algorithms to be compared.

Disadvantages of Deterministic Modelling:

- It requires exact numbers for input and its answers apply to only those cases.
- It is too specific.
- It requires too much knowledge to be useful.

3.2 Queuing Models

Queuing models employ probabilistic distributions for CPU and I/O bursts The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, etc. This area of study is called queuing-network analysis.

For instance, let *n* be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). Then we expect that during the time W that a process waits, $\lambda \times W$ new processes will arrive in the q. if the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Therefore,

$n = \lambda \times W.$

This equation is known as Little’s formula. The formula is particularly useful because it is valid for any scheduling algorithm and arrival distribution. It can be used to compute one of the three variables once the other two are known.

Advantages of Queuing Analysis:

- It can be useful in comparing scheduling algorithms.

Limitations:

- The classes of algorithms and distribution that can be handled is presently limited
- It is hard to express a system of complex algorithms and distributions.
- Queueing models are often an approximation of a real system. As a result, the accuracy of the computed results may be questionable.

3.3 Simulations

This is used to get a more accurate evaluation of scheduling algorithms. Simulations involve programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock; as this variable’s value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

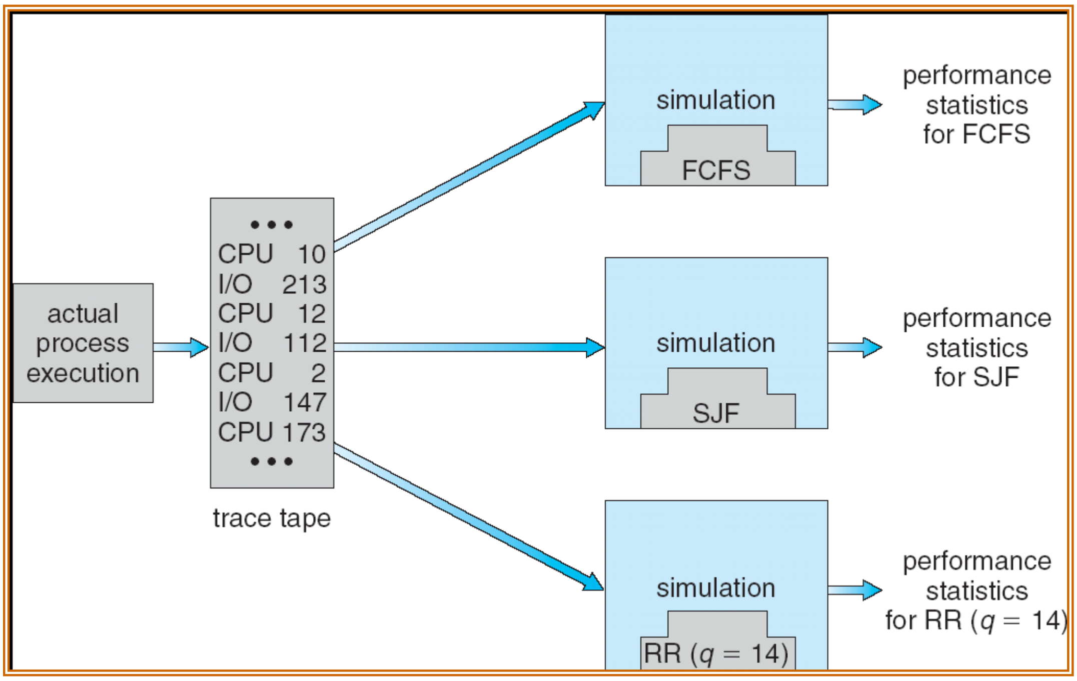


Figure 3.1: Evaluation of CPU scheduler by Simulation.

Advantages:

- It produces accurate results for its inputs.

Disadvantages:

- It can be expensive
- Trace tapes can require large amounts of storage space.
- The design, coding and debugging of the simulator can be a major task.

3.4 Implementation

Even a simulator is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

Limitations:

- This approach is very expensive. The expense is incurred not only in coding the algorithm and modifying the operating system to support it as well as its required data structure, but also in the reaction of the users to a constantly changing operating system.

4.0 Conclusion

This unit has taken you through the various scheduling evaluation algorithms. However, as you have seen there is no perfect evaluation algorithm. There are always difficulties to be encountered when evaluating scheduling algorithms. One of the major difficulties with any algorithm evaluation is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of small processes. If interactive processes are given priority over non-interactive processes, then users may switch to interactive use.

5.0 Summary

Due to the fact that a wide variety of scheduling algorithms are available, we need methods/means of selecting among them. Analytical methods use mathematical analysis to determine the performance of an algorithm. Simulation methods determine performance by imitating the scheduling algorithm on a “representative” sample of processes, and computing the resulting performance.

6.0 Tutor Marked Assignment

1. Briefly compare the evaluation algorithms discussed in this unit. Which one is best? Give reasons.

2. Briefly describe the deterministic model. What are its advantages and disadvantages?

7.0 References/Further Reading

1. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5.
2. Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems Design and Implementation*. Upper Saddle River, N.J.: Pearson/Prentice Hall. ISBN 0-13-142938-8.

Module 4: Process Synchronization

Unit 1: Race Condition

Table of Contents

1.0 Introduction

A **race condition** or **race hazard** is a flaw in a system or process whereby the output of the process is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two signals racing each other to influence the output first. Race conditions arise in software when separate processes or threads of execution depend on some shared state.

2.0 Objectives

At the end of this unit you should be able to:

- Define Race condition
- Describe some real life examples of race condition
- Describe computer security in view of race condition

3.0 Main Body

3.1 Race Condition

As said earlier, **race condition** is a flaw in a system or process whereby the output of the process is unexpectedly and critically dependent on the sequence or timing of other events. To illustrate this, let us look at this simple example:

Let us assume that two threads T1 and T2 each wants to increment the value of a global integer by one. Ideally, the following sequence of operations would take place:

1. Integer i = 0;
2. T1 reads the value of i from memory into a register : 0
3. T1 increments the value of i in the register: (register contents) + 1 = 1
4. T1 stores the value of the register in memory : 1
5. T2 reads the value of i from memory into a register : 1
6. T2 increments the value of i in the register: (register contents) + 1 = 2
7. T2 stores the value of the register in memory : 2
8. Integer i = 2

In the case shown above, the final value of i is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. The alternative sequence of operations below demonstrates this scenario:

1. Integer i = 0;
2. T1 reads the value of i from memory into a register : 0
3. T2 reads the value of i from memory into a register : 0
4. T1 increments the value of i in the register: (register contents) + 1 = 1
5. T2 increments the value of i in the register: (register contents) + 1 = 1
6. T1 stores the value of the register in memory : 1
7. T2 stores the value of the register in memory : 1
8. Integer i = 1

The final value of i is 1 instead of the expected result of 2. This occurs because the increment operations of the second case are non-atomic. Atomic operations are those that cannot be interrupted while accessing some resource, such as a memory location. In the first case, T1 was not interrupted while accessing the variable i, so its operation was atomic.

For another example, consider the following two tasks, in pseudocode:

```
global integer A = 0;

// increments the value of A and print "RX"
// activated whenever an interrupt is received from the serial controller
task Received()
{
    A = A + 1;
    print "RX";
}

// prints out only the even numbers
// is activated every second
task Timeout()
{
    if (A is divisible by 2)
    {
        print A;
    }
}
```

Output would look something like:

```
     0
     0
     0
     RX
     RX
     2
     RX
     RX
     4
     4
```

Now consider this chain of events, which might occur next:

1. timeout occurs, activating task `Timeout`
2. task `Timeout` evaluates `A` and finds it is divisible by 2, so elects to execute the "print A" next.
3. data is received on the serial port, causing an interrupt and a switch to task `Received`
4. task `Received` runs to completion, incrementing A and printing "RX"
5. control returns to task `Timeout`
6. task timeout executes print A, using the current value of A, which is 5.

Mutexes are used to address this problem in concurrent programming.

3.2 Real life examples

3.2.1 File systems

In file systems, two or more programs may "collide" in their attempts to modify or access a file, which could result in data corruption. File locking provides a commonly-used solution. A more cumbersome remedy involves reorganizing the system in such a way that one unique process (running a daemon or the like) has exclusive access to the file, and all other processes that need to access the data in that file do so only via interprocess communication with that one process (which of course requires synchronization at the process level).

A different form of race condition exists in file systems where unrelated programs may affect each other by suddenly using up available resources such as disk space (or memory, or processor cycles). Software not carefully designed to anticipate and handle this rare situation may then become quite fragile and unpredictable. Such a risk may be overlooked for a long time in a system that seems very reliable. But eventually enough data may accumulate or enough other software may be added to critically destabilize many parts of a system. Probably the best known example of this occurred with the near-loss of the Mars Rover "Spirit" not long after landing, but this is a commonly overlooked hazard in many computer systems. A solution is for software to request and reserve all the resources it will need before beginning a task; if this request fails then the task is postponed, avoiding the many points where failure could have occurred. (Alternately, each of those points can be equipped with error handling, or the success of the entire task can be verified afterwards, before continuing on.) A more common but incorrect approach is to simply verify that enough disk space (for example) is available before starting a task; this is not adequate because in complex systems the actions of other running programs can be unpredictable.

3.2.2 Networking

In networking, consider a distributed chat network like Internet relay chat (IRC), where a user acquires channel-operator privileges in any channel he starts. If two users on different servers, on different ends of the same network, try to start the same-named channel at the same time, each user's respective server will grant channel-operator privileges to each user, since neither server will yet have received the other server's signal that it has allocated that channel. (Note that this problem has been largely solved by various IRC server implementations.)

In this case of a race condition, the concept of the "shared resource" covers the state of the network (what channels exist, as well as what users started them and therefore have what privileges), which each server can freely change as long as it signals the other servers on the network about the changes so that they can update their conception of the state of the network. However, the latency across the network makes possible the kind of race condition described. In this case, heading off race conditions by imposing a form of control over access to the shared resource—say, appointing one server to control who holds what privileges—would mean turning the distributed network into a centralized one (at least for that one part of the network operation). Where users find such a solution unacceptable, a pragmatic solution can have the system:

1. Recognize when a race condition has occurred; and
2. Repair the ill effects.

3.2.3 Life-Critical Systems

Software flaws in life-critical systems can be disastrous. Race conditions were among the flaws in the Therac-25 radiation therapy machine, which led to the death of five patients and injuries to several more. Another example is the Energy Management System provided by GE Energy and used by Ohio-based FirstEnergy Corp. (and by many other power facilities as well). A race condition existed in the alarm subsystem; when three sagging power lines were tripped simultaneously, the condition prevented alerts from being raised to the monitoring technicians, delaying their awareness of the problem. This software flaw eventually led to the North American Blackout of 2003. (GE Energy later developed a software patch to correct the previously undiscovered error.)

3.3 Computer Security

A specific kind of race condition involves checking for a predicate (e.g. for authentication), then acting on the predicate, while the state can change between the *time of check* and the *time of use*. When this kind of bug exists in security-conscious code, a security vulnerability called a time-of-check-to-time-of-use (*TOCTTOU*) bug is created.

3.4 Asynchronous Finite State Machines

Even after ensuring that single bit transitions occur between states, the asynchronous machine will fail if multiple inputs change at the same time. The solution to this is to design a machine so that each state is sensitive to only one input change.

4.0 Conclusion

In this unit you have learnt about race condition, its cause, life examples and computer examples of race condition. In the next unit(s) you will be exposed to ways of preventing the occurrence of race condition especially unexpected race condition.

5.0 Summary

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called race condition. To guide against race condition, there is need for a form synchronization of processes. Such situations occur frequently in operating systems as different parts of a system manipulate resources and we want the changes not to interfere with one another. A major portion of this module is concerned with process synchronization and coordination issues.

6.0 Tutor-Marked Assignment

1. What do you understand by the term race condition?

2. Describe any two life examples of race condition.

3. Briefly explain reason why it is desirable to avoid race condition.

7.0 References/Further Reading

1. M. Herlihy, V. Luchangco and M. Moir. "Obstruction-Free Synchronization: Double-Ended Queues as an Example." 23rd International Conference on Distributed Computing Systems, 2003, p.522.

2. F. Fich, D. Hendler, N. Shavit. "Impossibility and universality results for wait-free synchronization" Proceedings of the seventh annual ACM Symposium on Principles of distributed computing, 1988, pp. 276 - 290.

3. F. Fich, D. Hendler, N. Shavit. "On the inherent weakness of conditional synchronization primitives." 23rd Annual ACM Symposium on Principles of Distributed Computing, 2004, pp. 80-87.

4. W. Scherer and M. Scott. "Advanced Contention Management for Dynamic Software Transactional Memory." 24th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 2005, pp. 240-248.

Module 4: Process Synchronization

Unit 2: Synchronization

Table of Contents

1.0 Introduction

Synchronization refers to one of two distinct, but related concepts: synchronization of processes, and synchronization of data. **Process synchronization** refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of action while **Data synchronization** refers to the idea of keeping multiple copies of a dataset in coherence with one another, or to maintain data integrity. Process synchronization primitives are commonly used to implement data synchronization. In this unit you are going to be introduced to process synchronization.

2.0 Objectives

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

- Define process synchronization
- Describe non-blocking synchronization
- Explain the motivation for non-blocking synchronization
- Describe various types of non-blocking synchronization algorithms

3.0 Main body

3.1 Process synchronization

Process synchronization refers to the coordination of simultaneous threads or processes to complete a task in order to get correct runtime order and avoid unexpected race conditions.

There are many types of synchronization:

- barrier
- lock/semaphore
- thread join
- mutex
- non-blocking synchronization
- synchronous communication operations (see: Comparison of synchronous and asynchronous signalling)

A **synchronization point** is the location, in a process or collection of threads or processes, where the synchronization occurs.

