Second Edition

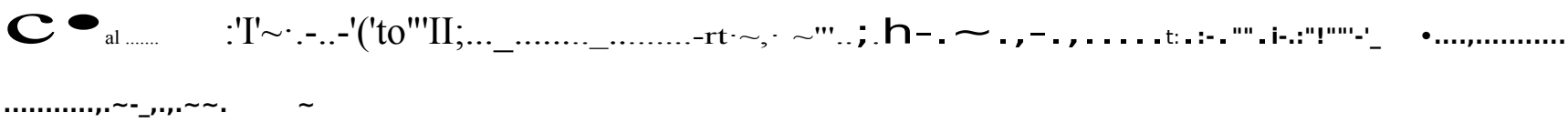
Tata McGraw-Hill

ISBN 0-07-059113-X

Published by the Tala McGraw-Hil.1 Publishing Company Limited,
7 West Patel Nagar. New Delhi 110 008. Typeset at Script Makers.
19, A 1-8. DDA Market, Paschim Vihar. New Delhi 110 ()63 and printed at
Adarsh Printers. C-51. Mohan Park, Naveen Shahdara. Delhi
110032

Cover Printer: Sbree Ram Enterprises

Contents



t:::::~'
~'::;;;;;.;;rc;;:;;;~;;;;;;;;~;;;;;;O:'::-'::::S ft ' '" :..:!:::;.:;;i;;o:::.;;.;;;:~gr.;;:::::;;-;; .. '"F'

_""Mer= $',

Preface to the Second Edition

In a sense, lowe this book to Mr. Narayana Murthy, a very successful businessman and a great computer scientist. When he left Patni Computer Systems (PCS)-which is now knO\\'11as "Patni"-Io start Infosys,
r joined PCS in his place. When I joined Patni, 1 found that I had to deal with a number of professionals
who were Bachelors and Masters in Computer Science especially from lIT. Al Patni, the work in those days also required a deep understanding of Computer Science and especially the Operating Systems. Though I was from Il'T, I was only a Chemical Engineer, and in my last three years at lIT, J bad read OIOreabout economics, literature and music than Engineering!

J had previous experience in designing and coding application systems in India, the UK and the US where 1 had participated in the design of an ERP (MRP 11)package and also was awarded by rBM several times for excellent performance. However, to manage young, sharp minds in computer science was a different ball game and. in fact. quite a challenge. I did not believe in only "managing from the top". J wanted to understand the technology thoroughly myself so that I could take the right decisions and be respected within the organization as well as the software industry.

This drove file to read a number of books on Digital Electronics. Algorithms, Computer Architecture and Operating Systems. After reading several of them (more than 20), I realized that there were many good books on "Operating Systems", but what was needed was a simple but rigorous book. which would demystify the subject for anybody, literally, and explain it in a step-by-step manner. This led me 10 write a book myself. However, when I wrote it first. I had not considered the syllabus of any University. J did not write with any specific commercial aim in mind. I only wanted to demystify the subject. Therefore, the exercises at the end of the chapters were meant only to belp the student grasp the subject in a simple fashion and not to help the students from the "examination" point of view.

The book was published. However, because it was not written as a text, it took a little while to get established in. the student community. Later on, the word of mouth did the tick. "You may read so and so book for the examination, but first read Godbole's book if you want to really understand the subject" were the kind of comments heard from hundreds of students and professors. Thus. even if the book was not "recommended" in some universities, many students even frOIDthose universities read u. Hundreds of students in the UK and USA started carrying it to those countries as well. The book went for more than
20 reprints, which was quite reassuring to witness this especially for a book, which was not meant to be a textbook. The finale came when Mr. Narayana Murthy as well as Mr F C Kohli-s-thc father of Indian Software lndustry-actually praised me publicly many times with words such as "\Ve know that Achyut has helped build many software companies in his career as a CEO or MD. However. the most important of all is that he is the only CEO in Asia to have written several deep technical books and on as complex a subject as "Operating Systems".

After many years of silent success, I decided to publish its second edition. Things had changed by then. Mainframes had given way to Minicomputers and the minicomputers in turn to desktops/laptops which, in fact were more powerful. Windows and UNIXlLinux had become virtually the global standards. The first

book had a fairly simple and detailed treatment of UNIX, but it did not cover Linux. The coverage of windows needed a tremendous improvement. Distributed Operating Systems, Real rime Operating Systems, and Multimedia Operating Systems were also gaining popularity. All these topics needed to be covered. The GUT had to be explained, examples in COBOL needed to be replaced by C. C++. etc. and the coverage on Inter Process Communications (lPC) and Deadlocks needed to be more comprebensi ve. All this has been done in the second edition again keeping simplicity, step-by-step approach and rigour in mind. This time, there was another change though. I decided to orient the book as a proper textbook after studying various syllabi. I hope that the students will not only find it useful from the point of view of examinations. but will also enjoy reading it.

There are five major groups of professionals:

- One group is the one who designs, and codes the Operating Systems. Tbis group needs to know the structure and the architecture of the Operating System at the deepest level. This group needs to know how various algorithms within an Operating System can be wrinen and organized in various layers to execute different functions and systems calls. This class of Systems Architects and Programmers are required to study the internals of the Operating Systems and also the Computer Architecture underneath it.
- The second group of professionals is that of "Systems Programmers", They need to know the "internals" of Operating Systems because they need to use the system calls. They are tbe ones who write various utilities. Typically, these are written in Assembly or C. This group is also called upon to write "device drivers" when a new device is to be supported by any Operating System.
- The third group consists of Systems Engineers and Systems Administrators who are required to install and oversee the smooth running of the Operating System. Even if they need not design or code the Operating Systems, they need to know different parts and modules of the Operating System. and how they are interconnected. They are expected to tune the Operating Systems for better performance, allocate/deallocate the disk space; manage passwords and the security of the Operating Systems, etc.
- The fourth group of professionals is that of Application Programmers. They write programs in C, C++, VC++, Java/J2EE or .NET environments on various databases. The Operating Systems shields all the complex algorithms for memory allocation, file UO and process synchronization! scheduling from this group. However, this group needs to know how an Operating System functions and helps the development and execution phases of the Application Systems.
- The fifth group of professionals is that of the end users who never program a computer. but only use it, This group will need to know bow to "use" an Operating System. This group basically is concerned only with Graphical User Interface (GUO of the Operating System. They need to know where to click to create a file, or to execute a specific program, etc.

The book is meant definitely for the first three groups. In theory, the fourth group of Application Programmers need not know the "internals" of an Operating System. However, if they do, they will get an extra kick due to the demystification of the subject. The last group of end users can keep away from the details of the Operating System, but again, the initiated ones with some logical thinking ability will enjoy the book, as it will explain in the simplest terms, the workings of Operating Systems. This is also the reason why there is a chapter oo Computer Architecture in this book.

I believe, this book will be very useful as an introductory text for any Computer Science or Information Technology Student.

This book tries to explain the subject in a step-by-step fashion so that just about anybody with logical thinking capability can understand it. This is done without sacrificing the rigour and accuracy, so that all five groups will benefit from this. Normally, a term is not used unless it is explained beforehand,

The book is organized as follows:

- Chapter 1 deals with the history of Operating Systems. It covers the various milestones in the history of Operating Systems, The chapter also covers the modem trends in Operating Systems.
- Chapter 2 begins with an overview of programming language levels, and presents a view at each of these levels, viz, 4GL, 3GL. 2GL and lGL (machine language). It shows the relationships amongst these levels, which essentially provide views of the same system at different levels of capabilities and, therefore, abstractions.
- Chapter 3 introduces the concept of the Operating System functions as provided by the various system calls. It presents the user's/application programmer's view of the Operating System and also that of the System Programmer, It shows how these are related. It discusses the system calls in three basic categories: Information Management (1M). Process Management (PM), and Memory Management (MM). It also shows the relationship between these three modules.
- Chapter 4 defines the concept of a "block" and goes on to explain how the data for a file is organized on a disk. It explains the functioning of hard and floppy disks in detail. it explains how the Operating System does the address translation from logical to physical addresses to actually Read/Write any record, It goes on to show the relationship between the Applicatiou Program (AP), Operating System *(O/S).* and the Data Management Software (DIv1S).
- Chapter 5 defines a "process" and discusses the concepts of context switching as well as multi-programming. It defines various process states and discusses different process state transitions. It gives the details of a data structure "Process Control Block (PCB),' and uses if to show how different operations on a process such as "create", "kill", or "dispatch" are implemented, each rime showing how the PCBs cbains would reflect the change. It then discusses the different methods used for scheduling various processes,
- Chapter 6 describes the problems encountered in the Inter Process Communications by taking an example of Producer-Consumer a1goritluns. It discusses various solutions that have been proposed so far, for mutual exclusion. The chapter concludes with a detailed discussion on semaphores and classic problems in Inter Process Communication.
- Chapter 7 describes and defines a deadlock and also shows how the situation can be graphically represented. It states the pre-requisites for the existence of a deadlock. It then discusses various strategies for handling deadlocks, viz. ignore, detect, recover from, prevent, and avoid. It concludes with a detailed discussion of Banker's algorithm.
- Chapter 8 discusses various Contiguous and Non-Contiguous memory allocation schemes. For all these schemes, it states the support that the Operating System expects from the hardware and then goes on to explain in detail the way the scheme is implemented.
- Security is an important aspect of any Operating System. Chapter 9 discusses the concept of security and various threats to it; and attacks on it. It then goes on to discuss how security violation can take place due to parameter passing mechanisms. It discusses computer worms, and viruses, explaining in detail. how they operate and grow, The chapter discusses various security design principles and also various protection mechanisms to enforce security,

- Chapter 10 introduces the concept of Parallel Processing and contrasts it with uniprocessing as well as distributed processing. discussing the merits and demerits of all. The chapter discusses the programming for parallel processing and also classification of computers.
- Chapter 11 defines the term "distributed processing" and contrasts centralized versus distributed processing. It also describes three ways in which the processing can be distributed, viz. distributed application. distributed data and distributed control. It takes an example to clarify these concepts.
- Chapter 12 provides a detailed case study of Windows NT and Windows 2000. Along with Linux: the Windows family of Operating Systems has become the most important concept that the technologist should know. The chapter provides a detailed description of Windows, including its architecture. design principles. and various Operating Systerus algcnrnrns/data structures.
- Chapter 13 provides a similar detailed case study of UNIX. The chapter provides a detailed description of UNIX, including its architecture. design principles, and various Operating Systerns algorithms/data structures.
- Chapter 14 is similar to Chapter 13 except that it describes Linux and not UN[X. The chapter provides notes on the differences between these two Operating Systems al appropriate places.
- Chapter IS provides a crisp overview and details of Multimedia Operating Systems. It starts with. a definition of multimedia and then examines the various issues involved in multimedia Operating Systems. It also lists the technology details associated with these types of Operating Systems and explains what is needed to enable Operating Systems to work in such realtime applications.

Every chapter provides chapter summaries and detailed questions and answers sections. making it a very valuable source of reference and teaching guide.

There were many colleagues and friends who helped me in this project. Satish Joshi, Vijay Khare, Ajay Chamania, Sunil Chitle, K. Shastri, Pradeep Kulkarni. Revi, Atul Kahate. Anand Savkar, Govind Dindore, Ashish Dhume, Cletus Pais, Joan Fernandes. Anand Kumar Pai, Pradeep Waychal, Dhiren Patel. Bhavesh Patel, Parmar, Utpal Kapadia, Jyoti Joshi, Ranjana Ranade and many others helped and encouraged me throughout this project. My late father helped me immensely by checking proofs many times. My cousin-friend Dr. Sunandan Kolhatkar has been a great source of encouragement for me not only in this project but life in general. PCS and Syntel also were very cooperative and helpful. The Tata McGraw-Hill staff was very helpful and cooperative. My wife supported this project a lot. without which it would never have been completed.

The second edition is possible basically because of Aml Kahate. He is a terrific guy! He helped me a lot in writing a few new chapters and improving the old ones. He has been a great help that would defy description. I am grateful 10 him. He was ably assisted by Mandar Khadilkar, Prasad Shembekar, and Bharat Kulkarni; whose assistance was extremely crucial.

One note of caution and request! While reading this book, the reader should keep in mind that when the book refers to "he" or "him" or "his"; it should be read as he/she or him/her or his/her. The reference in this book is purely for the sake of convenience to avoid clumsy sentences.