3.2 Non-blocking synchronization

Non-blocking synchronization ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion. Non-blocking is often called *lock-free*: an algorithm with guaranteed system-wide progress. However, since 2003, the term has been weakened to only prevent progress-blocking interactions with a pre-emptive scheduler.

In modern usage, therefore, an algorithm is *non-blocking* if the suspension of one or more threads will not stop the potential progress of the remaining threads. They are designed to avoid requiring a critical section. Often, these algorithms allow multiple processes to make progress on a problem without ever blocking each other. For some operations, these algorithms provide an alternative to locking mechanisms.

3.2.1 Motivation

The traditional approach to multi-threaded programming is to use locks to synchronize access to shared resources. Synchronization primitives such as mutexes, semaphores, and critical sections are all mechanisms by which a programmer can ensure that certain sections of code do not execute concurrently if doing so would corrupt shared memory structures. If one thread attempts to acquire a lock that is already held by another thread, the thread will block until the lock is free.

Blocking a thread, though, is undesirable for many reasons. An obvious reason is that while the thread is blocked, it cannot accomplish anything. If the blocked thread is performing a high-priority or real-time task, it is highly undesirable to halt its progress. Other problems are less obvious. Certain interactions between locks can lead to error conditions such as deadlock, livelock, and priority inversion. Using locks also involves a trade-off between coarse-grained locking, which can significantly reduce opportunities for parallelism, and fine-grained locking, which requires more careful design, increases overhead and is more prone to bugs.

Non-blocking algorithms are also safe for use in interrupt handlers: even though the preempted thread cannot be resumed, progress is still possible without it. In contrast, global data structures protected by mutual exclusion cannot safely be accessed in a handler, as the preempted thread may be the one holding the lock.

Non-blocking synchronization has the potential to prevent priority inversion, as no thread is forced to wait for a suspended thread to complete. However, as livelock is still possible in the modern definition, threads have to wait when they encounter contention; hence, priority inversion is still possible depending upon the contention management system used. Lock-free algorithms, below, avoid priority inversion.

3.2.2 Implementation

Non-blocking algorithms use atomic read-modify-write primitives that the hardware must provide, the most not able of which is compare and swap (CAS). Ultimately, all synchronizing algorithms must use these; however, critical sections are almost always implemented using standard interfaces over these primitives. Until recently, all non-blocking algorithms had to be written "natively" with the underlying primitives to achieve acceptable performance. However, the emerging field of software transactional memory promises standard abstractions for writing efficient non-blocking code.

Much research has also been done in providing basic data structures such as stacks, queues, sets, and hash tables. These allow programs to easily exchange data between threads asynchronously.

3.2.3 Wait-freedom

Wait-freedom is the strongest non-blocking guarantee of progress, combining guaranteed system-wide throughput with starvation-freedom. An algorithm is wait-free if every operation has a bound on the number of steps it will take before completing.

It was shown in the 1980s that all algorithms can be implemented wait-free, and many transformations from serial code, called *universal constructions*, have been demonstrated. However, the resulting performance does not in general match even naive blocking designs. It has also been shown that the widely-available atomic *conditional* primitives, compare-and-swap, cannot provide starvation-free implementations of many common data structures without memory costs growing linearly in the number of threads. Wait-free algorithms are therefore rare, both in research and in practice.

3.2.4 Lock-freedom

Lock-freedom allows individual threads to starve but guarantees system-wide throughput. An algorithm is lock-free if every step taken achieves global progress (for some sensible definition of progress). All wait-free algorithms are lock-free.

In general, a lock-free algorithm can run in four phases: completing one's own operation, assisting an obstructing operation, aborting an obstructing operation, and waiting. Completing one's own operation is complicated by the possibility of concurrent assistance and abortion, but is invariably the fastest path to completion.

The decision about when to assist, abort or wait when an obstruction is met is the responsibility of a *contention manager*. This may be very simple (assist higher priority operations, abort lower priority ones), or may be more optimized to achieve better throughput, or lower the latency of prioritized operations.

Correct concurrent assistance is typically the most complex part of a lock-free algorithm, and often very costly to execute: not only does the assisting thread slow down, but thanks to the mechanics of shared memory, the thread being assisted will be slowed, too, if it is still running.

3.2.5 Obstruction-freedom

Obstruction-freedom is possibly the weakest natural non-blocking progress guarantee. An algorithm is obstruction-free if at any point, a single thread executed in isolation (i.e. with all obstructing threads suspended) for a bounded number of steps will complete its operation. All lock-free algorithms are obstruction-free.

Obstruction-freedom demands only that any partially-completed operation can be aborted and the changes made rolled back. Dropping concurrent assistance can often result in much simpler algorithms that are easier to validate. Preventing the system from continually live-locking is the task of a contention manager.

Recent research has yielded a promising practical contention manager, whimsically named *Polka*, combining exponential backoff with "priority accumulation". As an operation progresses, it gains "priority"; when an operation is obstructed by another with higher priority, it will back off, with backoff intervals increasing exponentially. Each backoff increases the operation's priority; only when its priority is greater than that of its obstructor will it abort it. Aborted operations retain their former priority, giving their next attempt a greater chance of success.

Polka achieves good throughput in benchmarks because it minimizes both wasted effort, by prioritizing long transactions, and memory interconnect contention, using exponential backoff. This can inform other parallel algorithms, such as lock-free ones, to achieve greater throughput in the common case.

4.0 Conclusion

In this unit you have been introduced to synchronization, particularly non-blocking synchronization.

5.0 Summary

6.0 Tutor-Marked Assignment

1. Define Synchronization
2. What is the need for process synchronization
3. Describe any lock-free non-blocking synchronization algorithm
4. When is an algorithm wait-free?
5. Enumerate the various phases in which a lock-free algorithm can run.

7.0 References/Further Reading

1. M. Herlihy, V. Luchangco and M. Moir. "Obstruction-Free Synchronization: Double-Ended Queues as an Example." 23rd International Conference on Distributed Computing Systems, 2003, p.522.
2. F. Fich, D. Hendler, N. Shavit. "Impossibility and universality results for wait-free synchronization" Proceedings of the seventh annual ACM Symposium on Principles of distributed computing, 1988, pp. 276 - 290.
3. F. Fich, D. Hendler, N. Shavit. "On the inherent weakness of conditional synchronization primitives." 23rd Annual ACM Symposium on Principles of Distributed Computing, 2004, pp. 80-87.
4. W. Scherer and M. Scott. "Advanced Contention Management for Dynamic Software Transactional Memory." 24th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 2005, pp. 240-248.

Module 4: Process Synchronization

Unit 3: Mutual Exclusion

Table of Contents

1.0 Introduction

In the previous units of this module you have been introduced you to some pertinent concepts in process synchronization. This unit will further expose you to another important concept in process synchronization which is mutual exclusion. It is an algorithm that is often used in concurrent programming to avoid the simultaneous use of a common resource by pieces of computer code known as critical section (this will be discussed in this next unit.

2.0 Objectives

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

- Describe what you understand by mutual exclusion
- Describe ways to enforce mutual exclusion

3.0 Main Body

3.1 What is Mutual Exclusion?

Mutual exclusion (often abbreviated to **mutex**) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

Examples of such resources are fine-grained flags, counters or queues, used to communicate between code that runs concurrently, such as an application and its interrupt handlers. The problem is acute because a thread can be stopped or started at any time.

To illustrate: suppose a section of code is mutating a piece of data over several program steps, when another thread, perhaps triggered by some unpredictable event, starts executing. If this second thread reads from the same piece of data, the data, in the process of being overwritten, is in an inconsistent and unpredictable state. If the second thread tries overwriting that data, the ensuing state will probably be unrecoverable. These critical sections of code accessing shared data must therefore be protected, so that other processes which read from or write to the chunk of data are excluded from running.

A **mutex** is also a common name for a program object that negotiates mutual exclusion among threads, also called a lock.

Mutual exclusion is one of the control problems faced in the case of competing processes. The enforcement of mutual exclusion creates two additional control problems; deadlock and starvation (Stallings)

Mutual exclusion has two levels of concurrency:

1. Concurrency among processes
2. Concurrency among activities (threads) with a single process.

If concurrent processes or activities do not access common resources, there is no problem, but there s a problem if they do. A solution to this problem is to keep the critical activities sequential rather than concurrent. This solution is not always practical.

Problems in achieving mutual exclusion include lockstep, loss of mutual exclusion, deadlock and indefinite postponement.

3.2 Enforcing Mutual Exclusion

There exist both software and hardware solutions for enforcing mutual exclusion. The different solutions are presented below.

3.2.1 MUTUAL EXCLUSION HARDWARE APPROACH (TEST-AND-SET)

On a uniprocessor system the common way to achieve mutual exclusion is to disable interrupts for the smallest possible number of instructions that will prevent corruption of the shared data structure, the critical section. This prevents interrupt code from running in the critical section.

In a computer in which several processors share memory, an indivisible test-and-set of a flag is used in a tight loop to wait until the other processor clears the flag. The test-and-set performs both operations without releasing the memory bus to another processor. When the code leaves the critical section, it clears the flag. This is called a "spinlock" or "busy-wait."

The test and set instruction can be defined as follows:

```
function  testset (var i: integer) : boolean;
```

```
begin
   if i =0;   then
       begin
          i := 1
          testset : = true
        end
    else testset := false
end.
```

The instruction test the value of its argument *i*. If the value is 0, then it replaces it by 1 and returns true. otherwise, the value is not changed and false is returned. The entire testset function is carried out automatically; that is, it is not subject to interruption

Some computers have similar indivisible multiple-operation instructions, e.g., compare-and-swap, for manipulating the linked lists used for event queues and other data structures commonly used in operating systems.

3.2.2 MUTUAL EXCLUSION: SOFTWARE APPROACH

Beside the hardware supported solution, some software solutions exist that use "busy-wait" to achieve the goal. Examples of these include:

- Dekker's algorithm
- Peterson's algorithm
- Lamport's bakery algorithm
- The Black-White Bakery Algorithm
- Semaphores
- Monitor (synchronization)
- Message passing

Most classical mutual exclusion methods attempt to reduce latency and busy-waits by using queuing and context switches. Some claim that benchmarks indicate that these special algorithms waste more time than they save.

Software approaches can be implemented for concurrent processes that execute on a single processor or a multiprocessor machine with shared main memory. These approaches usually assume elementary mutual exclusion at the memory access level. That is, simultaneous accesses (reading and/or writing) to the same location in main memory are serialized by some sort of memory arbiter, although the order of access granting is not specified ahead of time. Beyond this, no support at the hardware, operating system, or programming-language level is assumed.

Peterson's Algorithm provided a simple and elegant solution. That mutual exclusion is preserved is easily shown. Consider process P0. Once it has set flag [0] to true, P1 cannot enter its critical section. If P1 already is in its critical section, then flag [1] = true and P0 is blocked from entering its critical section. On the other hand, mutual blocking is prevented. Suppose that P0 is blocked in its while loop. this means that flag [1] is true and turn = 1. P0 can enter its critical section when either flag [1] becomes false or turn becomes 0. Now consider three exhaustive cases:

1. P1 has no interest in its critical section. This case is impossible, because it implies flag [1] = false.
2. P1 is waiting for its critical section. This case is also impossible, because if turn = 1, P1 is able to enter its critical section.

P1 is using its critical section repeatedly and therefore monopolizing access to it. This cannot happen, because P1 is obliged to give P0 an opportunity by setting turn to 0 before each attempt to enter its critical section.

The algorithm for two processes is presented below.

```
var flag: array [0..1] of boolean;
      turn: 0..1;
procedure P0;
begin
  repeat
      flag [0] := true
      turn : = 1
      while flag [1] and turn = 1 do{ nothing };
      <critical section>;
       flag [0] : = false;
      <remainder>
   forever
end;

procedure P1;
begin
  repeat
      flag [1] := true
      turn : = 0;
      while flag [0] and turn = 0 do{ nothing };
      <critical section>;
      flag [1] : = false;
      <remainder>
   forever
end;
begin
       flag [0] := false;
       flag [1] := false;
       turn := 1;
       parbegin
            P0;P1
```

```
        parend
    end.
```

This algorithm is easily generalized to the case of *n* processes.

Many forms of mutual exclusion have side-effects. For example, classic semaphores permit deadlocks, in which one process gets a semaphore, another process gets a second semaphore, and then both wait forever for the other semaphore to be released. Other common side-effects include starvation, in which a process never gets sufficient resources to run to completion, priority inversion, in which a higher priority thread waits for a lower-priority thread, and "high latency" in which response to interrupts is not prompt.

Much research is aimed at eliminating the above effects, such as by guaranteeing non-blocking progress. No perfect scheme is known.

4.0 Conclusion

This unit has taken you through the concept of mutual exclusion. In the next unit, you will see how it can be used to solve critical-section problem.

5.0 Summary

Mutual Exclusion, as you have seen in this unit can be implemented by hardware or software. The hardware features can make the programming task easier and improve system efficiency. The software forms of mutual exclusion especially the classic semaphores have side-effects one of which is that it may lead to deadlocks.

6.0 Tutor-Marked Assignment

1. What do you understand by mutual exclusion
2. What is it used for?
3. State some of the software methods of implementing mutual exclusion algorithm
4. What do you understand by busy-wait?