AcHYVT 5 GODBOLE

Preface to the First Edition

Almost everybody involved in the development of software comes in contact with the Operating System. There are two major groups of people in this context. One group is concerned with knowing how an operating system is designed, what data structures are used by an operating system and how various algorithms within an operating system are organized in various layers to execute different functions. This *is* a class of system programmers who lire required to study the internals of an operating system and later on participate in either designing or installing and managing an operating system (including performance tuning) or enhancing it by writing various device drivers to support various new devices, etc. The curriculum for an introductory course on the operating system for Bachelors and Masters degree in Science, Engineering or Management precisely intends to cover all these topics. This book is meant for these students.

The book is meant for yet another group of professionals consisting of application programmers writing programs in COBOL, C ..dBASE, ORACLE or any other third or fourth genernrion language. This group, to which I personally belonged for many years comes into contact with only the Command language (JCLs and SHELL) of tbe operating system. For this group, Ibis JCL is what constitutes the "knowledge of' or "experience on" a particular operating system. But for Ibis group. the operating system remains a mystery, despite working with it for years. How does the operating system translate the commands issued at the terminal into actual actions in tenus of machine instructions, for instance?

There are excellent books available today. which describe internal workings of an opernung system and there are also books which teach an appl ication programmer what commands to issue at a terminal to achieve a specific result. The point is to link these two levels and demystify the subject. This book aims at doing this. in a step-by-step manner, Thus, existing application programmers will stand 10 gain a lot by this hook.

The book is organized as follows;

Chapter I covers the history of operating systems, sketching mainly the important landmarks in its evolution. It is quite intimately tied to tile developments in computer hardware from one generation to another. It is for this reason that it is presented along with the history of the evolution of tbe computer generations.

Chapter 2 covers [he introduction to Computer Architecture. My experience shows that without a basic understanding of the computer architecture. operating system cannot be fully grasped. One has to be aware of CPU registers and the basic instruction execution cycles. without which the concepts of interrupts, context switching, instruction indivisibility and many others would be difficult LO understand. I have tried to fill this gap. Starting from the basics, Chapter 2 covers only the details that are necessary to visualize the functions of the operating system. It is by DO means a comprehensive discussion on this vast subject. 10 fact, it. is a bit simplistic but would serve its purpose. Those who have done a full course 00 computer architecture can skip this chapter entirely.

Chapter 3 introduces the basic functions of the operating system. It introduces the concept of a system call. It presents two basic views of the operating system -that of systems programmers and the second that of application programrners and end-users. The chapter shows how the two are related. It also introduces the three main functions of the operating system, viz., Information Management (eM). Process Management (PM) and Memory Management (MM).

Chapter 4 is devoted to Information Management. Though it talks about the management of information by any operating system in general. it discusses disks and terminals in great detail. It covers some basic information on how data is organized on the disks. It then talks about File S ystems and Device Manage- ment at length. Again, starting from the very basics, it attempts to link the user's and programmer's views. It covers directory systems. disk space allocation, and address translation algorithms, etc. It abo explains the basic principles of the terminal drivers.

Chapter 5 covers Process Management. It defines a process. discusses various process states and their transitions and also various scheduling algorithms.

Chapter 6 covers an interesting but relatively difficult topic of Interprocess Communication and Serna- phores to avoid race conditions. The attempt is to simplify the topic without losing accuracy and rigour so that it can be understood by the application programmers also.

Chapter 7 discusses Deadlocks. The reasons for deadlocks and the methods of detection, prevention and avoidance are discussed later in the chapter.

Chapter 8 discusses Memory Management in detail. It covers contiguous and noncontiguous systems, single contiguous, fixed partitioned, variable partitioned, paged, segmented, combined and virtual memory management systems,

Chapter 9 discusses the 'Security' provided by the operating systems, We know lIlat in a multi-user system, we need to protect the data and resources of one user from others. The operating system plays a very significant role in this regard with the assistance of hardware, if need be. This topic discusses the impor- tance of security, the threats to it and various ways of providing it in centralized as well as distributed processing environments. It discusses the issues of access protection mechanisms, computer virus and data encryption, etc. in detail.

Chapter 10 covers the operating systems in a Multiprocessing Systems environment, where there are multiple CPUs that can execute the instructions from one or many processes parallely. Obviously, the CPU scheduling methods and various other considerations will differ from those in a single processor system. This chapter discusses the different ways in which the multiprocessing systems can be organized and the way the operating system handles its functions in such an environment,

Chapter 11 covers the important topic, operating systems in a distributed environment. Starting from the basic concepts in distributed processing. it covers issues such as Remote procedure caUs (RPC), file services, database servers, and security and protection in distributed environment. The chapter concludes with a detailed case study of Novell NetWare as a network operating system.

Chapter 12 covers the basic concepts in Graphical user interfaces (GUI). It discusses what they are, how they function and the role of the operating system in the entire process. It discusses a case study of Win- dows NT at the end.

Chapter 13 discusses a detailed case study of UNIX. There are already excellent books on the internals of UNIX. This chapter provides, a simplified (but not simplistic) and coherent approach to various con- cepts used in UNIX. I believe, any reader going through this chapter can easily embark on a very detailed, thorough study or research of UNIX internals.

The book is a step-by-step guide for those who want to learn the topic from the very basics. Normally, a new term is not used unless it is explained beforehand. A very serious attempt has been made to present the topic in a simple and systematic way without losing its rigour, 50 that any student of computer science or an application programmer can easily understand it.

This book would not have been completed without the help of a number of people, both friends and colleagues. I would specially like to mention the names of Nandakumar Pai, Satish Joshi, Dhiren Patel, Ajay Chamania, Pradeep Kulkarni, Sanjeevaoi Vaza, Revi, Bavesh Patel, Paramjeet Singh Parmar, Atul Kahate, Ravi Kale, Urpal Kapadia, Anand Savkar, Govind Dindore, Ashish Dhurne, Cletus Pais and Joan Fernandes without whose active help. the book would never have been written. I must thank Nandakumar Pai specially for taking an enormous interest in this project. In fact, these are just a few names amongst many colleagues at Patni Computer Systems (PCS) wbo made this project a reality. For instance. colleagues like T K Rema, Sunil Chitale, Ranjana Ranade, Jyoti Joshi. Pradeep Waychal and many other gave constant encouragement for the completion of this book. My father greatly encouraged and helped in checking the manuscript. I must thank Patni Computer Systems (PCS) as well as Syntel Software Private Limited (SSPL) for allowing me to make use of many facilities at their office. 1 am extremely grateful to Dr. N Subrahmanyarn, P.K. Madhavan and Sunil Patki of Tata McGraw-Hill for their help and patience in putting up with my schedules. Their encouragement has meant a lot to me. Last but not the least, I acknowledge with gratitude the support given by Iny wife Shobha, without which this book would never have seen the light of day.

While reading this book the readers should keep in mind that when the book refers to "he" or "his". it should. be read as he/she or hislher; the reference in the book is purely for tbe sake of convenience to avoid clumsy sentences.

AS GODBOLE

History of the Operating Systems

1

Chapt"e¥Objec:t'w~ **1 ----------....**

This chapter traces the origins of the Operating Systems and their subsequent developments. It traces this on the background of the generations of computer hardware systems, starting with the zeroth generation up LO the fourth generation. It introduces the concepts of Spooling. Timesharing. Multiprogramming. IOCS, and so on. Starting from Batch Oriented Systems. it introduces the concepts behind the latest trends in Operating Systems such as Operating Systems for handheld devices. Distributed Operating Systems and Operating Systems in multiprocessing environments.

The history of Operating Systems is inextricably linked with the history and development of various generations of computer systems. In this chapter, we will trace the history of the Operating Systems by delineating the chronological development of hardware generations.

'U'U·ZERomGE"NE'iU'tIoN-

The first digital computer was designed by Charles Babbage (1791-1871), an English mathematician. It had a mechanical design where wheels. gears. cogs. etc. were used. As this computer was slow and unreliable, this design could not really become very popular. There was no question of any Operating System of any kind for this machine,

GENERATION *(i945-1955)-* VACtiUM :tuBES

Several decades later, a solution evolved which was' electronic rather than mechanical. This solution emerged out of the concerted research carried out as part of the war effort during the Second World War. Around 1945, Howard Aiken at Harvard, John Von Neumann at Princeton, J. Eckert and William Mauchely at the University of Pennsylvania, and K. Zuse in Germany succeeded in designing calculating machines with vacuum tubes as the central components.

These machines were huge and their continued use generated a great deal of heat. The vacuum tubes also used to get burnt very fast. (Doring one computer run, as many as 10,000-20,000 tubes could be wasted!) The programming was done only in machine language, which could be termed "The First Generation" language. 'This was neither an assembly language nor any higher level language. Again, there was no Operating System for these macbines too! They were single-user machines, which were extremely unfriendly to both the users and the programmers.

Around 1955, transistors were introduced ill the USA at AT&T. The problems associated with vacuum tubes vanished overnight. The size and the cost of the machine dramatically dwindled. The reliability improved. For the first time, new categories of professionals called systems analysts, designers, programmers and operators came into being as distinct entities. Until then, the functions bandied by these categories of people had been managed by a single individual.

Assembly language, as a second generation language, and FORTRAN, as a High Level Language (third generation language), emerged, and the programmer's job was extremely simplified.

However, these were batch systems. The IBM-1401 belonged to that era. There was no question of having multiple terminals attached to the machine, carrying out different inquiries. The operator was continuously busy loading or unloading cards and tapes before and after the jobs. At a time, only one job could run. At the end of one job, the operator had to dismount the tapes, take out the cards (teardown operation), load the decks of cards and mount the tapes for the new job (setup operation). This entailed the usage of a lot of computer time. Valuable CPU time was, therefore, wasted. This was the case when ffiM-1401 was in use. An improvement came when ffiM-7094-a faster and larger computer was used in conjunction with IBM-J40t. which was then used as a sateUite computer. The scheme used to work as follows:

(i) There used to be control cards giving information about the job, the user, and so on, sequentially stacked, as depicted in Fig. 1.1. For instance, $JOB specified the job to be done, the user who is doing it, and may be some other information. $LOAD signified that what would follow were the cards with executable machine instructions punched onto them and that they were to be loaded in the main memory before it could be executed. These card's were therefore, collectively known as the object deck or object program. When the programmer wrote his program in an assembly language called a source program, the assembly process carried out by a special program called assembler would convert it into an object program before it could be executed. The assembler would also punch these machine instructions on the cards in a predefined format. For instance, each card had a sequence number to help it to be rearranged in case it fell out by mistake. The column in which the opcode of the machine instruction started was also fixed (e.g. column 16 in the case of Autocoder), so that the loader could do its job easily and quickly.

The $LOAD card would essentially signify that the object cards following it should then be loaded in the memory. Obviously, the object program cards followed the $LOAD card as shown in the figure. The $RUN control card would specify that the program just then loaded should be executed by branching to the rust executable instruction specified by the programmer in the ORG statement. The program might need some data cards which then followed. SEND specified the end of the data cards and $JOB card that followed specified the beginning of the next new job.

(ii) An advantage of stacking these cards together was to reduce the efforts of the operator in 'setup' and 'teardown' operations. and therefore, to save precious CPU time. Therefore, many such jobs were stacked together one after the other as shown in Fig. 1.1.

(iii) All these cards were then read one by one and copied onto a tape using a 'card-to-tape' utility program. This was done on an lBM-1401 which was used as a satellite computer. This arrangement is shown in Fig. 1.2. Controls such as 'total number of cards read' were developed and printed by the utility program at the end of the job to ensure that all cards were read.