7.0 References/Further Reading

1. Michel Raynal: *Algorithms for Mutual Exclusion*, MIT Press, ISBN 0-262-18119-3
2. Sunil R. Das, Pradip K. Srimani: *Distributed Mutual Exclusion Algorithms*, IEEE Computer Society, ISBN 0-8186-3380-8
3. Thomas W. Christopher, George K. Thiruvathukal: *High-Performance Java Platform Computing*, Prentice Hall, ISBN 0-13-016164-0
4. Gadi Taubenfeld, *Synchronization Algorithms and Concurrent Programming*, Pearson/Prentice Hall, ISBN 0-13-197259-6

Module 4: Process Synchronization

Unit 4: Critical Section Problem

Table of Contents

1.0 Introduction

Consider a system consisting of n processes {P_0, P_1,…, P_{n-1}}. Each process has a segment of code called critical section, in which the processes may be changing common variables, updating a table, writing a file, etc. The important feature of the system is that, when one process is executing, in its critical section, no other process is to be allowed to execute in its critical section. Therefore the execution of the critical section by the processes is mutually exclusive in time. The critical section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

```
do {
        entry section

                critical section

        exit section
```

```
            remainder section

    } while(1);
```

Figure 1.1 General structure of a typical process P_i

A solution to the critical section problem must satisfy the following three requirements:

1. **Mutual Exclusion**: if process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical sectionnext, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting**: there exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Based on these three requirements, we will discuss some solutions to critical section problem in this unit.

2.0 Objectives

At the end of this unit you should be able to:

- Explain the critical section problem
- State the different levels of critical section
- Define semaphores
- Define monitors
- Distinguish between monitors and semaphores

3.0 Main Body

3.1 The Critical Section Problem

A **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will only have to wait a fixed time to enter it (i.e. bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.

By carefully controlling which variables are modified inside and outside the critical section (usually, by accessing important state only from within), concurrent access to that state is prevented. A critical section is typically used when a multithreaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure a shared resource, for example a printer, can only be accessed by one process at a time.

How critical sections are implemented varies among operating systems.

The simplest method is to prevent any change of processor control inside the critical section. On uni-processor systems, this can be done by disabling interrupts on entry into the critical section, avoiding system calls that can cause a context switch while inside the section and restoring interrupts to their previous state on exit. Any thread of execution entering any critical section anywhere in the system will, with this implementation, prevent any other thread, including an interrupt, from getting the CPU and therefore from entering any other critical section or, indeed, any code whatsoever, until the original thread leaves its critical section.

This brute-force approach can be improved upon by using semaphores. To enter a critical section, a thread must obtain a semaphore, which it releases on leaving the section. Other threads are prevented from entering the critical section at the same time as the original thread, but are free to gain control of the CPU and execute other code, including other critical sections that are protected by different semaphores.

Some confusion exist in the literature about the relationship between different critical sections in the same program. In general, a resource that must be protected from concurrent access may be accessed by several pieces of code. Each piece must be guarded by a common semaphore. Is each piece now a critical section or are all the pieces guarded by the same semaphore in aggregate a single critical section? This confusion is evident in definitions of a critical section such as "... a piece of code that can only be executed by one process or thread at a time". This only works if all access to a protected resource is contained in one "piece of code", which requires either the definition of a piece of code or the code itself to be somewhat contrived.

3.1.1 Application Level Critical Sections

Application-level critical sections reside in the memory range of the process and are usually modifiable by the process itself. This is called a user-space object because the program run by the user (as opposed to the kernel) can modify and interact with the object. However the functions called may jump to kernel-space code to register the user-space object with the kernel.

Example Code For Critical Sections with POSIX pthread library

```
/* Sample C/C++, Unix/Linux */
#include <pthread.h>
pthread_mutex_t cs_mutex = PTHREAD_MUTEX_INITIALIZER; /*
This is the critical section object*/

/* Enter the critical section -- other threads are locked
out */
```

```
pthread_mutex_lock( &cs_mutex );

/* Do some thread-safe processing! */

/*Leave the critical section -- other threads can now
pthread_mutex_lock()  */
pthread_mutex_unlock( &cs_mutex);
```

Example Code For Critical Sections with Win32 API

```
/* Sample C/C++, Win9x/NT/ME/2000/XP, link to kernel32.dll
*/
#include <windows.h>
CRITICAL_SECTION cs; /* This is the critical section object
-- once initialized, it cannot
                        be moved in memory */

/* Initialize the critical section -- This must be done
before locking */
InitializeCriticalSection(&cs);

/* Enter the critical section -- other threads are locked
out */
EnterCriticalSection(&cs);

/* Do some thread-safe processing! */

/* Leave the critical section -- other threads can now
EnterCriticalSection() */
LeaveCriticalSection(&cs);

/* Release system object when all finished -- usually at
the end of the cleanup code */
DeleteCriticalSection(&cs);
```

Note that on Windows NT (not 9x/ME), the function **TryEnterCriticalSection()** can be used to attempt to enter the critical section. This function returns immediately so that the thread can do other things if it fails to enter the critical section (usually due to another thread having locked it). Note that the use of a CriticalSection is not the same as a Win32 Mutex, which is an object used for *inter-process* synchronization. A Win32 CriticalSection is for *inter-thread* synchronization (and is much faster as far as lock times), however it cannot be shared across processes.

3.1.2 Kernel Level Critical Sections

Typically, critical sections prevent process and thread migration between processors by interrupts. Also, pre-emption of processes and threads is prevent by interrupts.

Critical sections often allow nesting. Nesting allows multiple critical sections to be entered and exited at little cost.

If the scheduler interrupts the current process or thread in a critical section, the scheduler will either allow the process or thread to run to completion of the critical section, or it will schedule the process or thread for another complete quantum. The scheduler will not migrate the process or thread to another processor, and it will not schedule another process or thread to run while the current process or thread is in a critical section.

Similarly, if an interrupt occurs in a critical section, the interrupt's information is recorded for future processing, and execution is returned to the process or thread in the critical section. Once the critical section is exited, and in some cases the scheduled quantum completes, the pending interrupt will be executed.

Since critical sections may execute only on the processor on which they are entered, synchronization is only required within the executing processor. This allows critical sections to be entered and exited at almost zero cost. No interprocessor synchronization is required, only instruction stream synchronization. Most processors provide the required amount of synchronization by the simple act of interrupting the current execution state. This allows critical sections in most cases to be nothing more than a per processor count of critical sections entered.

Performance enhancements include executing pending interrupts at the exit of all critical sections and allowing the scheduler to run at the exit of all critical sections. Furthermore, pending interrupts may be transferred to other processors for execution.

Critical sections should not be used as a long-lived locking primitive. They should be short enough that the critical section will be entered, executed, and exited without any interrupts occurring, from neither hardware much less the scheduler.

3.2 Semaphores

The first major advance in dealing with the problems of concurrent processes came in 1965 with Dijkstra's treatise. The fundamental principle is this: two or more processes can cooperate by means of simple signal, such that a process can be forced to stop at a specified place until it has received a specified signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals. For signalling, special variables called semaphores are used. To transmit a signal via semaphores, a process executes the primitive signal(s). To receive a signal via semaphore s, a process executes the primitive wait(s); if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which three operations are defined:

1. A semaphore may be initialized to a non-negative value
2. The wait operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked
3. The signal operation increments the semaphore value. If the value is not positive, then a process blocked by wait operation is unblocked.

Other than these three operations, there is no way to inspect or manipulate semaphores.

Semaphores are operated on by a signal operation, which increments the semaphore value and the wait operation, which decreases it. The initial value of semaphore indicates the number of wait operations that can be performed on the semaphore. Thus:

$V= I - W + S$

where I is the initial value of the semaphore

W is the number of completed wait operations performed on the semaphore

S is the number of signal operations performed on it

V is the current value of the semaphore (which must be greater than or equal to zero).

As V is > 0 then I- W+ S > 0, which gives

$$I + S > W$$

or

$$W < I + S$$

Thus, the number of wait operations must be less than or equal to the initial value of the semaphore, plus the number of signal operations. A binary semaphore will have an initial value of 1 (I = 1), thus:

$$W< S+ 1$$

In mutual exclusion, *waits* always occur before signals, as *waits* happen at the start of a critical piece of code, with a signal at the end of it. The above equation states that no more than one *wait* may run to completion before a signal has been performed. Thus no more than one process may enter the critical section at a time as required

3.2.1 The Problem with Semaphores

Semaphores work but programmers still need to code carefully to ensure mutual exclusion and that synchronisation operate correctly.

3.2.2 Language Constructs

The problem with semaphores is that they are too **low level** in nature: they are similar to doing mutual exclusion and synchronisation in assembly language using **goto**'s. They have no **structure**. They are also damnably hard to prove correct and near to impossible to test!

What is needed are high-level language constructs that enforce the necessary discipline. The two main contenders for this job are:

- Critical Regions, and more usefully, **Conditional Critical Regions**
- Monitors

3.3 Monitors

Monitors are a common high-level synchronization tool which solve some of the problems associated with semaphores.

Monitors are actually a much nicer way of implementing mutual exclusion than semaphores. One of the reasons for this is that the code that implements mutual exclusion is all in one place, the monitor. With semaphores, code can distributed all over the place in the form of wait and signal semaphore function calls.

Additionally, it is next to impossible to setup a monitor incorrectly. On the other hand with semaphores it is quite common to do a wait (B) when you should have done a wait (C). Simple little mistakes are easy to make with semaphores.

3.3.1 Process Synchronization with Monitors

Process synchronization with monitors is implemented in much the same way as it is implemented with semaphores. However, with monitors you use condition variables rather than semaphores.

For this reason, it is important that you realize the difference between semaphores and condition variables. This is made more difficult because

- both semaphores and condition variables use wait and signal as the valid operations,
- the purpose of both is somewhat similar, and
- they are actually quite different.

The main difference is the operation of signal. With a semaphore the signal operation actually increments the value of the semaphore. So, if there are not any processes blocked on the semaphore, the signal will be "remembered" by the incrementing of the semaphore.

The signal function on a Monitor’s condition variable is different. If there are no processes blocked on the condition variable then the signal function does nothing. The signal is not remembered. In order to remember "empty signals", you have to use some other form of variables. The good part of this is that using other variables within a monitor is simple because we can be assured that mutual exclusion is being implemented.

4.0 Conclusion

In this unit, you learnt some of the methods for synchronizing co-operating processes and how these methods are implemented as well as the advantages and disadvantages of each approach. In the next unit we will discuss some of the classic problems of synchronization.

5.0 Summary

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided. One solution is to ensure that a critical section of code is in use by only one process or thread at a time. Different algorithms exist for solving the critical-section problem, with the assumption that only storage interlock is available.

The main disadvantage of these user-coded solutions is that they all require busy waiting. Semaphores overcome this difficulty. Semaphores can be used to solve various synchronization problems and can be implemented efficiently. However, there are some problems with using semaphores too.

Monitors overcome the problem with using semaphores because it is next to impossible to setup a monitor incorrectly.

6.0 Tutor-Marked Assignment

1. Enumerate the requirements that must be satisfied by a solution to the critical section problem

2. What do you understand by critical section? When is it used?

3. Compare Application-level critical section and kernel-level critical section.

4. What do you understand by semaphore?

5. What are the problems with using semaphores to implement mutual exclusion and how does monitor overcome these problems?

6. What are monitors?

7. Can monitors be incorrectly setup? Explain

7.0 References/Further Reading

1. Silberschatz, Abraham; Peterson, James L. (1988). *Operating Systems Concepts*. Addison-Wesley. ISBN 0-201-18760-4.

2. *The Little Book of Semaphores,* by Allen B. Downey, http://greenteapress.com/semaphores
3. Dijkstra, E. W. (1971, June). Hierarchical ordering of sequential processes. Acta Informatica 1(2): 115-138.
4. *Modern Operating Systems (2nd Edition)* by Andrew S. Tanenbaum (ISBN 0-13-031358-0)

Module 5: Deadlocks

Unit 1: Deadlock Characterization

Table of Contents

1.0 Introduction

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available, at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called deadlock. We have already mentioned this briefly in module 4 in connection with semaphores.

In this module, you will be taken through methods that an operating system can use to prevent or deal with deadlocks

Objectives

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

- Define deadlock
- State the necessary conditions for deadlock to occur
- Describe Resource-Allocation graph
- Explain how it can be used to describe deadlocks
- Describe some of the methods for handling deadlocks.

3.0 Main Body

3.1 System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resource are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type *CPU* has two instances. Similarly, the resource type *printer* may have five instances.

A process must request a resource before using it, and must release the resource after using it. a process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. i.e. a process cannot request three printers if the system has only two.

Under normal mode of operation, a process may utilize a resource in only the following sequence.

1. **Request**: If the request cannot be granted immediately (for example, if the resource is been used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use**: The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer)
3. **Release**: The process releases the resource.

Request and release of resources can be accomplished through the *wait* and *signal* operations on semaphores. Therefore, for each use, the operating system checks to make sure that the using process has requested and been allocated the resource. A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.

To illustrate deadlock state, consider a system with three tape drives. Suppose each of three processes holds one of these tape drives. If each process now requests another tape drive, the three processes will be in deadlock. Each is waiting for the event “tape drive is released” which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource type. E.g. Consider a system with one printer and one tape drive. Suppose that process P_1 is holding the tape drive and process P_2 is holding the printer. If P_1 requests the printer and P_2 requests the tape drive, a deadlock occurs.

A deadlock is also called a *deadly embrace.*

Deadlocks occur most commonly in multitasking and client/server environments. Therefore, a programmer who is developing multithreaded applications must pay particular attention to this problem: Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

3.2 Deadlock Characterization

In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we shall describe to you features that characterized deadlocks.

3.2.1 Necessary Conditions

A deadlock situation can arise if the following conditions hold simultaneously in a system:

1. **Mutual exclusion condition:** At least one resource must be held in a non-sharable mode; i.e. only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold-and-wait condition:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No-preemption condition:** Resources cannot be preempted; i.e. only a process holding a resource may voluntarily release the resource after completing its task.
4. **Circular-wait condition:** two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds. i.e. A set ($P_{0,}$ $P_{1,}$ …, P_n) of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by $P_{0.}$

Deadlock only occurs in systems where all these four conditions hold. You should note that the hold-and-wait condition leads to the circular-wait condition implies. So, the four conditions are not completely independent.

3.2.2 Resource-Allocation (R-A) Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set vertices V is partitioned into two different types of nodes $P = \{P_1, P_{2,} \ldots, P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_{2,} \ldots, R_m\}$, the set consisting of all resource types in the system.

A directed edge from process P_i, to resource type $R_{j,}$ is denoted by $P_i \rightarrow R_j$; it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow P_i$ is called an **assignment edge**.

Pictorially, each process P_i is represented as a circle, and each resource type R_j as a square. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the square. You should note that a request edge points only to the square whereas an assignment edge must also designate one of the dots in the square.

When a process P_i request an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled; the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted.

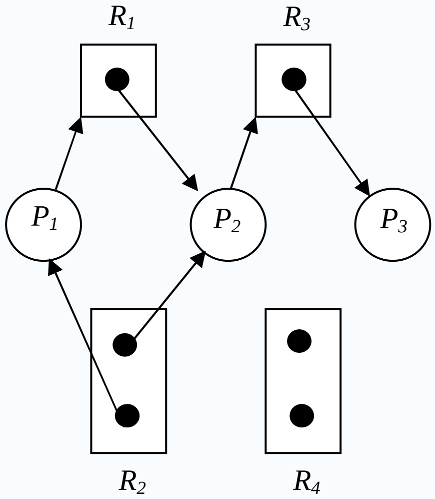


Figure 3.1: Resource-Allocation Graph (RAG)

The resource-allocation graph shown in figure 3.1 depicts the following situation:

- The sets *P*, *R* and *E*
    - $P = (P_1, P_{2,}, P_3)$
    - $R = (R_1, R_{2,}, R_3, R_4)$
    - $E = \{P_1 \rightarrow R_{1,} P_2 \rightarrow R_{3,} R_1 \rightarrow P_{2,} R_2 \rightarrow P_{2,} R_2 \rightarrow P_{1,} R_3 \rightarrow P_3\}$
- Resource instances:
    - One instance of resource type R_1
    - Two instances of resource type R_2
    - One instance of resource type R_3
    - Two instances of resource type R_4
- Process states:
    - Process is P_1 is holding an instance of resource type R_2, and is waiting for an instance of resource type R_1.

- Process is P_2 is holding an instance of resource type R_1 and R_2, and is waiting for an instance of resource type R_3.
- Process is P_3 is holding an instance of resource type R_3.

Given the definition of a RAG, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that deadlock has occurred. If the cycle involves only a set resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is both a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, let us return to the R-A graph depicted in figure 3.1 above. Suppose process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph (see figure 3.2). At this point, two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

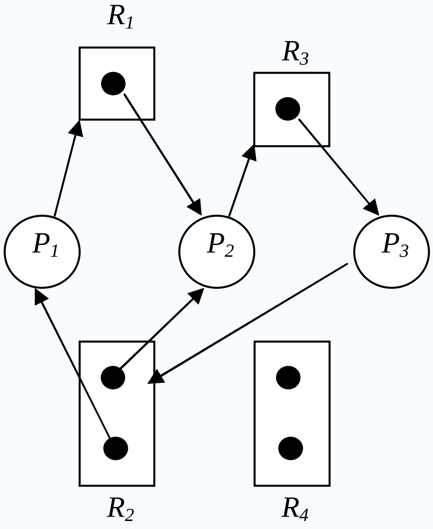


Figure 3.2: Resource-Allocation Graph with a Deadlock

Processes P_1, P_2, and P_3 are deadlocked. Process P_2 is waiting for resource R_3, which is held by process P_3. Process P_3, on the other hand, is waiting for either process P_1 or P_2 to release resource R_2. In addition, process P_1 is waiting for P_2 to release resource R_1.

Now consider the R-A graph in figure 3.3.

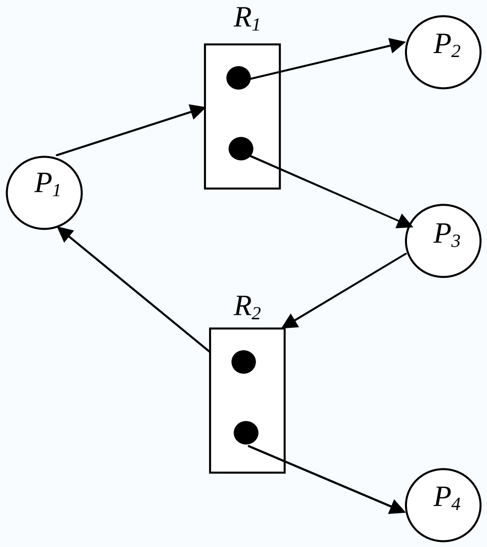


Figure 3.2: Resource-Allocation Graph with a cycle but no Deadlock

In this example, you also have a cycle:

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_4 and that resource could then be allocated to P_3, breaking the cycle

Conclusively, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state. This observation is important when you deal with deadlock problem.

3.3 Methods for Handling Deadlocks

Principally, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter add state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This method is use by most operating system including UNIX.

We shall elaborate briefly on each method. Then in the later units, we shall present you detailed algorithms.

To ensure that deadlocks never occur, the system can use either deadlock-prevention or a deadlock-avoidance scheme. Deadlock-prevention is a set of methods for ensuring that at least one of the necessary conditions (unit 2) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. These methods are discussed in unit 2.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently allocated to each process, and the future requests and releases of each process. These schemes are discussed in later units.

If system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock (if a deadlock has indeed occurred). This issue is discussed in unit 2.

If the system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in the deterioration of the system performance, because resources are being held by processes that cannot run, and because more and more processes, as they make requests for resources, enter a deadlock state. Eventually, system will stop functioning and will need to be restarted manually.

Although this method does not seem to be a viable approach to the deadlock problem, it is nevertheless used in some operating systems. In many systems, deadlocks occur infrequently like once in a year. Therefore, this method is cheaper than the costly deadlock-prevention, deadlock-avoidance, or deadlock-detection and recovery methods that must be used constantly.

4.0 Conclusion

In this unit, you have been exposed to the concept of deadlock problem, necessary conditions for its occurrence and some of the ways it can be handled when it occurs. In subsequent units, you will be taken through some specific algorithms on each of the methods for handling deadlock problems.

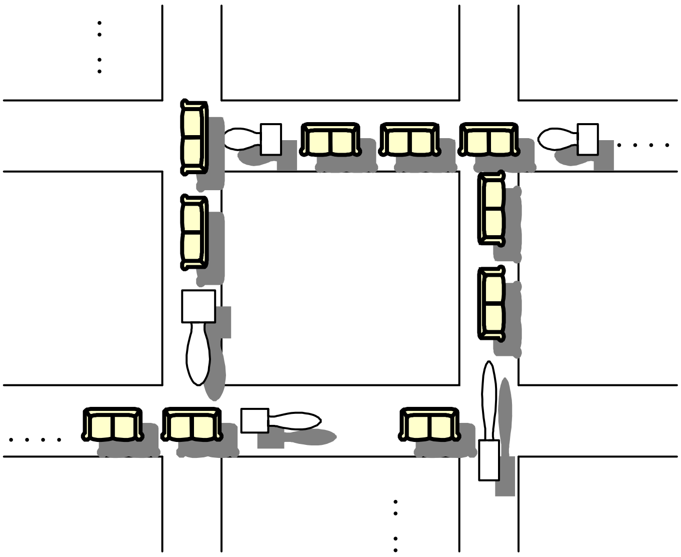
5.0 Summary

A deadlock state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

As discussed in this unit, a deadlock situation may occur if and only if four necessary conditions hold simultaneously in the system and there are three principal methods for dealing with deadlocks. You will learn about these methods in the next unit.

6.0 Tutor-Marked Assignment

1. Consider the traffic deadlock depicted below:
   (a) Show that the four necessary conditions for deadlock indeed hold in this example.
   (b) State a simple rule that will avoid deadlocks in this system.



2. List three examples of deadlocks that are not related to a computer system environment.
3. Is it possible to have a deadlock involving only one process? Explain your answer.
4. Using R-A graph, describe deadlocks
5. State the necessary conditions for deadlock to occur.
6. What are the various methods for handling deadlocks?

7.0 References/Further Reading

1. E. W. Dijkstra "EWD108: Een algorithme ter voorkoming van de dodelijke omarming" (in Dutch; *An algorithm for the prevention of the deadly embrace*)
2. Lubomir, F. Bic (2003). *Operating System Principles*. Prentice Hall.
3. "Operating System Concepts" by Silberschatz, Galvin, and Gagne (pages 259-261 of the 7th edition)

Module 5: Deadlocks

Unit 2: Methods for Dealing with Deadlocks

Table of Contents

1.0 Introduction

As you have seen in the previous unit, for a deadlock to occur, each of the four necessary conditions must hold. You were also introduced to some of the methods for handling a deadlock situation. In this unit you will be fully exposed to deadlock prevention and deadlock avoidance approaches. As discussed before, deadlock prevention is all about ensuring that at least one of the four necessary conditions cannot hold, we will elaborate further by examining each of the four conditions separately.

Deadlock avoidance is an alternative method for avoiding deadlocks which takes care of some of the shortcomings of deadlock-prevention such as low device utilization and reduced system throughput. In this unit, you will therefore learn how some of the algorithms for deadlock prevention, deadlock avoidance and deadlock detection and recovery are implemented.

2.0 Objectives

At the end of this unit you should be able to:

- Describe deadlock prevention
- Explain what is meant by deadlock avoidance
- Describe Banker's algorithm
- Describe Resource-Allocation graph algorithm
- Explain what is meant by safe state
- Describe Deadlock lock detection algorithms and how to recover from deadlock

3.0 Main Body

3.1 Deadlock Prevention

As you have seen in unit 1 of this module, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of the four necessary conditions cannot hold, you can prevent the occurrence of a deadlock. Now, we will elaborate on this approach by examining each of the four conditions separately.

3.1.1 Mutual Exclusion

The mutual exclusion condition must hold for non-sharable resources. For instance, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically nonsharable. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.

3.1.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs, in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the tape drive, disk file and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the tape drive and disk file. It copies from the tape drive to the disk file. The process must then request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

These protocols have two main disadvantages:

i. Resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and again request the disk file and printer, only if we can sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.
ii. Starvation is possible. A process that needs several popular resources may have to wait indefinitely; because at least one of the resources that it needs is always allocated to some other process.

3.1.3 No Pre-emption

The third necessary condition is that there be no pre-emption of resources that have been allocated. To ensure that this condition does not hold, we can use the following protocol.

If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources are pre-empted. In other words, these resources are implicitly released. The pre-empted resources are added to the list of resources for which the process is waiting. The process will be started only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available we check whether they are allocated to some other process that is which waiting for additional resources. If so, we pre-empt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.

While it is waiting, some of its resources may be pre-empted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were pre-empted while it is waiting.

This protocol is often applied to resources whose state can be easily saved and restored later such as CPU registers and memory space. It cannot generally be applied to such resources as printer and tape drives.

3.1.4 Circular Wait

The fourth and final condition for deadlocks is the circular wait condition. Circular wait prevention consists in allowing processes to wait for resources, but ensure that the waiting cannot be circular. One approach might be to assign a precedence/ordering to each resource and force processes to allocate resources in order of increasing precedence. That is to say that if a process holds some resources and the highest precedence of these resources is *m*, then this process cannot request any resource with precedence/ordering smaller than *m*. This forces resource allocation to follow a particular and non-circular ordering, so circular wait cannot occur.

Another approach is to allow holding only one resource per process; if a process requests another resource, it must first free the one it's currently holding (or hold-and-wait).

3.2 Deadlock Avoidance

Deadlock can be avoided if certain information about processes is available in advance of resource allocation. For every resource request, the system sees if granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock. The system then only grants request that will lead to *safe* states. In order for the system to be able to figure out whether the next state will be safe or unsafe, it must know in advance at any time the number and type of all resources in existence, available, and requested.

Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

The various algorithms differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information, about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the deadlock-avoidance approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

One known algorithm that is used for deadlock avoidance is the Banker's algorithm, which requires resource usage limit to be known in advance. However, for many systems it is impossible to know in advance what every process will request. This means that deadlock avoidance is often impossible.

Two other algorithms are Wait/Die and Wound/Wait, each of which uses a symmetry-breaking technique. In both these algorithms there exists an older process (O) and a younger process (Y). Process age can be determined by a timestamp at process creation time. Smaller timestamps are older processes, while larger timestamps represent younger processes.

Table 1

	Wait/Die	**Wound/Wait**
O is waiting for a resource that is being held by Y	O waits	Y dies
Y is waiting for a resource that is being held by O	Y dies	Y waits

It is important to note that a process may be in unsafe state but would not result in a deadlock. The not ion of safe/unsafe state only refers to the ability of the system to enter a deadlock state or not . For example, if a process requests A which would result in an unsafe state, but releases B which would prevent circular wait, then the state is unsafe but the system is not in deadlock.

3.2.1 Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formerly, a system is in a safe state only if there exists a safe sequence. A sequence of processes $<P_1, P_{2,},\dots, P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resources that $P_{i,}$ can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < i$. In this situation, if the resources that process P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all its needed resources, complete its designated task, return its allocated resources and terminate. When P_i terminates, P_{i+1} can obtain its needed resources and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks, however, an unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlock) state. In an unsafe state, the operating system cannot prevent processes from request resources such that a deadlock occurs: The behaviour of the processes controls unsafe states.