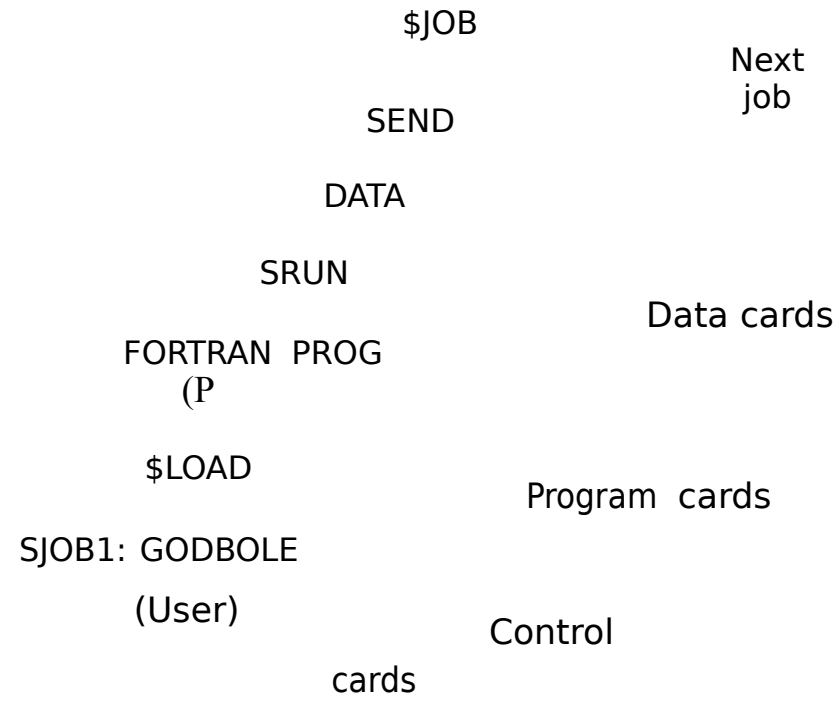


Fig. 1.1 Structure of the batch job

Job 2

Job 1

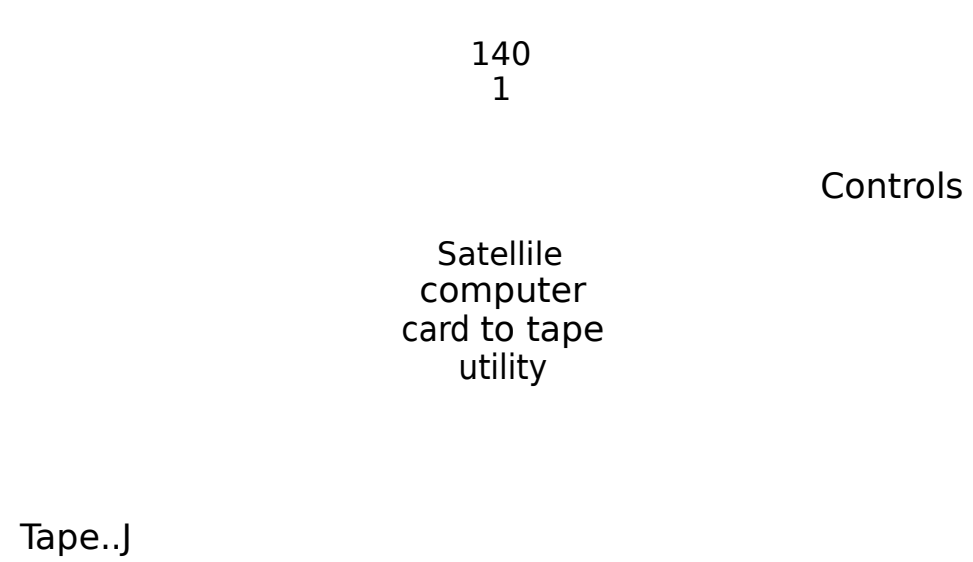


Fig. 1.2 Card-to-tape copying function on a satellite computer

(iv) The prepared tape (Tape-J shown in Fig. 1.2) was taken to the main 7094 computer and processed as shown in Fig. 1.3. The figure shows Tape-A as an input tape and Tape-B as an output tape. The printed reports were not actually printed on 7094, but the print image was dumped onto the tape (Tape-P) which was carried to slower 1401 computer again, which did the final printing as shown in Fig. 1.4. Due to this procedure, 7094 computer, which was a faster and more expensive machine was not locked up for a long time unuecessarily.

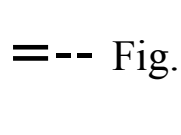


Fig. 1.3 7094 actually processes the job

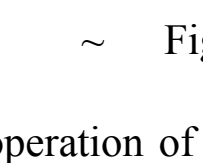


Fig. 1.4 1401 used for printing

The logic of splitting the operation of printing into two stages here was simple. The CPU of a computer was quite fast as compared to any 1/0 operation. This was so, because the CPU was a purely electronic device, whereas *1/0* involved electromechanical operations. Secondly, within two types of *1/0* operations, writing on a tape was faster than printing a line on paper. Therefore, the lime of the more powerful, more expensive 7094 was saved.

If 7094 was used to print a report, it would be idle for most of the lime. When a Linewas being printed, the CPU could not be doing anything else. This is because, the CPU can execute only one instruction at a time. Of course, some computer bad to read the print image tape (Tape-P) and print a report. But tben, that could be delegated to a relari vely less expensive satellite computer, say the 140 L. Actually, writing 00 tape and then printing on the printer appear to be wasteful and more expensi ve, but it was not so, due to the differential powers and costs of the 7094 and J 40 I.

This scheme was very efficient and improved the division of labour. Since the three operations required for the three stages shown in Figs 1.2-1.4 were repetitive, the efficiency further increased. The only difference was tbat the computer 7094 bad to have a program which read tbe card images from Tape-J and interpreted them (e.g. on hitting a $LOAD card image, it actually started loading the program from the following card image records). This was essentially a rudimentary Operating System. The ffiM-7094 bad two Operating Systems 'ffiSYS' and 'Fortran Monitor System (FMS)'.

Similarly, the ffiM-J401 had to have a program which interpreted the print images from tbe tape and actually printed the report. This program was a rudimentary spooler. One scheme was to have the exact print image 011 the tape. For instance, if there were 15 blank lines between two printed valid report lines, one would actually write 15 blank lines on the print image tape. In this case, the spooler program was very simple. All it had to do was to dump the tape records on the printer. But this scheme was clearly wasteful, because, the ffiM-7094 program had to keep writing actual blanklines; additionally, the tape utilization was poor.

A better scheme was to use special characters (which are normally not used in common reports, etc.) to denote end-of-line, end-of-page, number of lines to be skipped, and so 011. In this case, the program on the ffiM-7094 wbich created tbe print-image tape became a little more complex but far more efficient .. The actual tape was used more efficient ly, but then the spooler program also became more complex .. It bad to actually interpret the special characters on tbe tape and print the report accordingly.

This was a single user system. Only one program belonging to only one user could run at a time. When the program was reading or writing a record, the CPU was idle; and this was very expensive. Due to the electromechanical nature, the JJO operations used to be extremely time consuming as compared to the CPU operations (This is true even today despite great improvements in the speeds of the *110* devices). Theretore, during tbe complete execution of a job, the actual CPU utilization was very poor.

Despite these limitations, the rudimentary Operating System did serve the purpose of reducing operator intervention in the execution of computer jobs. 'Setup' and 'teardown' were then applicable only for a set of jobs stacked together instead of each job.

During this period, the mode of file usage was almost always sequential. Database Management Systems (DBMS) and Online systems were unheard of at that time.

One more development o.f this era was the introduction of a library of standard routines. For example, the Input Output Control System (IOCS) was developed in an assembly language of the mM-J 401, called •Autocoder'. and was supplied along with the hardware. This helped the programmers significantly because they no longer had to code these tedious and error-prone routines every rime, in their programs. The concept of ,I system call, wbere the Operating System carried out a function on behalf of the user, was still not in use. Therefore these routines in the source code had to be included along with the other source program for all programs before the assembly process. Therefore, these routines went through the assembly process every time.

An improvement over this was to predetermine the memory locations where the IOCS was expected to be loaded, and to keep the pre-assembled lOCS routines ready. They were tben added to the assembled

object program cards to be loaded by the loader along with the other object deck. This process saved the repetitive assembly of IOCS routines every time along with every source program. The source program had simply to branch to the subroutine residing at a predefined memory address to execute a specific 110 instruction.

In the early 60s. many companies such as National Cash Register (NCR). Control Data Corporation (CDC). General Electric (GE). Burroughs. Honeywell. RCA and Sperry Univac started providing their computers with Operating Systems. But these were mainly batch systems concerned primarily with throughput Transaction processing systems started emerging with the users feeling the need for more and more online processing. In fact. Burroughs was one of the few companies which produced an Operating System called the Master Control Program (MCP) which had many features of today's Operating Systems such as multiprogramming (execution of many simultaneous user programs). multiprocessing (many processors controlled by one Operating System) and virtual storage (program size allowed to be more than the available memory).

t··c")+.""".,.~,, 'o"_ .... .~

ffiM announced Systeml360 series of computers in 1964. IBM had designed various computers in this series which were mutually compatible so that the conversion efforts for programs from one machine to the other in the same family were minimal. This is how the concept of 'family of computers' carne into being. ffiM-370. 43XX, and 30XX systems belong to the same family of computers.

ffiM faced the problem of converting the existing 1401 users to Systeml36O, and there were many. IBM provided tbe customers with utilities such as si.mulators (totally software driven and therefore. a little slow) and emulators (using hardware modifications to enhance the speed at extra cost) to enable the old 1401 based software to run on the ffiM-36O family of computers.

Initially, IBM had plans for delivering only one Operating System for all the computers in the family. However, this approach proved to be practically difficult and cumbersome. The Operating System for the larger computer in the family. meant for managing larger resources. was found to create far more burden and overheads if used on the smaller computers. Similarly, the Operating System that could run efficiently on a smaller computer would not manage the resources fora large computer effectively. At least, IBM thought so at that time. Therefore, IBM was forced to deliver four Operating Systems within the same range of computers. These were:

- CP-67/CMS for the powerful *360167.* using virtual storage.
- OSIMVT for the bigger 360 systems.
- OSIMFT for the medium 360 systems.
- OOS1360 for the small 360 systems.

The major advantages/features and problems of this computer family and its Operating Systems were as follows:

Integraled Circuits

The Systeml360 was based on Integrated Circuits (ICs) rather than transistors. With ICs, the cost and the size of the computer shrank substantially. and the performance also unproved.

Portability

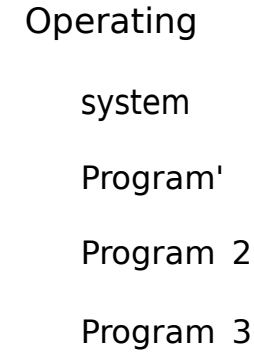
The Operating Systems for the System!360 were written in assembly language. The routines were therefore, complex and time-consuming to write and maintain. Many bugs persisted for a long time. As these were written for a specific machine and in the assembly language of that machine, they were tied to the hardware. They were not easily portable to machines with a different architecture not belonging to the same family.

Job Control Language

Despite the portability problems, the user found them acceptable, because, the operator intervention (for 'setup' and 'teardown') decreased. A Job Control Language (JCL) was developed to allow communication between the user/programmer and the computer and its Operating System. By using the JCL, a user/programmer could instruct the computer and its Operating System to perform certain tasks, in a specific sequence for creating a file, running a job or sorting a file.

Multiprogramming

The Operating System supported mainly batch programs but it made 'multiprogramming' very popular. This was a major contribution. The physical memory was divided into many partitions, each holding a separate program. One of these partitions was holding the Operating System as shown in Fig. 1.5.



"", Fig. 1.S Physical memory in multiprogramming

However, because, there was only one CPU, at a time only one program could be executed. Therefore, there was a need for a mechanism to switch the CPU from one program to the next. This is exactly what the Operating System provided.

One of the major advantages of this scheme was the increase in the throughput. If the same three programs shown in Fig. 1.5 were to run one after the other, the total elapsed time would have been much more than under a scheme which used multiprogramming, The reason was simple. In a uniprogramming environment, the CPU was idle when any I/O operation for any program was going on (and that was quite a lot I), but in a multiprogramming Operating System, when the 110 for one program was going on, the CPU was switched to another program. This allowed for the overlapped operations of 110 for one program and the other processing for some other program by the CPU, thereby increasing the throughput.

Spooling

The concept of Simultaneously Peripheral Operations Online (spooling) was fully developed during this period. This was the outgrowth of the same principle that was used in the scheme discussed earlier and depicted in Figs 1.2-1.4. The only advantage of spooling was that you no longer had to carry tapes to and fro the 1401 and 7049 machines. Under the new Operating System, all jobs in the form of cards could be read into the disk first (shown as *a* in Fig. 1.6) and later on, the Operating System would load as

many jobs in the memory. one after the other. until the available memory could accommodate them (shown as *b* in Fig. 1.6). After many programs were loaded in different partitions of tile memory. the CPU was switched from one program to another to achieve multiprogramming, We will later see different policies used to achieve this switching. Similarly, whenever any program printed something. it was not wrinen directly on the printer, but the print image of the report was written onto tbe disk in the area reserved for spooling (shown as c in Fig. 1.6). At.any convenient time later. the actual printing from this disk file could be undertaken (shown as *d* in Fig. 1.6).

---" Fig. 1.6 Spooling

Spooling had two distinct advantages. Firstly. it allowed smooth mutriprogramming operations. Imagine if two programs, say. Stores Ledger and Payslips Printing, were allowed to issue simultaneous • instructions to write directly on the printer, the kind of hilarious report that wou.ld be produced with intermingled lines from both the reports on the same page. Instead. the print images of both the reports were written on to the disk at t\VOdifferent locations of the Spool file first, and the Spooler program subsequently printed them one by one. Therefore. while printing. the printer was allocated only to the Spooler program. In order to guide this subsequent printing process, the print image copy of the report on the disk also contained some preknown special characters such as for skipping a page. These were interpreted by the Spooler program at the time of producing the actual report.