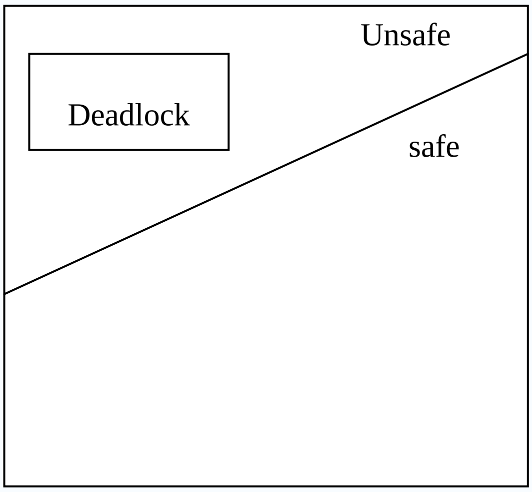


Figure 3.1: Safe, Unsafe, and Deadlock State Spaces

To illustrate, let us consider a system with 12 magnetic tape drives, and processes: P_0, $P_{1,}$,and P_2. Process P_0, requires 10 tape drives, process P_1 may need as many as 4, and process P_2 may need up to 9 tape drives. Suppose that at time t_0, process P_0 is holding 5 tape drives, process P_1 is holding 2, and process P_2 is holding 2 tape drives. Therefore, there are 3 free tape drives.

Table 2

Processes	**Maximum Needs**	**Current Needs**
P_0	10	5
P_1	4	2
P_2	9	2

At time t_0, the system is in a safe state. The sequence < P_1, P_0 P_3> satisfies the safety condition, since process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process P_0 can get all its tape and return them (the system will then have 10 available tape drives), and finally process P_2 could get all its tape drives and return them (the system will then have all its 12 tape drives available).

A system may go from a safe state to an unsafe state. Suppose that at time t_1, process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drive. If we had made P_2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Given the concept of safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

In this scheme, if a process requests a resource that is currently available, it may still have to wait. Therefore, resource utilization may be lower than it would be without a deadlock-avoidance algorithm.

3.2.2 Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined in Section 3.2.2 of the last unit can be used for deadlock avoidance.

In addition to the request and assignment edges, we introduce a new type of edge, called a **claim edge**. A claim edge $P_i \rightarrow R_j$ indicates that process Pi may request resource R_j at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process Pi requests resource Rj, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly when a resource Rj is released by Pi, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. Note that the resources must be claimed a priori in the system. That is, before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.

Suppose process P_i request resource R_j. The request can be granted only if converting the request edges $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. You check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

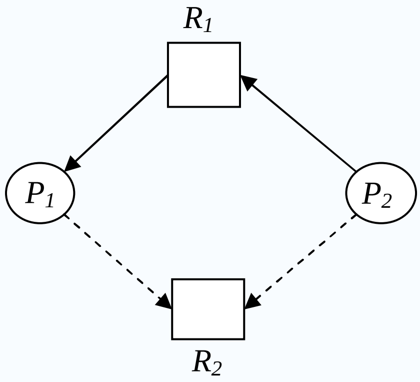


Figure 3.2 Resource-Allocation graph for deadlock avoidance

If no cycles exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process P_i will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 3.2. Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this will create a cycle in the graph (Figure 3.3). A cycle indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

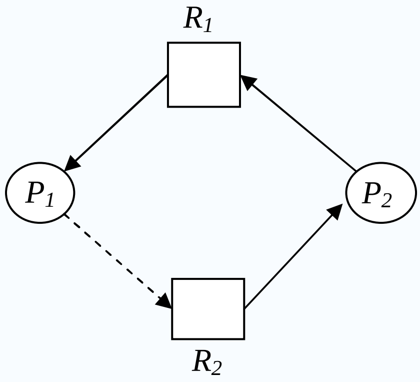


Figure 3.3 ***An unsafe state in a Resource-Allocation graph***

3.2.3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the "Banker's Algorithm". The name was chosen because this algorithm could be used in a banking system to ensure bank never allocates its available cash such that it can no longer satisfy the needs of all it customers.

3.2.3.1 Algorithm

The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

3.2.3.2 Resources

For the Banker's algorithm to work, it needs to know three things:

- How much of each resource each process could possibly request?
- How much of each resource each process is currently holding?
- How much of each resource the system has available?

Some of the resources that are tracked in real systems are memory, semaphores and interface access.

Example 1

Assuming that the system distinguishes between four types of resources, (A, B, C and D), the following is an example of how those resources could be distributed. *Note that this example shows the system at an instant before a new request for resources arrives. Also, the types and number of resources are abstracted. Real systems, for example, would deal with much larger quantities of each resource.*

```
Available system resources:
A B C D
4 3 3 3

Processes (currently allocated resources):
   A B C D
P1 1 2 2 1
P2 1 0 3 3
P3 1 1 1 0

Processes (maximum resources):
   A B C D
P1 3 3 2 2
P2 1 2 3 4
P3 1 1 5 0
```

3.2.3.3 Safe and Unsafe States

A state (as in the above example) is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. This is a reasonable assumption in most cases since the system is not particularly concerned with how long each process runs (at least not from a deadlock avoidance perspective). Also, if a process terminates without acquiring its maximum resources, it only makes it easier on the system.

Given that assumption, the algorithm determines if a state is safe by trying to find a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an unsafe state.

Example 2

We can show that the state given in the previous example is a safe state by showing that it is possible for each process to acquire its maximum resources and then terminate.

1. P1 acquires 2 A, 1 B and 1 D more resources, achieving its maximum
   - The system now still has 1 A, no B, 1 C and 1 D resource available
2. P1 terminates, returning 3 A, 3 B, 2 C and 2 D resources to the system
   - The system now has 4 A, 3 B, 3 C and 3 D resources available
3. P2 acquires 2 B and 1 D extra resources, then terminates, returning all its resources
   - The system now has 5 A, 3 B, 6 C and 6 D resources
4. P3 acquires 4 C resources and terminates
   - The system now has all resources: 6 A, 4 B, 7 C and 6 D
5. Because all processes were able to terminate, this state is safe

Note that these requests and acquisitions are *hypothetical*. The algorithm generates them to check the safety of the state, but no resources are actually given and no processes actually terminate. Also note that the order in which these requests are generated – if several can be fulfilled – does not matter, because all hypothetical requests let a process terminate, thereby increasing the system's free resources.

For an example of an unsafe state, look at what would happen if process 2 were holding 1 more unit of resource B at the beginning.

3.2.3.4 Requests

When the system receives a request for resources, it runs the Banker's algorithm to determine if it is safe to grant the request. The algorithm is fairly straight forward once the distinction between safe and unsafe states is understood.

1. Can the request be granted?

    - If not, the request is impossible and must either be denied or put on a waiting list
2. Assume that the request is granted
3. Is the new state safe?
    - If so, grant the request
    - If not, either deny the request or put it on a waiting list

Whether the system denies an impossible or unsafe request or makes it wait is an operating system specific decision.

Example 3

Continuing the previous examples, assume process 3 requests 2 units of resource C.

1. There is not enough of resource C available to grant the request
2. The request is denied

On the other hand, assume process 3 requests 1 unit of resource C.

1. There are enough resources to grant the request
2. Assume the request is granted
    - The new state of the system would be:

```
       A B C D
  Free 3 1 0 2
  P1   1 2 2 1
  P2   1 0 3 3
  P3   1 1 2 0
```

1. Determine if this new state is safe
    1. P1 can acquire 2 A, 1 B and 1 D resources and terminate
    2. Then, P2 can acquire 2 B and 1 D resources and terminate
    3. Finally, P3 can acquire 3 C resources and terminate
    4. Therefore, this new state is safe
2. Since the new state is safe, grant the request

Finally, assume that process 2 requests 1 unit of resource B.

1. There are enough resources
2. Assuming the request is granted, the new state would be:

```
       A B C D
  Free 3 0 1 2
  P1   1 2 2 1
  P2   1 1 3 3
```

```
     P3   1 1 1 0
```

1. Is this state safe? Assuming P1, P2, and P3 request more of resource B and C.
   - P1 is unable to acquire enough B resources
   - P2 is unable to acquire enough B resources
   - P3 is unable to acquire enough C resources
   - No process can acquire enough resources to terminate, so this state is not safe
2. Since the state is unsafe, deny the request

Note that in this example, no process was able to terminate. It is possible that some processes will be able to terminate, but not all of them. That would still be an unsafe state.

3.2.3.5 Trade-offs

Like most algorithms, the Banker's algorithm involves some trade-offs. Specifically, it needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making the Banker's algorithm useless. Besides, it is unrealistic to assume that the number of processes is static. In most systems the number of processes varies dynamically. Moreover, the requirement that a process will eventually release all its resources (when the process terminates) is sufficient for the correctness of the algorithm, however it is not sufficient for a practical system. Waiting for hours (or even days) for resources to be released is usually not acceptable.

3.3 Deadlock Detection

Often neither deadlock avoidance nor deadlock prevention may be used. Instead, deadlock detection and process restart are used by employing an algorithm that tracks resource allocation and process states, and rolls back and restarts one or more of the processes in order to remove the deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler or OS.

Detecting the possibility of a deadlock *before* it occurs is much more difficult and is, in fact, *generally* undecidable, because the halting problem can be rephrased as a deadlock scenario. However, in *specific* environments, using *specific* means of locking resources, deadlock detection may be *decidable*. In the *general* case, it is not possible to distinguish between algorithms that are merely waiting for a very unlikely set of circumstances to occur and algorithms that will never finish because of deadlock.

3.4 Distributed deadlock

Distributed deadlocks can occur in distributed systems when distributed transactions or concurrency control is being used. Distributed deadlocks can be detected either by constructing a global wait-for graph from local wait-for graphs at a deadlock detector or by a distributed algorithm like edge chasing.

Phantom deadlocks are deadlocks that are detected in a distributed system but do not actually exist - they have either been already resolved or no longer exist due to transactions aborting.

3.5 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exists. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to pre-empt some resources from one or more of the deadlocked processes.

3.5.1 Process Termination

To eliminate deadlocks by aborting process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes:** this method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated:** this method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job.

If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. This determination is a policy decision, similar to CPU scheduling problems. The question is basically an economic one. We should abort those processes the termination of which will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may determine which process is chosen, including:

1. What the priority of the process is?

2. How long the process has computed, and how much longer the process will compute before completing its designated task.?
3. How many and what type of resources the process has used (for example, whether the resources are simple to pre-empt)?
4. How many more resources the process needs in order to complete?
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch?

3.5.2 Resource Pre-emption

To eliminate deadlocks using resource pre-emption, we successfully pre-empt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If pre-emption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim**: which resources and which processes are to be pre-empted? As in process termination, w must determine the order of pre-emption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
2. **Rollback**: if we pre-empt, a resource from a process, what should be done with that process? Clearly it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state.

   Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback. Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

3. **Starvation**: how do we ensure that starvation will not occur. That is, how can we guarantee that resources will not always be pre-empted from the same process?

   In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

3.6 Livelock

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of *resource starvation*; the general definition only states that a specific process is not progressing.

As a real-world example, livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they always both move the same way at the same time.

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can repeatedly trigger. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.

4.0 Conclusion

In this unit, you have learnt more about deadlocks especially the three methods for dealing with deadlocks:

- Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- Allow the system to enter deadlock state, detect it, and then recover.
- Ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems including UNIX.

5.0 Summary

A deadlock situation may occur if and only if four necessary conditions hold simultaneously in the system: mutual exclusion, hold and wait, no pre-emption, and circular wait. To prevent deadlocks, we ensure that at least one of the necessary conditions never holds.

Another method for avoiding deadlocks that is less stringent than the prevention algorithms is to have a priori information on how each process will be utilizing the resources. The banker's algorithm, for example, needs to know the maximum number of each resource class that may be requested by each process. Using this information, we can define a deadlock-avoidance algorithm.

If a system does not employ a protocol to ensure that deadlocks will never occur, then detection algorithm must be invoked to determine whether a deadlock has occurred. If a deadlock is detected, the system must recover either by terminating some of the deadlocked processes, or by pre-empting resources from some of the deadlocked processes.

In a system that selects victims for rollback primarily on the basis of cost factors, starvation may occur. As a result, the selected process never completes its designated task.

6.0 Tutor Marked Assignment

1. In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, new resources are bought and added to the system. If deadlocks is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

   a. Increase *Available* (new resources added)
   b. Decrease *Available* (resources permanently removed from system)
   c. Increase *Max* for one process (the process needs more resources than allowed, it may want more)
   d. Decrease *Max* for one process (the process decides it does not need that many resources)
   e. Increase the number of processes
   f. Decrease the number of processes

2. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

3. Consider a system that runs 5,000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about N2.00 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted.

   A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

   a. What are the arguments for installing the deadlock-avoidance algorithm?
   b. What are the arguments against installing the deadlock-avoidance algorithm?

4. Consider the following snapshot of a system:

	Allocation				*Max*				*Available*			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Answer the following questions using the banker's algorithm:

a) What is the content of the matrix *Need*?
b) Is the system in a safe state?
c) If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

5. Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

   For example, consider a system with three resource types and the vector *Available* initialized to (4,2,2). If process P_0 asks for (2,2,1), it gets them. If P_1 asks for (1,0,1), it gets them. Then, if P_0 asks for (0,0,1), it is blocked (resource not available). If P_2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P_0 (since P_0 is blocked). P_0's *Allocation* vector goes down to (1,2,1), and its *Need* vector goes up to (1,0,1).

a) Can deadlock occur? If so, give an example. If not, which necessary condition cannot occur?
b) Can indefinite blocking occur?

7.0 References/Further Reading

1. E. W. Dijkstra "EWD108: Een algorithme ter voorkoming van de dodelijke omarming" (in Dutch; *An algorithm for the prevention of the deadly embrace*)
2. Lubomir, F. Bic (2003). *Operating System Principles*. Prentice Hall.
3. "Operating System Concepts" by Silberschatz, Galvin, and Gagne (pages 259-261 of the 7th edition)
4. "EWD623: The mathematics behind the Banker's Algorithm" (1977) by E. W. Dijkstra, published as pages 308–312 of Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982. ISBN 0-387-90652-5

Module 6: Memory Management

Unit 1: Memory Management Fundamentals

Table of Contents

1.0 Introduction

In module 3 you were shown how the CPU can be shared by a set of processes. As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer’s response to its users. To realise this increase in performance, however, we must keep several processes in memory; that is, we must share memory.

In this module, we will discuss various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system. As you shall see, many algorithms require hardware support, although recent designs have closely integrated the hardware and operating system.

As you learnt in the first module of this course, memory is central to the operation of a modern computer system. Memory consist of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, etc.) or what they are for (instructions or data). Accordingly, we can ignore how a memory address is generated by a program. We are interested in only the sequence of memory addresses generated by a running program.

2.0 Objectives

At the end of this unit you should be able to:

- Describe address binding
- Define logical and physical address space
- Briefly explain dynamic loading
- Distinguish between dynamic loading and dynamic linking
- Define what is meant by shared libraries
- Describe the principle of overlays and its uses

3.0 Main Body

3.1 Address Binding

Usually a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk memory during its execution. The collection of processes on the disk that is waiting to be brought into memory for execution forms the input queue.

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Most systems allow user process to reside in any part of the physical memory. Therefore, although the address space of the computer starts at 00000, the first address of the user process does not need to be 00000. This arrangement affects the addresses that the user program can use. In most cases, a user program will go through several steps, some of which may be optional, before being executed (Figure 3.1). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as *count*). A compiler will typical bind these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”). The linkage editor or loader will in turn bind these relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically the binding of instructions and data to memory addresses can be done at any step along the following ways:

Compile time: if you know at compile time where the process will reside in memory, then absolute code can be generated. For instance, if you know a priori that a user process resides starting at location *R*, then the generated compiler code will start at that location and extend up from there. If at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are absolute code bound at compile time.

Load time: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load. If the starting address changes, you need only to reload the user code to incorporate this changed value.

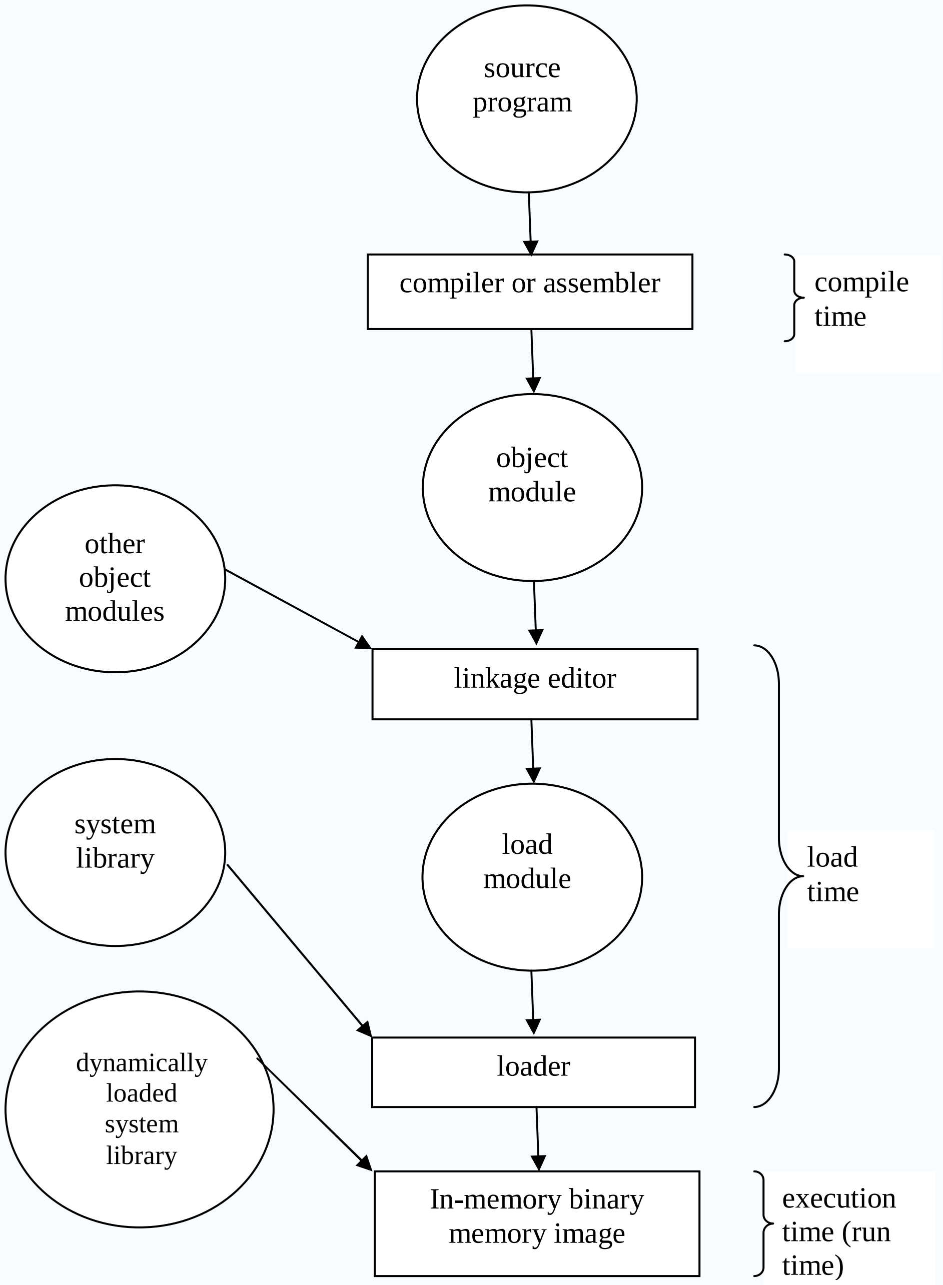


Figure 3.1: Multistep processing of a user program.

Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as you will learn in the next section. Most general purpose operating system use this method.

A major part of this module is devoted to showing how these various bindings can be implemented effectively in a computer system and to discussing appropriate hardware support.

3.2 Logical-Address Space Versus Physical-Address Space

An address generated by the CPU is commonly referred to as logical address, whereas an address seen by the memory unit – that is, the one loaded into the **memory-address register** of the memory – is commonly referred to as a **physical address**.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a virtual address. We use logical address and virtual address interchangeably in this text. The set of all logical addresses generated by a program is a logical-address space; the set of all physical addresses corresponding to these logical addresses is a physical-addresses space. Thus, in the execution-time address-binding scheme, the logical- and physical-address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU). We can choose from among many different methods to accomplish such a mapping as you will learn in subsequent sections of this unit.

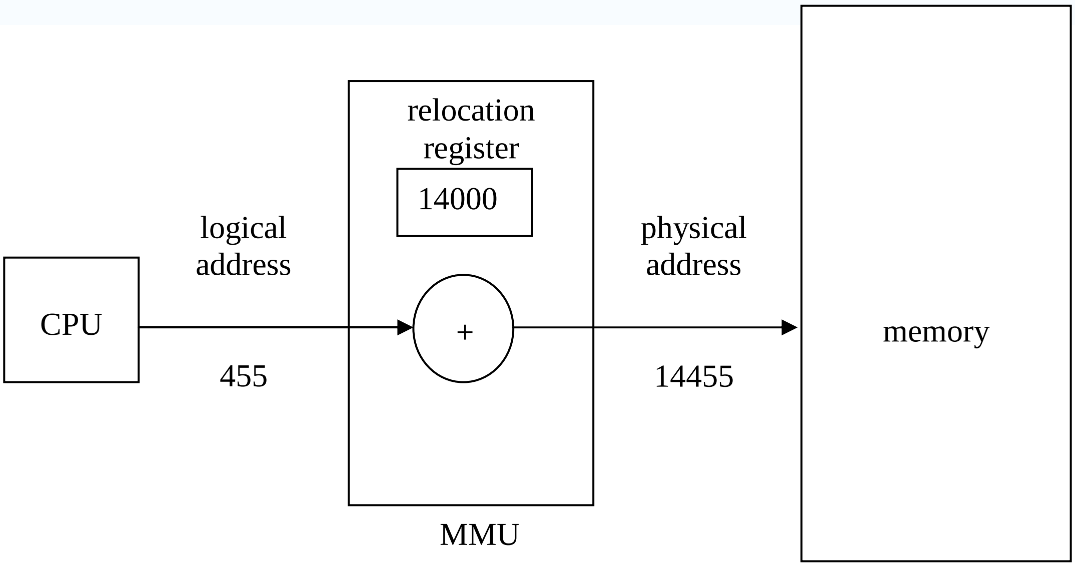


Figure 3.2: Dynamic relocation using a relocation register.

Meanwhile, we illustrate this mapping with a simple MMU scheme which is a generalization of the base-register scheme.

This method requires the hardware support illustrated in figure 3.2. The base register is here called relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 455 is mapped to location 14455. The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes.

The user program never sees the real physical addresses. The program can create a pointer to location 455, store it in memory, manipulate it, compare it to other addresses – all as the number 455. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. You learnt about this form of execution-time binding in the previous section. The final location of a referenced memory address is not determined until the reference is made.

We now have two different types of addresses: logical addresses (in the range of 0 to *max*) and physical addresses (in the range R + 0 to R + *max* for base value R). The user generates only logical addresses and thinks that the process runs in location 0 to *max*. The user program supplies logical addresses, these logical addresses must be mapped to physical addresses before they are used.

Note that the concept of *logical-address space* that is bound to a separate *physical-address space* is central to proper memory management.

3.3 Dynamic Loading

So far you have learnt that the entire program and data must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such method. Operating system may help the programmer, however, by providing library routines to implement dynamic loading.

3.4 Dynamic Linking and Shared Libraries

Figure 3.1 also shows the dynamically linked libraries. Some operating systems support only static linking, in which the system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, all programs on a system need to have a copy of their language library (or at least the routine referenced by the program) included in the executable image. This requirement wastes both disk space and main memory. With dynamic linking, a stub is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.

When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine, and executes the routine. Hence, the next time that the code segment is reached, the library routine is executed directly incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Minor changes retain the same version number, whereas major changes increment the version number. Therefore only programs that are compiled with new library version are affected by the incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries.**

Unlike dynamic loading, dynamic linking generally requires help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process' memory space, or that can allow multiple processes to access the same memory addresses.

3.5 Overlays

You can use **overlays** to enable a process to be larger than the amount of memory allocated to it. The idea is to keep in memory only the instructions and data that are need at any given time. When other instructions are needed, they are loaded into space previously occupied by instructions that are no longer needed.

For example, consider a two- pass assembler. During pass 1, it constructs a symbol table and during pass 2, it generates machine-language code. You may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table and common support routines used by both passes 1 and 2. Assume that the sizes of these components are as follows:

Pass 1	90 KB
Pass 2	60 KB
Symbol table	40 KB

Common routines	50 KB

To load everything at once, you would require 240 KB of memory. If only 200 KB is available, you cannot run your process. However, not ice that pass 1 and pass 2 do not need to be in memory at the same time. You can therefore define two overlays as follows:

Overlay A is the symbol table, common routines and pass 1.

Overlay B is the symbol table, common routines and pass 2.

You then add an overlay driver of say 10 KB and start with overlay A in memory. When you finish pass 1, you jump to the overlay driver which reads overlay B into memory overwriting overlay A, and then transfer control to pass 2. Overlay A needs 180 KB while overlay B needs only 150 KB (see figure 3.3). You can then run your assembler in the 200 KB memory. It will load faster due to the fact that fewer data need to be transferred before execution starts. However, it will run slower, due to the extra I/O to read the code for overlay B over the code for overlay A.

The codes for overlay A and B are kept on disk as absolute memory images, and are read by the overlay drivers as needed.

As in dynamic loading, overlays do not require any special support from the operating system.

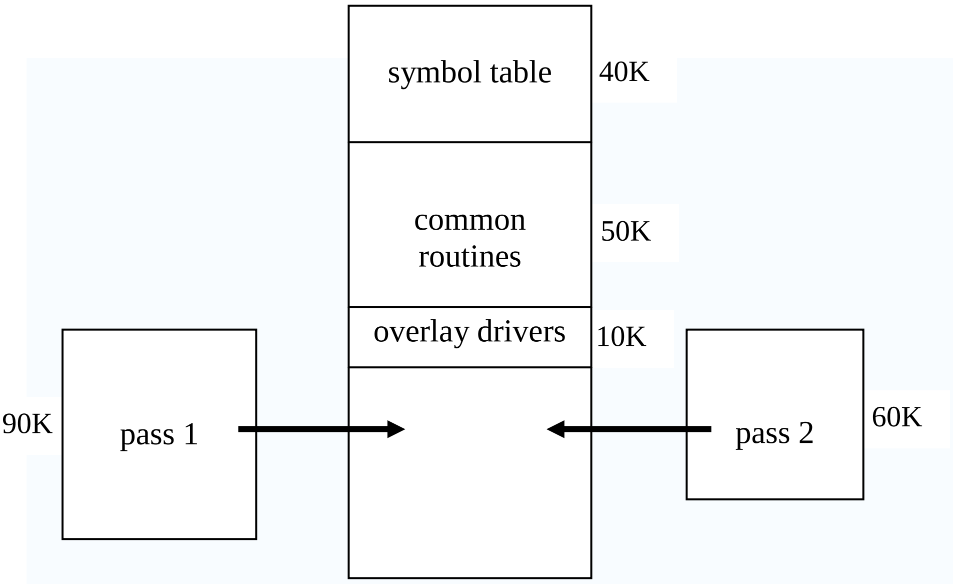


Figure 3.3: Overlay for a two-pass assembler

3.5 Swapping

As you have learnt so far in the course, 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 algorithm. 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 has been freed (Figure 3.4). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process.