Secondly, the *VO* of aJl the jobs was essentially pooled together in the spooling method and therefore. this could be overlapped with the CPU bound computations of all the jobs at an appropriate time chosen by the Operating System to improve the throughput.

Time Sharing

The Systeml360 with its Operating Systems enhanced multiprogramming, but the Operating SysteJ11S were not geared to meet the requirements of interactive users. They were not very suitable for the query systems for example.

The reason was simple. In interactive systems, the Operating System needs to recognise a terminal as an input medium, In addition, the Operating System has to give priority to the interactive processes over
batch processes. For instance, if you fire a query on the terminal, "what is the flight time of Flight SQ024?," and the passenger has to be serviced within a brief time interval, the Operating System must

gi ve higher priority to this process than, say, for a payroll program running in the batch mode. The classical multiprogramming batch Operating Systems did not provide for this kind of scheduling of various processes.

A change was needed. ffiM responded by giving its users a program called Customer Information Control System (CICS) which essentially provided Data Communication (DC) facility between the terminal and the computer. It also scheduled various interactive users' jobs on top of the Operating System. Therefore, CICS functioned not only as a Transaction Processing (TP) monitor but also took over some functions of the Operating System such as scheduling. ffiM also provided the users with the Time Sharing Option (TSO) Software later to deal with tbe situation.

Many other vendors came up with Time Sba.ring Operating Systems during the same period. For instance. DEC came up with TOPS-IO on the DEC-IOmachine, RSTSlEandRSX-llM for the PDP-II family of computers and VMS for the VAX-II family of computers. Data General produced AOS for its l6-bit minicomputers and *AOSNS* for its 32-bit super-mini computers.

These Operating Systems could learn from the goodlbad points of the Operating System running on the Systeml360. Most of these were far more user/programmer friendly. Terminal handling was inbuilt in the Operating System. These Operating Systems provided for batch as well as online jobs by allowing both to coexist and compete for the resources, but giving higher preference to servicing the online requests.

One of the first time sharing systems was the Compatible Time Sharing System (CTSS)-developed at the Massachusetts Institute of Technology (M.I.T.). It was used on the ffiM-7094 and it supported a large number of interactive users. Time sharing became popular at once.

Multiplexed Information and Computing Service (MUL TICS) was the next one to follow. It was ajoint effort of M.I.T., Bell Labs and General Electric. The aim was to create a computer utility which could support hundreds of simultaneous time sharing users.

MUL TICS was a crucible which generated and tested almost all the important ideas and algorithms which were to be used repeatedly over several years in many Operating Systems. But the development of MUL TICS itself was very painful and expensive. Finally, Bell Labs withdrew from the project. In fact, in the process, GE gave up its computer business altogether. Despite its relative failure, MUL TICS had a tremendous influence on the design of an Operating System for many years to come.

One of the computer scientists, Ken Thompson working on the MUL TICS project through Bell Labs subsequently got hold of a PDP-7 machine which was unused. Bell Labs had already withdrawn from MULTICS. Ken Thompson hit upon the novel idea of writing a single user, stripped down version of MULTICS on PDP-7. Another computer scientist-Brian Kernighan-started calling this system "UNICS". out of fun. Later on, the name UNIX was adopted. None of these people were aware of the tremendous impact this event was to have on all the future developments. The UNIX Operating System was later ported to a larger machine, PDP-l 1/45.

There were, however, major problems in this porting. The problems arose because UNIX was written in the assembly language. A more adventurous idea struck another computer scientist-Dennis Ritchie-that of writing UNIX in a Higher Level Language (HLL). Ritchie examined all the existing HLLs and found none suitable for this task. He, in fact, designed and implemented a language called 'C' for this purpose. Finally, UNIX was written in C. Only 10% of the kernel and hardware-dependent routines where the architecture and the speed mattered were written in the assembly language for that machine. The rest of it (about 90%) was written in C. This made the job of porting the Operating System far easier. Today, to port UNIX to a new machine, you need to have a C compiler on that machine to compile

the 900/0 of the source code written in C language into the machine instructions of the target computer. You also need to rewrite, test and integrate only remaining 10% of the assembly language code on that machine. Despite this facility, the job of porting is not a trivial one. though, it is far simpler than the one for porting earlier Operating Systems.

This was a great opportunity for the hardware manufacturers. With new hardware and newer architectures, instead of writing a new Operating System each time, porting of UNIX was a far better solution. They could announce their products faster, because all the other products such as Dataliase Management Systems, Office Automation Systems, language compilers and so on could also then be easily ported, once the system calls under UNIX were known and available. After this, porting of Application Programs also became a relati vely easier task.

. Meanwhile. Bell Labs which later became AT&T .Iicensed the UNIX source code to many universities almost freely, It became very popular amongst the students who later became designers and managers of software development processes in many organisations. This was one of the main reasons for its popularity.

F<:)11>&1'HQJjW.h~:r40N-f1980"'PR~1SF.Nl~"",l;AR>G@S(M-b. .
"N'I'IOO'R1Vl'feN •

When Large Scale Integration (LSI) circuits came into existence, thousands of transistors could be packaged on a very small area of a silicon chip. A computer is made up of many units such as a CPU. memory, I/O interfaces, and so on. Each of these is further made up of different modules such as registers, adders, multiplexers, decoders and a variety of other digital circuits. Each of these, in turn, is made up of various gates (e.g. one memory location storing I-bit is made up of as many as seven gates). Those gates are implemented in digital electronics using transistors. As the size of a chip containing thousands of such transistors shrank. obviously the size of the whole computer also shrank. But thc process of interconnecting these transistors to form all the logical units became more intricate and complex. It required tremendous accuracy and reliability. Fortunately, with Computer Aided Design (CAD) techniques. one could design these circuits easily and accurately, using other computers themselves, Mass automated production techniques reduced the cost but increased the reliability of the produce computers. The era of microcomputers and Personal Computers (PC) had begun.

1.S.1 Desktop Systems

With the hardware. you obviously need the software to make it work. Fortunately, many Operating System designers on the microcomputers had not worked extensively on the larger systems and therefore, many of them were not biased in any manner. They started with fresh minds and fresh ideas to design the Operating System and otber software on them.

Control Program for Microcomputers (CPIM}-was almost the first Operating System on the microcomputer platform. It was developed on Intel 8080 in 1974 as a File System by Gary Kindall. Intel Corporation bad decided to use PLIM instead of the assembly language for the development of systems software and needed a compiler for it badly. Obviously, the compiler needed some support from some kind of utility (Operating System) to perform all the file-related operations. Therefore, CP/M was born as a very simple, single user Operating System. It was initially only a File System to support a resident PUM compiler. This was done at "Digital Research Inc. (DRl)".

After the commercial licensing of CP/M in 1975, other utilities such as editors, dcbuggers, etc. were developed, and CP/M became very popular. CPflvl went through a number of versions. Finally. a 16-bit multiuser, time sharing MPIM was designed with real time capabilities, and a genuine competition with the minicomputers started. In 1980, CPINET was released to provide networking capabilities With MPI M as the server to serve the requests received from other *CPIM* machines.

One of the reasons for the popularity of CP/M was jts userfriendUness. This had a lot of impact on all the subsequent Operating Systems on microcomputers.

After the advent of the WM-PC based on Intel 8086 and then its subsequent models, the Disk Operating System (DOS) was written. IBM's own PC-DOS, and MS-DOS by Microsoft are close cousins with very similar features. The development of PC-DOS again was related to CP/M. A company called "Seattle Computer" developed an Operating System called QDOS for Intel 8086. The main goal was to enable the programs developed under CP/M on Intel 8080 to run on Intel 8086 without any change. Intel 8086 was upward compatible to Intel 8080. QDOS, however, had to be faster than CP/M in disk operations. Microsoft Corporation was quick to realise the potential of this product, given the projected popularity of Intel 8086. It acquired the rights for QDOS which later became MS-DOS (the WM version is called PC-DOS).

MS-DOS is a single user, user-friendly Operating System, In quick succession, a number of other products such as Database Systems (dB AS E), Word Processing (WORDST AR), Spreadsheet (LOTUS .1-2-3) and many others were developed under MS-DOS, and the popularity of MS-DOS increased tremendously. The subsequent development of compilers for various High Level Languages such as BASIC, COBOL, and C added to this popularity, and, in fact, opened the gates to a more serious software development process. This was to play an important role after the advent of Local Area Networks (LANs). MS-DOS later was influenced by UNIX and it has been evolving towards UNIX over the years. Many features such as hierarchical file system have been introduced in MS- DOS over a period of time.

With the advent of Intel 80286, the IBM PClAT was announced. The hardware had the power of catering simultaneously to multiple users, despite the name "Personal Computer". Microsoft quickly adapted UNIX on this platform to announce "XENIX". WM joined hands with Microsoft again to produce a new Operating System called "OS/2". Both of these run on 286-and 386-based machines and are multi-user systems. While XENIX is almost the same as UNIX, *OS/2* is fairly different, even though influenced by MS-DOS, which runs on the IBM PCI AT as well as the PS/2.

With the advent of386 and 486 computers bit mapped graphic displays became faster and therefore, more realistic. Therefore. Graphical User Interfaces (GUIs) becamepossible and in fact necessary for every application. With the advent of GUIs, some kind of standardisation was necessary to reduce development and training time. Microsoft again reacted by producing MS-WINDOWS. MS-WINDOWS 3.x became extremely popular. However. MS-WINDQWS was actually not an Operating System. Internally it used MS-DOS to execute various system calls. On the top of DOS, however, MS-WINDOWS enabled a very user friendly GUI-(a.~ against the earlier text based ones) and also allowed windowing capability.

MS-WINOOWS did nor lend a true mulntasking capability to the Operating System. WINDOWS-NT developed a few years later incorporated this capability in addition to being windows based. *OS/2,* UNIX provided multitasking, but were not windows based. They had to be used along with Presentation Manager or X- WINDOWS/MOT(P, respectively, to achieve that capability.

In the last few years, the Intel Pentium processor and its successors have offered tremendous power and speed to the designers of the Operating Systems. As a result, new versions of the existing Operating

Systems have emerged, and have actually become quite popular. Microsoft has released Windows 2000, which is technically Windows NT Version 5.0. Microsoft had maintained two streams of its Windows family of Operating Systems-one was targetted at the desktop users. and the other was targetted at the business users and the server market. For the desktop users. Microsoft enhanced its popular Windows
3.11 Operating System to Windows 95, then 10 Windows 98, Windows ME and Windows XP. For the business users, Windows NT was developed. and its Version 4.0 had become extensively popular. This meant that Microsoft had to support. two streams of Windows Operating Systems-one was the stream of Windows *951981ME1XP,* and the other was the Windows NT stream. To bring the (\vo streams together, Microsoft developed Windows 2000, and it appears that going forward, Windows 2000 would be targened at both the desktop users, as well as the business users.

On the UNIX front, several attempts were made to take its movement forward. Of them all, the Linux Operating System has emerged as the major success story. Linux is perhaps the most popular UNIX variant available today. *Thefree software* movement has also made Linux more and more appealing.

Consequently, today, there are two major camps in the Operating System world: Microsoft Windows 2000 and Linux. It is difficult to predict which one of these would eventually emerge as the winner, However, a more likely outcome is that both would continue to be popular, and continue to compete with each other.

I.S.2 Multiprocessor Systems

The various configurations discussed so far in this chapter are examples of tile Uniprocessor Systems. The Uniprocessor Systems consist of only one CPU, memory and peripherals. However, in the last few years, l\-Iultiprocessor Systems, which consist of two or more CPUs sharing the memory and peripherals have become popular. Figure 1.7 depicts the architecture of a typical Multiprocessor System.

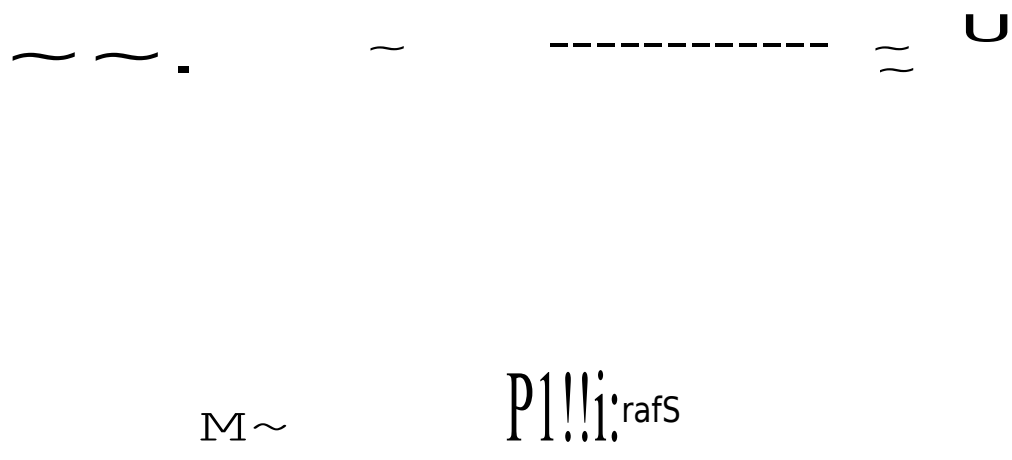


Fig. 1.7 Multiprocessor System

The Multiprocessor Systems have potential to provide much greater system throughput than their uniprocessor counterparts because multiple programs can run concurrently on different processors. However, it must be noted that some overhead is incurred during dividing a task amongst many CPUs as well as during contention for shared resources ..As a result, the speedup obtained by using Multiprocessor Systems is not linear, that is, it is not directly proportional to the number of CPU s.

A Multiprocessor System can be implemented in one of the two ways-Master-8lave architecture and Symmetric Multiprocessing (SMP) architecture. 10 the Master-Slave architecture. one processor, known as Master Processor assumes overall control on the system. It allocates work to all the slave

processors. The SMP architecture, which is most common architecture for Multiprocessor Systems, employs a different mechanism. No master-slave relationship exists in the SMP architecture. Instead, all the processors operate at the same level of hierarchy and run an identical copy of the underlying operating system. While this sounds very easy in theory, utmost care needs to be taken while designing the Operating System for SMP architecture. It is the job of the Operating System to ensure isolation, fairly equal work distribution amongst all the processors and protection from possible corruption by not allowing two or more programs to write to the same memory location simultaneously.

Nearly all the modem Operating Systems like Windows 2000, Linux, Solaris, and Mac OS X provide built-in SMP support.

1.5.3 Distributed Processing

With the era of smaller but powerful computers. Distributed Processing started becoming a reality. Instead of a centralised large computer, the trend towards having a number of smaller systems at different work sites but connected through a network became stronger.

There were two responses to this development. One was Network Operating System (NOS) and the other. Distributed Operating System (DOS). There is a fundamental difference between the two. In Network Operating System. the users are aware that there are several computers connected to each other via a network. They also know that there are various databases and files on one or more disks and also the addresses where they reside. But they want to share the data on those disks. Similarly. there is one or more printers shared by various users logged on 10 different computers. NOVELL's NetWare 286 and the subsequent NetWare 386 Operating Systems fall in this category. In this case, if a user wants to access a database on some other computer, he has to explicitly state its address.

Distributed Operating System, on the other hand. represents a leap forward. It makes the whole network transparent to the users. The databases, files, printers. and other resources are shared amongst a number of users actually working on different machines, but who are not necessarily aware of such sharing. Distributed systems appear to be simple, but they actually are not. Quite often, distributed systems allow parallelisms i.e, they find out whether a program can be segmented into different tasks which can then be run simultaneously on different machines. On the top of it, the Operating System must hide the hardware differences which exist in different computers connected to each other. Normally, distributed systems have to provide for high level of fault tolerance, so that if one computer is down. the Operating System could schedule the tasks on the other computers. This is an area in which the substantial research is still going on. This clearly is the future direction in the Operating System technology .

1.5.4 Clustered S.vstems

Clustered Systems combine best features of both Distributed Operating Systems and Multiprocessor Systems. Although there is no concrete definition of Clustered Systems, we will refer to a group of connected computers working together as one unit as a Clustered System. The connected computer systems can be either Uniprocessor or Multiprocessor Systems. Clustered Systems were originally developed by the DEC in the late 1980s for the VAXNMS Operating System.

Clustered Systems provide an excellent price-performance ratio. A system consisting of hundreds of nodes can easily give a traditional supercomputer a run for money. An example of such system is System X assembled by Virginia Tech University in 2003 using 1100 Apple Macintosh G5 computers. It is capable of performing a whopping 10 trillion (10.000,000,000,000) floating point operations per sec-

ond. Yet, the cost of the system is about $5 million, which is much cheaper than a traditional supercomputer.

Clustered Systems also.pro.vide excellent fault tolerance, which is the ability to.continue operation at an acceptable quality despite an unexpected hardware or software failure. The cluster software running on tDP of the clustered nodes monitors one Dr more nodes. If a node fails, then the monitoring node acquires the resources of the failed node and resumes operations.

One of the most popular implementation of Clustered Systems is Beowulf Clusters which is a group of mostly identical PCs running an Open Source Operating System such as Linux and a cluster management software running on top to implement parallelism. Apart from this, many ether commercial clustering products like Grid Engine, Open S51 are available and an extensive research and development is going on in this area.

. 1.5.5 **Handheld** Systems

The quest fer smaller size of Personal Computers has spawned an entirely new type of systems over the years, known as **Handheld** SysteDL~.The Newton MessagePad released by Apple Computers in 1993 heralded the era of Handheld Systems. Today, these systems encompass a vast category of devices like Personal Digital Assistant (PDA), Handheld Personal Computer (HPC), Pocket PC and even modern cellular phones with network connectivity. These systems are small computers with applications such as word processing, spreadsheets, personal organizers, and calculators. One major advantage of using these systems is their ability to synchronize data with desktop computers.

Compared to.desktop computers, the Handheld Systems have much smaller memory, smaller display screen and slower processor, As a result, the designers of the Operating Systems for these systems are faced with often-contradictory requirements of managing memory and processing power very efficiently yet providing rich GUI.

The two. most popular Operating Systems for handheld systems today are Palm OS frOIDPalmsource and Windows CE from Microsoft. Palm OS is shipped with one of the most commonly used PDAs, Palm Pilot. Windows CE, accompanying the rival PDAs and HPCs, allows the ease and familiarity of a typical desktop Windows system like Windows 95 and includes scaled down versions of popular Microsoft applications like Pocket Word, Pocket Excel, Pocket Powerl'oinr etc. Recently, Linux has been success- fully ported to handheld devices and it is steadily making inroads with a number of handheld device manufacturers announcing support for Linux.

~EY TERMS.

- Assembler
- Control Cards
- Data Communication
- Distributed Processing
- Family of Computers
- [nput/Output Control System
- lob Control Language
- Multiplexed Information and Computing Service
- Multiprogramming
- Object Deck
- Online Systems
- Compatible Time Sharing System
- Data Base Management System
- Distributed Operating System
- Emulators
- Graphical User Interface
- Integrated Circuits
- Large Scale Integration
- MUltiprocessing
- Network Operating System
- Object Program
- Portable

- Computers
- Satellite
- Simulators
- Setup Operation
- Source Program

- Spooler
-
- System Call
- Throughput.

» Time Sharing Option Software
- Transaction Processing Manager
- Virtual Storage

Spooling

- Teardown Operation
- Time Sharing Operating Systems

- Transaction Processing
» Userfriendliness

The first digital computer was designed by Charles Babbage

» e (1732-1871). an English mathemati- cian, This computer was made of mechanical parts like wheels. gears. cogs, ClC.

» Around 1945, Howard Aiken at Harvard. John Von Neumann at Princeton, 1. Eckert and William Mauchely at the University of Pennsylvania. and K. Zuse in Germany succeeded in designing calculating machines which used vacuum tubes as the central components.

)- Around 1955, the AT&T Bell Lab introduced tran.q~tors by replacing the vacuum tubes, II solved all the problems associated with vacuum tubes. The size and the cost of the machine reduced.

)- Ajob is a complete task comprised of a set of operations. When more than one task is carried out one after the other. it is called as a batch system.

» To unload the cards of one job is called a tear down operation.

» To load or mount the tapes for new job is called setup operation.

» The programmers write the instructions to be executed in a special language called the assembly language. This program is called as source program.

)- A special program called the assembler performs an assembly process which converts the source program in a predefined executable format. 11 is called as object deck or object program.

» loes was a set of standard library routine which was the readymadc programs for *UO* operations and can be included in any program al the time of assembling.

,. The system caU is an instruction to the operating system to carry out a funcnon on behalf of the user.

)- Multiprogramming is a feature of Operating Systems, which enables execution of more than one program simultaneously.

)- Multiprocessing is a feature of OS where many processors area are used and controlled by one operating system.

» Virtual storage is the memory which is not real and appears to the user to be more than the available memory.

» Family of c:omputers is the series of mutually compatible computers in which conversion of a program for one system can be easily done with minimal or no changes.

» Simulators are totally software driven utility programs which are used to convert software written for one system to run on another.

)- Emulators are partly hardware driven utility programs which are used to COli vert software written for one system to run on another.

,. The System/360 computer was based on Integrated CIrcuits (ICs) rather than transistors, [t reduced the cost and the size of the computer substantially and improved the performance .

')0 Multiplexed Information and Computing Service (MULTICS) was a joint effort of M.I.T .• Bell Labs and General Electric to create a computer utility which could support hundreds of simultaneous rime sharing users.

)- Portability is the ability of software where it can be used on different machines with different hardware architecture without any modification.

)- In multiprogrammiag environmeut, the physical memory is divided into many partitions usually each holding a separate program.

» Throughput is the amount of output gained per unit time.
» Spooling is a feature of operating systems in which the output of any program is not printed directly on printer but dumped onto a reserved area of a disc caUed the spool area.
» An interactive system is a system in which (here is continuous interaction between user and the system.
» The first time sharing systems was Compatible Time Sharing System (CTSS) developed at the Masachaseus Institute ofTecbnology (M.l.T.).1I was used on the IBM· 7094 and it supported a large number of interactive users.
» Time Sharing Operating System allows a number of users to execute their programs on the same machine by sharing time a0100g the users.
)- Time Sharing Option (TSO) Software is the software used by the terminal user to connect to a time sharing system.
» Ken TholDpson of Bell Labs developed the operating system UNIX which was written in assembly language.
» Dennis Ritchie developed the UNIX operating system in a High level Language C to make it portable.
» Large Seale Integration (LSI) circuits are packaged with thousands of transistors on a small area of silicon chip. This shrinks the size of computer but increases the speed tremendously.
» Control Program for Microcomputers (CP/M) was the first Operating System on tbe microcomputer platform. It was developed on Intel 8080 in J 974 by Gary Kindall.
» !\-IS-00S is a single user. user-friendly Operating System, owned by Microsoft Corporation.
» Microsoft developed XENIX and OS/2 for [BI\{ PC!AT based on Intel 286 and 386 processors.
» MiCTOSOJ't developed the popular operating system !\-IS-Windows featuring Graphical User Interlace.
» In Distributed processing, instead of a centralised large computer, a number of smaller systems at different work sites but connected through a network are used.
» In Network Operating System, the users are aware of the other computers connected in the network.
» Distributed Operating SystelD makes the whole network transparent to. the users.
» Linux Operating System is the most popular UNIX variant and is a *free software.*
» Today, there are two major camps in the Operating System world: Microsoft Windows 2000 and Linux.

True or False Questions

1.1. Programmers enjoy working on first generation of computers due to. their flexibility and friendliness.
1.2 An assembler is used for the conversion of assembly language program into. machine language program,
1.3 Previous generations of computers supported multiprogramming,
1.4 With tbe introduction of Integrated Circuits (ICs). the performance of system degrades and the size of system grows up significantly.
1.5 Introduction of multiprogramming and spooling increases tbe throughput of CPU.

1.6 Portability means same program running on machines with different architecture.
1.7 Multiprogramming and multiprocessing are one and the same concept.
1.8 Spooling provides the advantage of multiprogramming.

Multiple Choice Questions

1.1 The programming of first generation computers was done in
_ (a) Assembly language (b) Machine language
(c) High level language (d) C language
1.2 In earlier generation of computer programs, processing of program was _
(a) Batch processing (b) Real-time processing
(c) Multiprogramming (d) Online processing
1.3 Multiprogramming is _
(a) Single program executing on a machine
(b) More than one program executing on a machine
(c) Single program executing on more than one machine
(d) More than one program executing on multiple machines
1.4 Hardware used to check the execution of program is
_ (a) Emulator (b) Simulator
(c) Debugger (d) Compiler
1.5 is used to establish communication between the user program and the operating system.
(a) C language (b) Assembly language
(c) Job Control Language (d) Machine language
1.6 In the entire network is transparent to the user.
(a) NOS (b) DOS
(c) Windows (d) Linux
1.7 routines were developed to perform the I/O operation which minimize the over- head of coding these routines into programs.
(a) IOCS (b) Interrupt Service Routine (ISR)
(c) CTSS (d) DBMS

Test Questions

1.1 Describe the way the computer jobs were being processed before the advent of an Operating System.
1.2 What is meaot by 'teardowo' and 'setup' operations'! How did they affect the througbput and CPU utilization?
1.3 What was the use of a 'Satellite Computer'? Give an example to illustrate this.
1.4 Explain what is meant by 'Job Control Language (JCL), and how it affected the CPU utilization and throughput?
1.5 What was IOCS? How was it a step towards an Operating System?
1.6 Discuss multiprogramming versus siogle user systems in terms of throughput and CPU utilization.