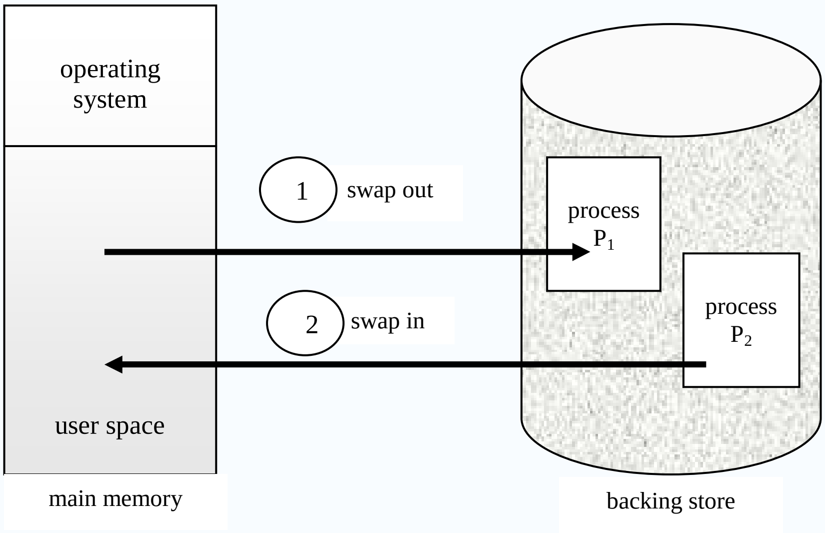


Figure 3.4: Swapping of two processes using a disk as a backing store

Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously especially if binding is done at assembly or load time. However, if execution time binding is being used, then the process can be swapped into a different memory space because the physical addresses are computed at execution time.

As earlier mentioned, swapping requires a backing store. The backing store is commonly a fast disk which must be large enough to accommodate copies of all memory images for all users. It must also provide a direct access to these memory images. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfer control to the selected process.

The context-switch time in such a swapping system is fairly high.

4.0 Conclusion

In this unit, you have been taken through some fundamental concepts of memory management. In the subsequent units of this module you will learn more about the various techniques of memory management. Meanwhile you are advised to consult the references for in-depth knowledge of the various concepts treated in this module.

5.0 Summary

In this unit, you have learnt the following:

Address binding can be done at any of the following stages:

- Compile time
- Load time
- Execution time

An address generated by the CPU is called logical address while the one seen by the memory unit is called physical address.

Dynamic loading and dynamic linking are used to obtain better memory space utilization. However, dynamic linking requires help from the operating system while dynamic loading does not .

Overlays are used to allow a process to be larger than the amount of memory allocated to it. Overlays, like dynamic loading do not require special operating system support.

A process can be swapped in and out of memory to a backing store.

6.0 Tutor Marked Assignment

1. Name two differences between logical and physical addresses.
2. Distinguish between dynamic linking and dynamic loading.
3. How is dynamic linking related to shared libraries?

7.0 References/Further Reading

1. Lubomir, F. Bic (2003). *Operating System Principles*. Prentice Hall.
2. "Operating System Concepts" by Silberschatz, Galvin, and Gagne (7th edition)

Module 6: Memory Management

Unit 2: Memory Allocation Techniques

Table of Contents

1.0 Introduction

As you have learnt from the previous unit, memory management is essential to process execution which is the primary job of the CPU. The main memory must accommodate both the operating system and the various user processes. You therefore need to allocate different parts of the main memory in the most efficient way possible. In this unit, therefore, you will about the some memory management algorithms such as contiguous memory allocation and its different flavours. Also, the problems that may arise from contiguous memory allocation (fragmentation) will be discussed in this unit.

2.0 Objectives

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

- Describe contiguous memory allocation
- Describe the various variants of contiguous memory allocation such as best-fit, worst-fit, and first-fit
- Distinguish between internal and external fragmentation
- Describe methods of solving external fragmentation

3.0 Main Body

3.1 Contiguous Memory Allocation

As you may not iced on your system, the memory is usually divided into two partitions: one for resident operating system and one for the user processes. You may place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Therefore, in this course, we shall only discuss the situation where the operating system resides in low memory.

We usually want several user processes to reside in memory at the same time. We, therefore, need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

3.2 Memory Allocation

Memory allocation can be done in two ways:

- fixed partition
- variable partition

These two methods are discussed in the following sections.

3.2.1 Fixed Partition Methods

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. in this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT). It is no longer in use. The method we are going to describe next is a generalization of the fixed-partition scheme (called MVT). It is used primarily in a batch environment. The ideas presented are also applicable to time-sharing environment in which pure segmentation is used for memory management.

3.2.2 Variable Partition Methods

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially all memory are available for user processes, and is considered as one large block of available memory, a **hole**. When a process arrives and needs memory, we search for a hole large enough for this process. if we find one, we allocate only as much as is needed, keeping the rest available to satisfy future requests.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied i.e. no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, a set of hole of different sizes is scattered throughout memory at any given time. When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. if the hole is too large, is too large, it is split into two. One part is allocated to the arriving process, the other is returned to the set of holes. When a process terminates, it releases its block of memory. Which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

The procedure is a particular instance of the general dynamic storage allocation problem, which is how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate.

3.3 Memory Allocation Strategies

The first-fit, best-fit and worst-fit strategies are the most common ones used to select a free hole from the set of available holes.

3.3.1 First-Fit

In first-fit algorithm, you allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. You can stop searching as soon as you find a free hole that is large enough.

3.3.2 Best-Fit

In best-fit algorithm, you allocate the smallest hole that is big enough. You must search the entire list from top to bottom except in a case where the list is ordered by size. This strategy produces the smallest leftover hole.

3.3.3 Worst-Fit

In worst-fit algorithm, you allocate the largest available hole. As in best-fit, you must search the entire list, unless the list is kept ordered by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

It can be shown, using techniques such as simulations, that both first-fit and best-fit are better than worst-fit in terms of decreasing both time and storage utilization. Neither first-fit nor best-fit is clearly better in terms of storage utilization, but first-fit is generally faster.

However, these algorithms suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough total memory space exists to satisfy a request but it is not contiguous. Storage is fragmented into large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all this memory were in one big free block, we might be able to run several more processes.

The selection of the first-fit versus best-fit strategies can affect the amount of fragmentation. First-fit is better for some systems whereas best-fit is better for others. Another factor is which end of a free block do you allocate? However, you should note that no matter which algorithm you use, external fragmentation will be a problem.

3.4 Fragmentation

In the previous section you learnt about external fragmentation. You should, however, note that memory fragmentation can be internal or external.

3.4.1 Internal Fragmentation

To illustrate this, consider a multiple partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If you allocate the requested blocks, you are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach is to break the physical memory into fixed-sized blocks, and allocate memory in unit of block sizes. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The different between these two numbers is internal fragmentation i.e. memory that is internal to a partition but is not being used.

3.4.2 Solutions to External Fragmentation

1. **Compaction:** this is a solution to the problem of external fragmentation. The goal is to shuffle the memory contents to place all free memory together in one large block. But compaction is not always possible. If relocation is static and is done at assembly or load time, compaction cannot be done. Compaction is only possible if relocation is dynamic, and is done at run time.

2. **Noncontiguous Logical-Address Space:** Another solution to external fragmentation problem is to permit the logical-address space of a process to be noncontiguous. Therefore, allowing a process to be allocated physical memory wherever the latter is available. Two ways of achieving this are through paging and segmentation or you can combine the two techniques of paging and segmentation. You will be exposed to these two techniques in the next unit.

4.0 Conclusion

In this unit, you have learnt about some of the algorithms for memory management especially contiguous memory allocation. You have also learnt about the problem of fragmentation especially external fragmentation. In the next unit, we will go further and discuss paging and segmentation which are ways of implementing noncontiguous logical-address space solution to external fragmentation.

5.0 Summary

Memory management algorithms for multiprogrammed operating systems range from simple single-user system approach to paged segmentation. In this unit. We have only discussed contiguous memory allocation. In unit 3, we will continue with our discussion on memory allocation algorithms and also outline the criteria to use in comparing the various memory management algorithms.

6.0 Tutor Marked Assignment

1. Explain the difference between internal and external fragmentation.

2. Describe the following allocation algorithms:

- First-fit
- Best-fit
- Worst-fit

Hence or otherwise, given the memory partitions of 100 KB, 500 KB, 200 KB, 300 KB and 600 KB (in that order), how would each of the fist-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in that order)? For this particular case, which algorithm makes the most efficient use of memory?

7.0 References/Further Reading

1. Lubomir, F. Bic (2003). *Operating System Principles*. Prentice Hall.
2. "Operating System Concepts" by Silberschatz, Galvin, and Gagne (7th edition)
3. *Modern Operating Systems (2nd Edition)* by Andrew S. Tanenbaum (ISBN 0-13-031358-0)

Module 6: Memory Management

Unit 3: Non-Contiguous Allocation

Table of Contents

1.0 Introduction

In the last university, you learnt about contiguous memory allocation in which it was highlighted that external fragmentation is a major problem with this method of memory allocation. It was also mentioned that compaction and non-contiguous logical address space are solutions to external fragmentation. In this unit we will go further to discuss some of the techniques for making the physical address space of a process non-contiguous such as paging, segmentation, etc.

2.0 Objectives

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

- Describe paging
- Describe segmentation
- Explain the differences between paging and segmentation
- State the advantages and disadvantages of both paging and segmentation
- Describe a method for solving the problems of both paging and segmentation

3.0 Main Body

3.1 Paging

This is a memory-management scheme that permits the physical-address space of a process to be contiguous. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store from which most of the previous memory-management schemes suffered.

Paging permits the logical address space to be mapped to a number of equal size blocks called *page frames*, by dividing the logical address space into *pages* of the same size. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same as the memory frames.

The hardware support for paging is as illustrated in Figure 3.1 below. Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number is used as an index into a **page table**. The page table contains the based address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

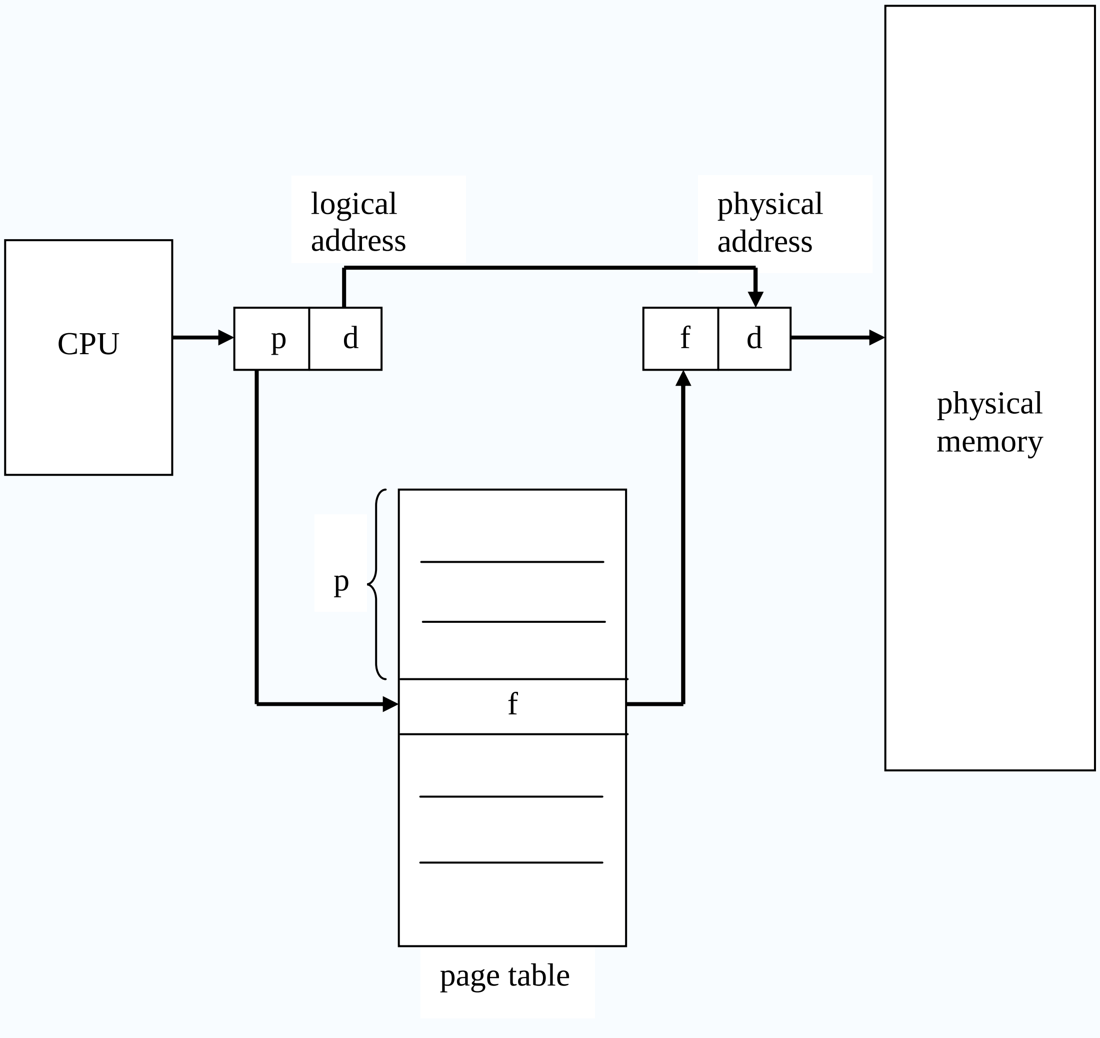


Figure 3.1: Paging Hardware

The paging model of memory is shown in Figure 3.2 below.