1.7 What is spooling'?
1.8 Contrast Multiprogramming batch and Time sharing Operating System.
1.9 How did UNIX originate? Why did it become so popular?
1.10 Contrast Network and Distributed Operating Systems.
1.11 Describe various generations of hardware and prograrruning languages in conjunction with the development of Operating Systems.
1.12 Explain the evolution of the Operating Systems from simple batch processing system to today's standard Operating Systems.

Computer Architecture 2

Chapt"eY <9bjec:t'We1T **1** ----------...

This chapter begins with an overview of programming language levels, followed by a brief discussion on each of these levels, *viz.* 4GL, 3GL, 2GL and IGL (machine language). II shows the relationships hetween these levels. which essentially provide views of the same system at different levels of capabilities and, therefore. abstractions.

The chapter further talks about an assembly process to convert the assembly level (2GL) program into a machine level (lGL) program. In order to do this. it introduces tbe concept of instruction formats by taking an example of a hypothetical computer. It also talks about the process of loading and relocation.

With the help of the same hypothetical computer. the chapter then explains the concepts of ALU, CPU, Registers and the Buses. and how a typical machine instruction is executed in the fetch and execute cycles.

The chapter concludes with an explanation of the 'context' of a program and how the interrupts are handled.

Computer architecture is a very vast subject and cannot be covered in great detail in a smaU chapter in the book on Operating Systems. However. no book on Operating Systems can be complete unless it touches upon the subject of computer architecture. This is because Operating Systems are intimately connected to the computer architecture. In fact, the Operating System has to be designed, taking into account various architectural issues.

For instance. the Operating System is concerned with the way instruction is executed and the concept of instruction indivisibility. The Operating System is also concerned with **interrupts:** What they are and how they are handled. The Operating System is concerned with the **organization** of memory into a hierarchy, i.e. disk, main memory, cache memory and CPU registers. Normally at the beginning of any

program. the data resides on the disk, because the entire data is too large to be held in the main mem .ory permanently. During the execution of a program, a record of interest is brought from [he disk into the main memory, If the data is going to be required quite often. it can be moved further up to the cache memory if available. Cache can be regarded as a faster memory. However, no arithmetic or logical operations such as add or cO.lnpareor even data movement operations can be carried our unless and until the data is moved from the memory to the CPU registers finally. This is because, the circuits to carry out these functions are complex and expensive. They cannot be provided between any two memory locations randomly, They are provided only for a few locations which we call CPU registers. The circuits are actually housed in a unit called Arithmetic and Logical Unit (ALU) to which the CPU registers are connected as we shall see.

The point is: who decides what data resides where? It is the Operating Systems which takes this important decision of which data resides at what level in this hierarchy. It also controls the periodic movements between them. The Operating System takes the help of the concept of Direct M.emory Access (DMA) which forms the very foundation of multiprogramming. Finally, the Operating System is also concerned with parallelism. F~r instance, if the system has multiple CPUs (multiprocessing system), the philosophy that the Operating'S ystem employs for scheduling various processes changes.

The Operating System, in fact, makes a number of demands on the hardware 1"0 function properly. For instance, if the virtual memory management system bas to work properly. the hardware must keep a track of which pages in a program are being referenced more often/more recently and which are not, or which pages have been modified.

We will present an overview of computer architecture, limited to what a student of any Operating Systems needs to be aware of. As we knew, the hardware and software of a computer are organized in a number of layers. At each layer, a programmer forms a certain view of the computer. This, in fact, is what is normally termed as the level of a programming language, which implies the capabilities and limitations .0f the hardware/software of the system at a given level, This structured view helps us to understand various levels and layers cornprehensi vely in a step-by-step fashion. For instance, a manager who issues a 'SOL' instruction to 'produce the Sales Summary report' does not specify which files/ databases are to be used to produce this report or how it is to be produced. He just mentions his basic requirements. Therefore. it is completely non-procedural. A non-procedural language allows the user to specify *what* he wants rather than *how* it is (.0 be done. A procedural language has to specify both of these aspects.

. ..

A 4GL programmer (e.g. a person programming in ORACLE, SYBASE) has to be bothered about which databases are to be used, how the screens should be designed and the logic with which the sales summary is ro be produced. Therefore, a 4GL program is not completely non-procedural, though almost all vendors of the so-called 4GLs claim that they are. As of today, the 4GLs are in between completely procedural and completely non-procedural languages. Today's 4GLs have a lot of non-procedural elements built into them. Fer instance, they can have an instruction to the effect 'Print a list of all invoices for all customers belonging to a state XYZ and where the invoice amount is >SOO and the list should contain invoice number, invoice amount and the invoice date'.

A 3GL program is completely procedural. COBOL, FORTRAN, C and BASIC are examples of 3G1.11I.n these languages. you specify in detail not only what you want, but also how it is to be achieved. For instance, the same 4GL instruction described in Sec. 2.2 could give rise to the 3GL program carrying out the following steps:

I. Until it is end of invoice file, do the following:
2. Read an invoice record.
3. If the invoice amount <= 500, bypass the record; go to the next one; else proceed further.
4. Extract the customer code from the invoice record.
5. Access the customer record for that customer code by making a database call.
6. Extract the state from the customer record.
7. If state is not = "XYZ", bypass that invoice record and go to the next one; else proceed further.
8. If this is a desired record, extract invoice number, date and amount from the invoice record.
9. Print a line to the report.

It can be seen that one 4GL instruction can give rise to a number of 3GL instructions. This is the reason why a 4GL is considered to be at a 'higher' level vis-a-vis 3GL. Despite this, a 3GL, such as COBOL or C, also presents a level of abstraction to the programmer. For instance, in COBOL you can move a record of 2000 bytes from one location in the memory to another location-in one instruction. You can carry out a number of instructions iteratively by using a 'while ... do' or 'repeat ... until' construct. A 3GL also deals with symbolic names which are English-like and therefore, easy to use. Therefore, data and paragraph names can be easier to use and remember, This helps in the development and maintenance of software. In short, the 3G1. is still close to the programmer than the hardware. Therefore, if the hardware has certain limitations, it is still possible to imagine that one could use multiple such lower level instructions to simulate one 3GL instruction. This is also the reason why, on two different machines with different capabilities, you could write suitable compilers for the same 3GL, so that it can work on both the machines. Therefore, when one defines a 3GL, no specific computer hardware has to be kept in mind. In this sense, it is hardware independent.

A 2GL or Assembly Language (AL), on the other band, is very close to the hardware, and is therefore, restricted by the capabilities of the hardware. The assembly programmer knows and bas to know that a computer has a Central Processing Unit (CPU) which has an Arithmetic and Logical Unit (ALU), a Control Unit (CU) and a few CPU registers. The CPU registers are like any other memory location, except tbat they are connected together with the ALU circuits to allow certain operations at assembly programming level, For instance, there are normally AL instructions to add two CPU registers, producing the results in a third one, or to carry out logical operations such as 'AND' or 'OR' upon two registers producing a third one, or to compare two CPU registers. These operations are not directly possible between two memory locations.

If you have to compare t\VOvalues in HLL, you could say, 'IF Quantity-A = Quantity-B'. In ALs, normally you have to move these quantities to two CPU registers and then compare. The hardware is constructed such that a certain bit treated as a flag or a condition code in a flags register or a condition

code register is set on or off depending upon the result of the comparison. The condition code register also is one of the special CPU registers. If two character strings of, say tOOcharacters each are to be compared, it is an easy task for a HLL (or a 3GL) programmer. In AL, however, the strings have to be moved to the CPU registers word-by-word, compared successively, storing the flag bit somewhere each time, and in the end, deciding. based on all the flag bits, whether both the strings were completely matching or nOI. AU these details are hidden from a HLL programmer.

Data movement instruction in an assembly language can normally move only a certain number of bytes at a rime such as a word. In many ALs, you can not move data directly between two memory locations. You have 10 copy the data from the source memory location to the CPU register by a load operation and then copy that CPU register to the destination memory location by issuing a separate AL instruction such as store. Therefore, in such cases, we will need to execute a few AL instructions a number of times repetitively, in a loop. to simulate the HLL instruction to move a 2000 byte record from one place to another. In some more advanced ALs, it is possible to have an instruction to move data between thc memory locations directly.

In such cases, the hardware circuit itself keeps moving the data, word-by-word, from the same memory address to the CPU register and then moves it subsequently each time to the corresponding words in the target memory address. The AL programmer need not be aware of this, because his AL is more 'powerful' . However, regardless of the sophistication of the AL, in any computer, ultimately the data has to be internally moved or routed through the CPU registers. It is ooly a question of the level of abstraction that the AL provides to the programmer,

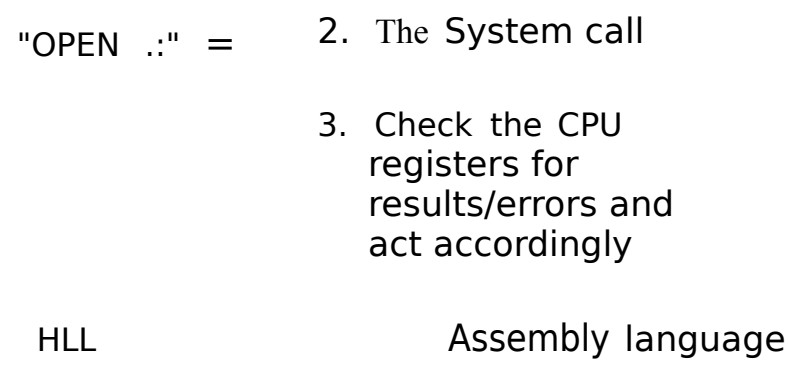
Similarly, an instruction in HLL such as "Compute A = (B*C) + (D*E)IF - G" does not have a corresponding instruction in most of the ALs. The AL normally provides for simple arithmetic instructions such as to add, subtract, multiply and divide two numbers ooly and one operation ala time. Also, normally, the operation is allowed on either two CPU registers or one CPU register and another memory location (though internally, the addition has to take place ultimately only between two CPU registers and therefore in such a case, the hardware itself will have to bring the data from the memory location into another CPU register. before the addition can actually be carried out). Therefore, depending upon the sophistication. abstraction or the 'level' within the AL, we will need different numbers of AL instructions to simulate the same HLL instruction given above. As this job of translation of the 3GL instruction is done by the compiler, we will need different compilers for machines with different architectures ..

The same is true about the flow control instructions. AL does not have an instruction like "while ... do" or "repeat ... until" or "if ... then ... else". Therefore. it is not possible to write a structured program in AL in a strict sense. AL provides for conditional and unconditional jump instructions which have to be used to simulate the "if ... then ... else" and other constructs.

The HLL or any 3GL allows for an instruction to Read or Write the data from/to an external device. These instructions are called external data movement instructions. AL does not provide for such a direct instructions. For these instructions. requests in the form of system calls have to be made to the Operating System. AU the *TIO* functions have to be carried out ultimately by the Operating System for reasons of complexity and security. Therefore. corresponding to the HLL instruction, "OPEN FILE-A". the assembly programmer has to code the instructions (o load the CPU registers with the parameters (in the case of OPEN, they will be file name, mode, access rights. etc.). He then will have to issue an appropriate system call to the Operating System.

The Operating System executes the system call using the parameters in the predefined CPU registers. It then loads some CPU registers with the outcome of the system call. This could be in the form of the results or the error indicators, if the system call has failed (e.g, the file to be opened was non-existent). The assembly programmer has to test these CPU registers for results/errors after the system call, and then take the appropriate action. This can be shown diagrammatically as shown in Fig. 2.1.

1. Load CPU registers with Parameters (preparatory)



....:;;;F, ig. 2.1 A system call in assembly language

A 2GL assembly language has an instruction "ISR" to jump to a subroutine and a corresponding "Return" instruction. This is equivalent to a "Perform" in COBOL or "GOSUB" in BASIC. As soon as the "ISR" instruction is encountered, the system "remembers" the point from where the program should continue after returning from the subroutine. As soon as it encounters the "Return" instruction, it uses this "knowledge" to return to that address. (Internally, it uses a stack memory/register to store these return addresses as we shall see a little later.)

The lGL corresponds to the machine language. There is a one-to-one correspondence between the assembly and machine instructions. For example, take an assembly language instruction "ADD RO, TOT' where ADD is an Operation Code (Opcode), RO is a CPU Register, and TOT is a symbolic name denoting, may be, the total. This instruction is supposed to add the two numbers in RO and TOT respectively and deposit the result again in RO. There is a program called assembler which converts the assembly language program into a machine language program. We will study this in next section.

2.5.1 Assembler

After the assembly language program is written, it has to be converted into a machine language program by using another piece of software called the Assembler. The Assembler starts generating the instructions or defining the data as the data/instructions are encountered, starting from 0 as the address, unless specified by the "ORG" instruction.

2.S.2 Instruction Format

Each machine instruction has a predefined format. In a simplified view, the format could be as shown in
Fig. 2.2 for our hypothetical computer.

Opcode bits	Addressing Mode bits	Register bits	Address .. bits
4	1	1	10

"~",..Fig. 2.2 A simplified instruction format

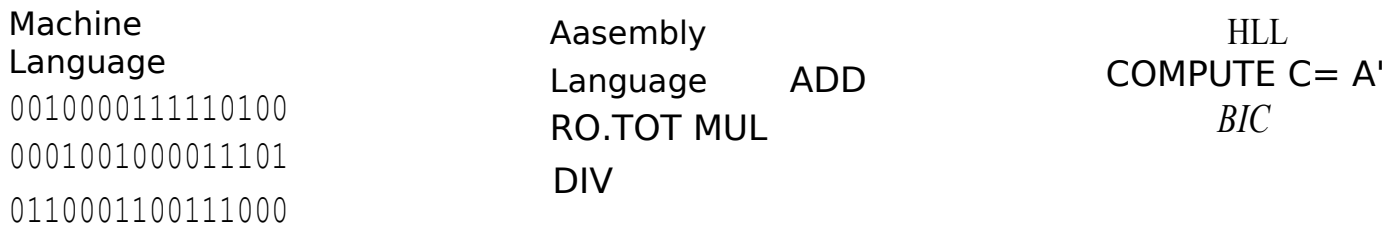
In this case, the instruction consists of 16 bits divided into different pans as shown in the figure. If there arc 16 different opcodes such as "Add" that are possible, the 4 bits in the Operation Code (opcode) specify the chosen opcode, e.g. 0000 could mean "Load", 0001 could be "Store", 0010 could be "Add" and 1111 may mean "Halt".

The addressing mode specifies whether the address in the instruction is a direct address or an indirect address. If it is direct (Mode = 0), it indicates that the memory location whose address is in the instruction (the last field in Fig. 2.2) is to be operated upon (e.g. added to RO in our instruction "Add RO, TOT"). If it is indirect (Mode = I), it means that the address in the instruction points to a memory location which does not actually contain the data, but it, in tum. contains the address of the memory location to be operated upon. In our example, we have assumed a direct address.

The register number specifies whether RO(bit = 0) or R I (bit = I) is to be considered in the operation. assuming that there are only two registers. (If there were four, for instance, we would need 2 bits to specify this register number.)

The memory address is in binary. For instance. in our assembly instruction, "TOT" is a counter to accumulate the totals. Let us assume that it is at memory location with address = 500. It will, therefore, be mentioned as "0111110100" in the instruction. because this is the binary representation of 500 in decimal. Therefore. our instruction "ADD RO. TOT" would be converted to "0010000111110100" in machine language. This is becauseOO10= ADD. 0= Direct Addressing. 0= ROand 0111110100 = 500.

This is bow an assembly language (20L) instruction would be converted into a machine language program by an assembler in a predefined instruction format. The machine language program would look as shown in Fig. 2.3. As the figure shows, one HLL instruction is equivalent to many assembly instructions. However, there is one and only one machine instruction for each assembly instruction.



"!I'-"?- Fig. 2.3 A typical machine language program

In our example, each of these instructions occupies 16 bits or 2-byt.es. The assembler starts defining data or generating instructions from 0 as the starting address, unless specified by the "ORO" statement. Knowing the length of each data item defined in the program, it can arrive at the addresses of the "next" data item. This is how, it arrives at the addresses of all the data items. This is also how the symbol table is built using the data names and their addresses. For the instructions, knowing the length of each instruction based on the opcode and the type of instructions, it can calculate the starting address of each

instruction as well. If the instruction bas a label, tbe assembler adds the label as well as the instruction addresses into the symbol table.

The assembler can be written as a one-pass or two-pass assembler, though it normally is designed as a two-pass one. In this scheme, the assembler goes through the assembly program and converts whatever it can convert to the machine language. It also generates addresses for the data items as well as the instructions, and builds a symbol table. [t cannot generate addresses for any forward references during the first pass, e.g. a PERFORM or a GOSUB starement with label yet to be encountered. In the second pass, it resolves these remaining addresses and completes its task. This is possible, because, these forward addresses will have been generated during tbe first pass. .

This assembled machine language program can be stored as a file on tbe disk. A program written in any HLL or a 3GL requires a compiler to generate. the machine code. In this case, typically, multiple instructions are generated for each instruction in the HLL. Therefore, there is no one-to-one correspondence. between the HLL and machine language, unlike in the case of AL. Ultimately, any program has to be in the binary machine format before it can be executed.

2.S.3 Loading/relocation

At tbe time of execution. the machine language program bas to be brought into the memory from tbe disk and then loaded at the appropriate locations. This is done by a piece of software called loader. The loader consults the Memory Management Unit to find out which memory locations are free to accommodate this program. Therefore, the actual starting address of the program could be 1000 instead of 0. This means Ulat each address in every instruction of this program has to be changed (relocated) by adding 1000 to it. This process. known as relocation, can be done either in tbe beginning only before the actual execution (static. relocation), 01' at the run time for each instruction dynamically (dynanllc relocation). Therefore, the assembly/machine language view, or the perspective of the computer architecture is as foHows:

(i) The computer consists of the CPU and the main memory.

(ii) Tbe CPU consists of the Arithmetic and Logical Unit (ALU), the Control Unit *(CV)* and the CPU registers.

(iii) All arithmetic operations are performed within the ALU. Therefore, no arithmetic operations are possible between the two memory locations directly, leave alone on the disk itself. If one wants to add fWO numbers in the memory. these two numbers have to be brought to tbe CPU registers first and then added. One could imagine an assembly instruction to add the contents of the two memory locations directly, but internally. only the ALU can add the data into two CPU registers.

(iv) The same is true in the case of data movement instructions. If data has to be moved from any memory word to any other memory word, the connections between the memory locations would be very complex.

Data is therefore, moved only between the CPU registers lind the memory. Therefore, the data movement between any two memory locations take place in two steps:

- Load data frOID the source memory location to a CPU register (Load)
- Store data from the CPU register to the target memory location (Store)

If 2000 bytes in a record are to be moved from a SOIU'CC memory address to the target memory address. the operation actually internally takes place as 2000 operations, if 8 bits or 1 byte is transferred at a time, This is the case when the CPU registers and the data bus typically consists of 8 bits. If the data bus and the CPU registers are I 6-bit wide, it will take only L000 operations to move 2000 bytes. as 2 bytes or L 6

bits can be moved in each operation. 10 a machine with a data bus of 32 bits, it will take only 500 operations. Therefore, compiler of any HLL will generate only appropriate number of machine instructions for such an instruction. This will also clarify why a machine with a data bus of 32 bits is faster than the one with the data bus of 8 bits only. However. in all these cases. each operation consists of two suboperarions (Load and Store), as gi ven above. This picture is true. even if the assembly language has an instruction 10 move a block of bytes between memory locations (Block Move Instruction). Internally, it has to be executed as a number of operations and suboperations in a loop until all the bytes are transferred. This loop has to be managed by the hardware itself. thereby making the hardware architecture more complex and expensive, as compared to the one where only one word can be moved at a time and therefore. the assembly programmer has to essentially set up tbis loop.

There is, therefore, a trade-off in this choice. However, with the hardware becoming more and more powerful and yet cheaper and cheaper. providing a block move instruction at the assembly language level is easily possible.

The only deviation from this trend is advent of Reduced Instructions Set Computers (RISC) machines where, you provide only a limited set of simple instructions. However. a detailed discussion of RlSC is beyond the scope of the current text.

2.6.1 Basic Concepts

This is the level at which we want to study the actual computer architecture. Certain CPU registers known as General Purpose Registers (e.g. RO. RI •...• RII) are accessible to the assembly language programmer.In addition, the CPU has a number of 'hidden' registers as shown in Fig. 2.4 which shows the architecture for our hypothetical computer with the instruction format shown in Fig. 2.2.

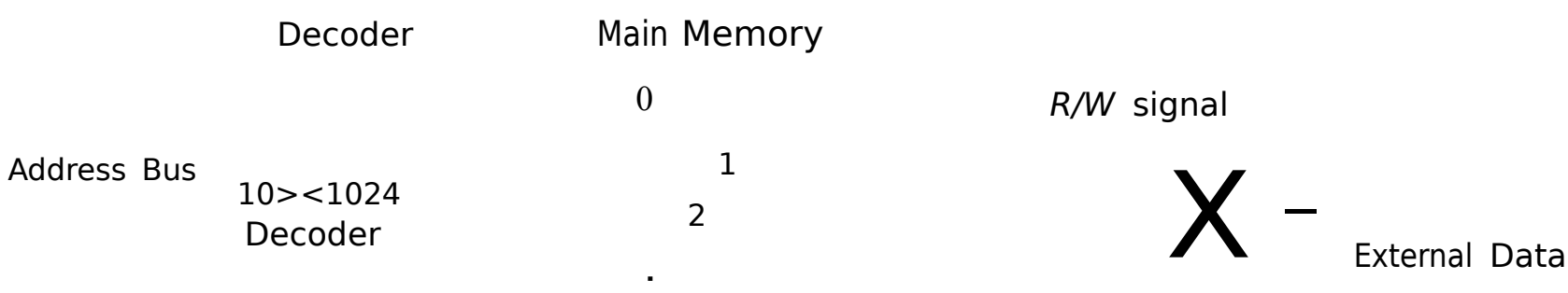
A computer consists of three buses as listed below:

- Data Bus (External and Internal)
- Address Bus
- Control Bus

The external data bus connects the memory with a CPU register called Memory Buffer Register (MBR) or Memory Data Register (MDR). The internal data bus connects all the CPU registers. including MBR. This implies that if any data has to be loaded from the memory 10 any CPU register. say RO, it has to be brought into MBR first through the external data bus and then moved to a CPU register such as RO using the internal data bus as shown in Fig. 2.9. The *store* operation stores a CPU register into a specified memory location. This operation also takes place in the two substeps as given above except that it will be in the reverse direction. All these data movements can be achieved by manipulating the switches SO, S J. etc. shown in the figure.

In some architectures. the external and internal data buses are combined into a single data bus, in which case, theoretically, there is no need for a register such as MBR. This is because, in this case. data front any memory location can be moved to any CPU register directly and vice versa without the mediation of MBR. However, we will assume the presence of MBR and two separate data buses, as shown in the figure.