The page size, like the frame size, is defined by the hardware. The size of a page is of power 2 and it varies between 512 bytes and 16 MB per page, depending on the computer architecture.

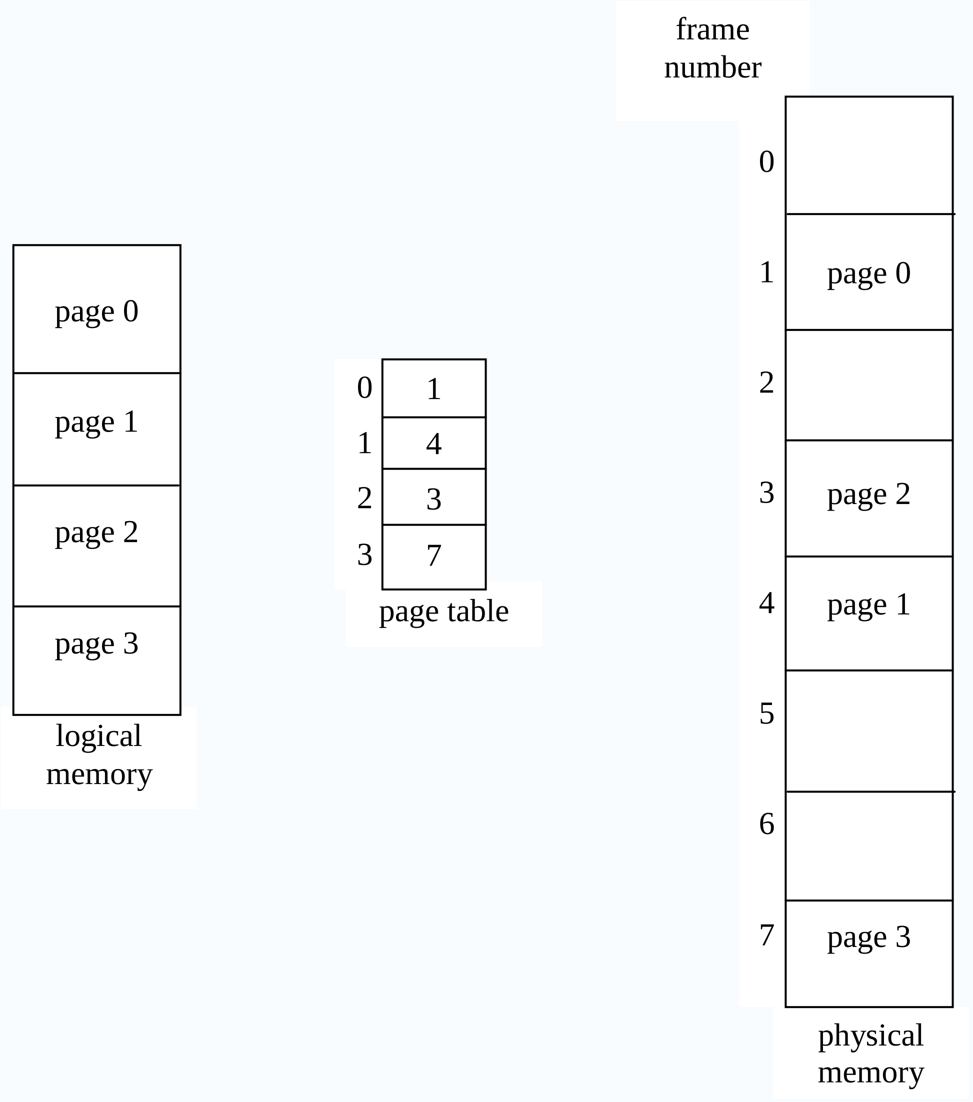


Figure 3.2: Paging model of logical and physical memory

3.1.1 Translating the memory addresses

To minimize the performance penalty of address translation, most modern CPUs include an on-chip memory management unit (MMU), and maintain a table of recently used virtual-to-physical translations, called a Translation Lookaside Buffer (TLB). Addresses with entries in the TLB require no additional memory references (and therefore time) to translate. However, the TLB can only maintain a fixed number of mappings between virtual and physical addresses; when the needed translation is not resident in the TLB, action will have to be taken to load it in.

On some processors, this is performed entirely in hardware. The MMU has to do additional memory references to load the required translations from the translation tables, but no other action is needed. In other processors, assistance from the operating system is needed. An exception is raised, and the operating system handles this exception by replacing one of the entries in the TLB with an entry from the primary translation table, and the instruction which made the original memory reference is restarted.

3.1.2 Protected memory

Hardware that supports virtual memory almost always supports memory protection mechanisms as well. The MMU may have the ability to vary its operation according to the type of memory reference (for read, write or execution), as well as the privilege mode of the CPU at the time the memory reference was made. This allows the operating system to protect its own code and data (such as the translation tables used for virtual memory) from corruption by an erroneous application program and to protect application programs from each other and (to some extent) from themselves (e.g. by preventing writes to areas of memory that contain code).

3.1.3 Issues with Paging

As you may have not iced, paging is a form of dynamic relocation. Every logical address is bounded by the paging hardware to some physical address. Using paging is similar to using a table of base/relocation registers, one for each frame.

When you use a paging scheme, you have no external fragmentation. However, internal fragmentation may occur since frames are allocated as units.

3.2 Segmentation

Users do not think of memory as a linear array of bytes with some containing instructions and other containing data. Instead users prefer to view memory as a collection of variable-sized segments with no necessary ordering among segments. See Figure 3.3 below.

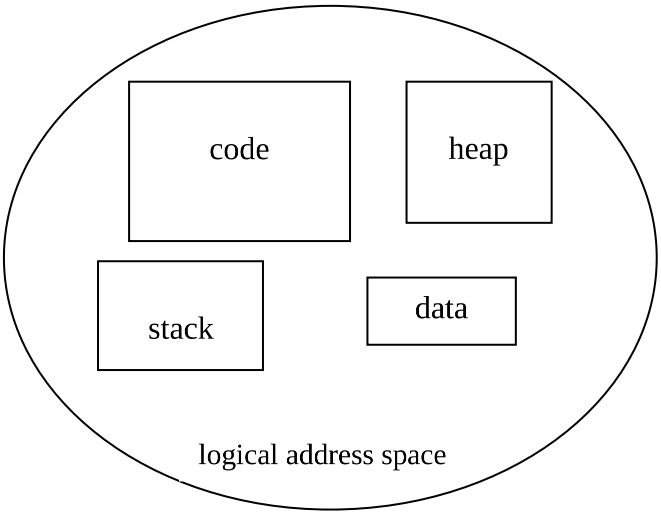


Figure 3.3: User's view of a program

Segmentation is a memory-management scheme that supports this user view of memory. A logical-address space is a collection of segments. Each segment has a name and a length. The addresses specify both segment name and the offset within the segment. The user, therefore, specifies each address by two quantities: a segment name and an offset.

For implementation simplicity, segments are numbered and are referred to by a segment number rather than by a segment name. Therefore, a logical address consists of a two tuple:

<segment-number, offset>.

Normally the user program is compiled and the compiler automatically constructs segments that reflects the input program.

3.2.1 Hardware Implementation

Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still a one-dimensional sequence of bytes. Hence, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by **segment table**. Each entry of the segment table has a segment *limit.* The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

The use of a segment table is as illustrated in Figure 3.4 below. A logical address consists of two parts: a segment number, *s* and an offset into that segment, *d*. The segment number is used as an index into the segment table. The offset *d* of the logical address must be between

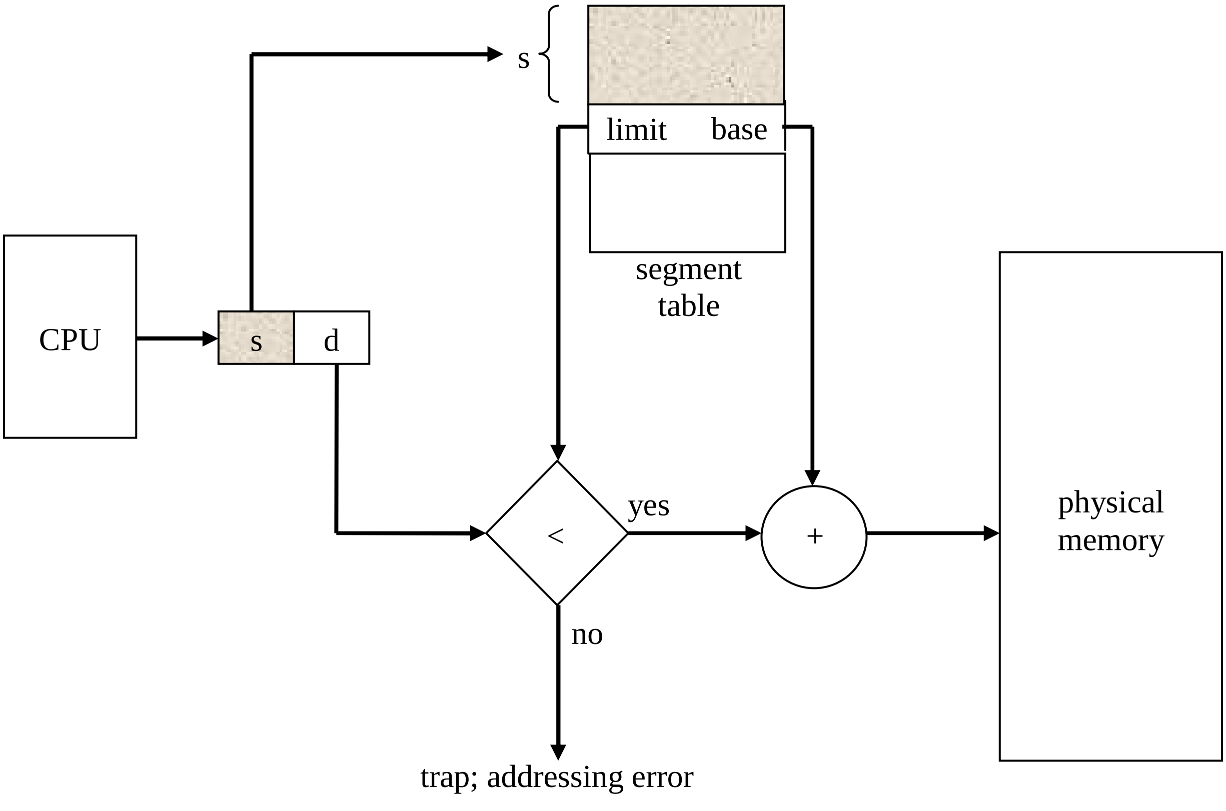


Figure 3.4: Segmentation hardware

0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is therefore essentially an array of base-limit register pairs.

3.2.2. Advantages and Problems of Segmentation

Advantages:

- Operating system may allow segments to grow and shrunk dynamically with unchanging addressing
- Protection on segment level of related data
- Sharing on segment level is easy.

Problems:

- Contiguous allocation of memory with all its attendant problems as you learnt from unit 2 of this module
- May cause external fragmentation
- Dynamic shrinking/growing is expensive. The operating system may have to move things around.

3.3 Segmentation with Paging

As you have learnt so far in this unit, both paging and segmentation have advantages and disadvantages. But the problems/disadvantages of these two can be solved by paging of the segments. In this combined technique, each segment has its own page table. Segment table entries now refer to base of the per segment page table and the offset within the segment is subdivided into page number and offset within page and used as earlier discussed.

This combination is the one used in the Intel 386 architecture.

4.0 Conclusion

In this concluding unit of this module and the course general, you have been further exposed to some memory-management algorithms. You should please note that the topics treated are not exhaustive. You therefore advised to refer to the references/further Reading sited at the end of each unit for more in-depth knowledge of the subject matter.

5.0 Summary

The various memory-management algorithms discussed in the last two units differ in many aspects. In comparing different memory-management strategies, you should use the following considerations:

- Hardware support
- Performance
- Fragmentation
- Relocation
- Swapping
- Sharing
- Protection

6.0 Tutor Marked Assignments

1. Why are page sizes always powers of 2?
2. Why are segmentation and paging sometimes combined into one scheme?
3. State the various advantages and disadvantages of paging
4. State the various advantages and disadvantages of segmentation
5. What are the differences between paging and segmentation?

7.0 References/Further Reading

1. Deitel, Harvey M.; Deitel, Paul; Choffnes, David (2004). *Operating Systems*. Upper Saddle River, NJ: Pearson/Prentice Hall. ISBN 0-13-182827-4.
2. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). *Operating System Concepts*. Hoboken, NJ: John Wiley & Sons. ISBN 0-471-69466-5.
3. Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems. Design and Implementation*. Upper Saddle River, N.J.: Pearson/Prentice Hall. ISBN 0-13-142938-8.
4. Tanenbaum, Andrew S. (2001). *Modern Operating Systems*. Upper Saddle River, N.J.: Prentice Hall. ISBN 0-13-092641-8.