The address bus carries the address of the memory location to be accessed. If the data bus is 16-bit wide, it will carry a word of2 bytes at a time. Therefore, the words in the memory will have. addresses *0,1, ..., //* where $n = 1/2$ of the total number of bytes in the memory. This is what we mean by addressable location in our discussion. If there are 1024 locations (numbered as 0 to 1023 in the figure) to be



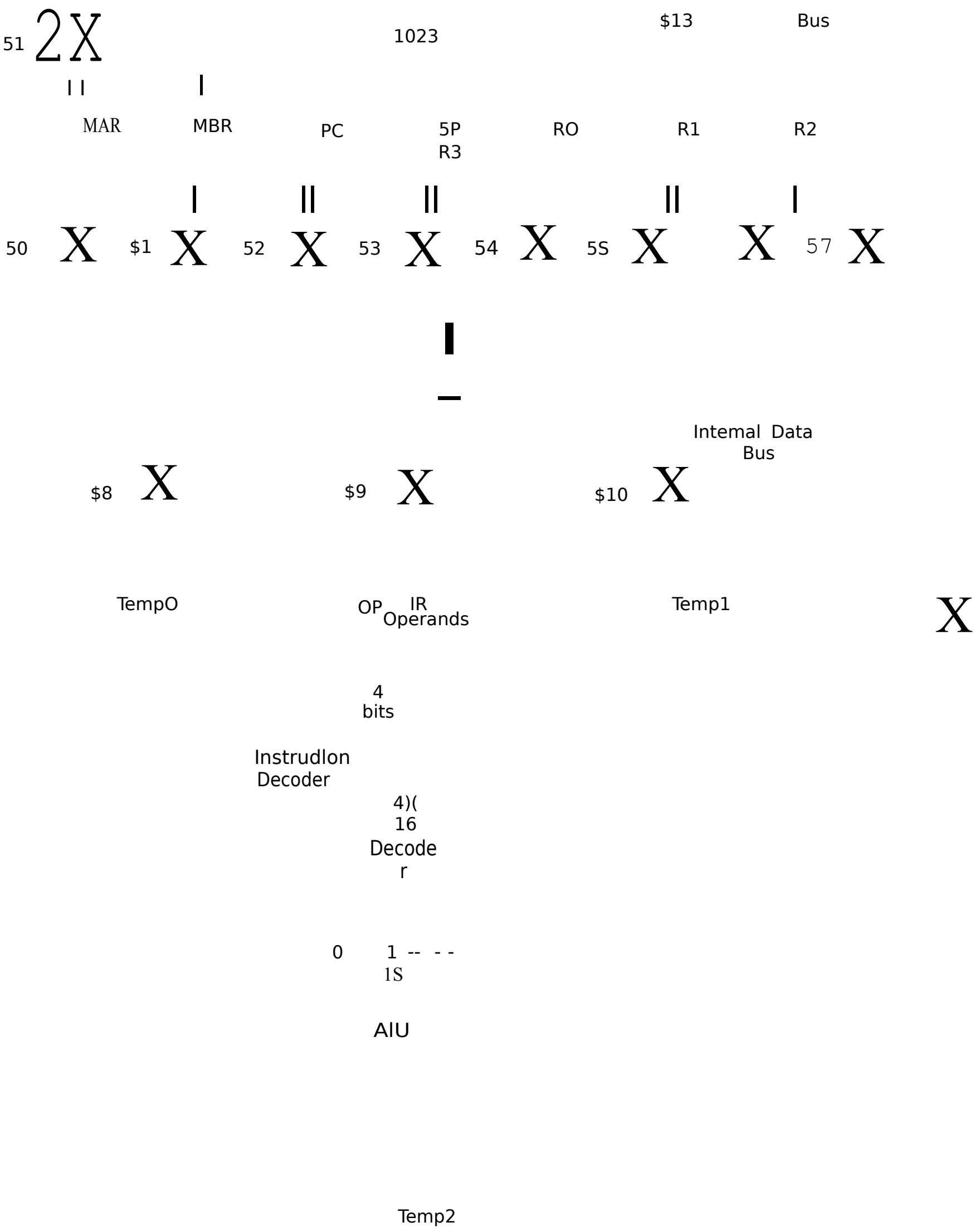


Fig. 2.4 Architecture of our hypothetical computer

accessed, the address bus will need to have 10 wires to carry 10 bits in the address. This is because 2^{10} = 1024. There is a CPU register called Memory Address Register (1\1AR) which has to contains the address of the memory location to be accessed. Therefore, in our example, MAR will consist of 10 bits. The address bus connects the MAR and a Memory Address Decoder.

This decoder bas L0 inputs from the address bits of the MAR. These address bits act as control signals to the decoder. This decoder has 1024 output lines, one going to each memory location separately, as shown in the figure. This is the reason, why this decoder is called 10 x 1024 decoder. Depending upon the binary value of the address, the corresponding memory location is activated, e.g, if the address is 00000o 1010, then the location number 10 (in decimal) will be activated. There is a read/write control Signal shown for the memory. This signal decides the direction of the data movement, i.e. from the main memory to the CPU registers (as used in the load operation) or from the CPU registers to the main memory (as used in the store operation).

Therefore, to move the data from memory location 11 (decimal) to register RI, the following steps will need to be taken:

(i) MAR is loaded with 00000o I0 11 (11 in decimal).

(ii) A "Read" signal is given to memory (equivalent to 'load'). (iii) The data is loaded into MBR by manipulating switch S13.

(iv) The data is moved from MBR to RI by manipulating S I and *S5*.

This will clarify how the system works together. Each of these small steps is called microinstruction. The control bus carries all the control signals.

2.6.2 CPtJ Registers

Figure 2.4 Shows various CPU registers as given below:

- Memory Address Register (MAR)
- Memory Buffer Register (MBR)
- Program Counter (PC)
- Instruction Register (IR)
- Stack Pointer (SP)
- General Purpose Registers RO, R 1..... , Rn
- Temporary Registers TEMPO, TEMP I, TEMP2

Of these, only the general purpose registers RO to RJI are accessible to the assembly language programmer, and are therefore, visible. All others are not accessible and they are hidden from the programmers. They are only internally used during the execution of instructions.

We will now study the functions of these various registers.

Memory Address Register (MAR) contains the address of the memory location which is activated and accessed for both the LOAD and STORE instructions, as discussed earlier.

Memory Buffer Register (l\-IBR) stores the data temporarily. before it is transferred to/front the desired memory location.

Program Counter (PC) contains an address of the "next" instruction to be executed. Therefore, when the program starts executing, the PC contains the address of the first executable instruction. This address, as calculated by the compiler, is stored in the header of the executable file created by the compiler itself at the tune of compilation. When the program execution begins. this header of the executable file is consulted and the PC is loaded with that value.

This is the reason why the computer begins at the first executable instruction in the program (e.g. the first instruction is tbe JUMP to PROCEDlJRE DIVISION in COBOL). When any instrucrion is executed. the PC is incremented by I so that it points to the next instruction. In a machine where the instruction length is more than 1 (word), it is incremented by that number. This cycle is broken only by the "JUMP" instruction. The jump instruction specifies the address where the control should be transferred. At ibis juncture, this address is moved into the PC. This is how the jump instruction is executed.

Stack Pointer (SP): Most computers make lise of stacks. A stack is a data structure which has a Last In First Out (LIFO) property. It works as a stack of books. When you add a new book to the stack, you add it on the top of the stack. When you remove it, you normally remove the one that was added the latest (LIFO). A number of memory words are organized in a similar fashion which can therefore be termed as a stack. At any point, some of these words contain some stored data, while others are empty. The stack ., pointer provides an address of the first free entry where a new element can be copied from a CPU register by a "l>USH" instruction. After this instruction, the SP is incremented to reflect the change. TheSP then points to the new free entry. Another instruction "POI'" does the reverse of ·'PUSH". It copies into tbe CPU register, all element from the stack whose address is given by the SP and then decrements tbe SP. The stack is a very useful data structure to implement nested subroutines. When you branch to a series of subroutines one after the other in a nested fashion, the rerum addresses can be "PUSHed" onto the stack. On returning, they can be "POPed" in the LIFO order. Which is very necessary if the nested subroutines have to be executed properly.

General Purpose Registers: Registers, RO through R7 can be addressed in various assembly instructions. RO to R7 are 8 such registers as shown in Fig, 2.4. In actual practice, there may be 1. 4, 8. 16 or 32 such registers. When there is only one such register, it is popularly called an Acculnulator in the literature of microprocessors.

Instrucnon Register (IR) holds an instruction before it is decoded and executed. Therefore, to execute an instruction, the instruction whose address is given by "PC" has to be "fetched" from the memory into the lR. This constitutes the 'fetch cycle'. This is followed by die 'execute cycle', in which the instruction is decoded and executed.

Temporary Registers (TEMPO, TEMPI, TEMP2): These are the registers within the ALU to hold the data temporarily, TEl\.fPO and TEMPI are input registers into ALU, and TEMP2 is all output register from the ALU containing the results from any operation. Therefore, when you want to add any rwo numbers, they have to be brought to TEMPO and TEMPI and the "ADD" circuit in the ALU has to be activated. After tbe addition, TEMP2 will contain the result. It now can be .moved to any other CPU register or a memory location as required, through the internal data bus. What we have described is only tbe architecture of hypothetical cornpurer. In reality, an architecture could be different. though, in principle, it bas to be similar.

2.6.3 Tbe ALU

The ALU can be logically seen as consistiug of different circuits-one for each instruction as shown in Fig. 2.5. For instance. if a computer bas 16 different instructions such as }\DD, SUB, MVL, DIV, LDA, ST A, etc. then we can imagine a different circuit for each of these instructions. How does a specific circuit gel activated'? The answer lies in a decoder circuit, which we will study next.

We must remember that this is only a logical view. In practice, the subrracrion circuit is derived from only a slight modification of the adder circuit instead of having separate circuit for each instruction. In

When any instruction arrives *in* the IR, the opcode bits from the IR are separated out and they form the control signals to the 4 x 16 instruction decoder. This means that it has 4 input control signal lines and 16 outputlines-one for each opcode, as shown in Figs 2.4 and 2.5. Depending upon the opcode, the appropriate circuit is activated and the operation takes place using the registers TEMPO. TEMPI and TEMP2.

2.6.6 The Machine Cycle

The machine cycle means the steps in which the instruction is executed. It consists of two subcycles: Fetch Cycle, and Execute Cycle. We will now discuss these with reference to Fig. 2.4.

(i) Fetch Cycle In the fetch cycle, the instruction is brought into the m from the memory. This is the instruction whose address is in the PC. Therefore, the fetch cycle consists of the following steps:

(a) Move PC to MAR by manipulating (opening/closing) switches 52 and 50.
(b) Give the 'Read' signal.
(c) Manipulate switch 513 to deposit the instruction in MBR.
(d) Move MBR to IR by manipulating switches SI and S9.
(e) Increment PC by I.

(ti) Execute Cycle The execute cycle consists of the following steps:

(a) Load TE~lPO and TEMPI with the appropriate values as necessary. This is achieved by adjusting the appropriate switches.
(b) Activate the appropriate ALU circuit by using the opcode bits in the instruction as the control signals to the instruction decoder in the ALU. (After this, TEMP2 will contain the result of the operation.)
(c) Move tbe result in TEMP2 to the destination, by again manipulating the appropriate switches.

Remember that TEMPO, TEMPI and TEMP2 are internal, hidden registers not accessible to the assembly programmer. Internally. they must be used appropriately to execute any instruction by the hardware as a pan of the machine execution cycle itself.

In this fashion, instructions are executed one after the other. An instruction is indivisible. It cannot be executed only to a certain extent. It is executed completely or not at all. This continuous execution of successive instructions is broken only by the occurrence of an interrupt. The hardware bas a register which is used for interrupt processing. If an interrupt takes place due to the hardware malfunction or due to the completion of an 110 operation or on account of a certain time slice getting over (i.e. timer interrupt), certain bits in that register are set on.

Interrupt suggests an urgent action and therefore, the currently running program has to be put aside and resumed only.after the interrupt is processed or handled. However, the current instruction cannot be left half complete. This is why, only after each instruction is over, the hardware itself checks for the interrupt register to ascertain whether an interrupt has taken place or not, and if no interrupt has taken place, execute the next instruction (the PC is already pointing to the next instruction by step (e) of the fetch cycle if you recall I). We will study interrupts more closely later.

Let us now take some examples to cement our ideas.

2.6.7 Some Examples

Direct Addressing

Let us say that we have an instruction "ADD RO, *TOT',* as we have seen in Sec. 2.5.2. Let this instruction reside at location (word) number 700, which is 1010111100 in binary. (We have assumed a 10-bit address in our example.) We have assumed that the variable TOT is defined at location or word 500 which is 0111110100 in binary. We have also assumed the coding scheme that ADD = 0010 and the register bit in the instruction for RO = O. We have used a direct addressing mode. Therefore, the addressing mode bit in the instruction also is =0. Therefore, the binary machine instruction will be
0010000111110100. This is what. will be produced by the assembler, Let this instruction be already loaded at the address 700 (= 10 10111100). Let us also assume that the PC has the same address (=1010111100).

This instruction is executed as follows (refer to Fig. 2.4):

(i) Fetch Cycle

(a) The PC is moved to t.lAR by manipulating Switches S2 and SO. MAR also now contains 700 (=1010111100).
(b) ,\ 'Read' signal is given.
(c) The word at address 700 is selected and its contents are deposited into MBR by adjusting Switch *S13*
.
(d) MBR is copied into lR by adjusting Switches S I and *S9*.
(e) The PC is incremented by I. This can be done by moving the PC to TEMPO, tbe number 1 (i.e., 00000001 in binary) to TEMPI, carrying out the addition in the ALU and moving TEMP2 back to the PC. This will require a number of sub-steps and manipulations of several switches. Alternatively, a binary counter also could be used. We need not go into tbe details of how this is achieved.

(ii) Execute Cycle

(a) The instruction is decoded field-by-field. For instance, the register bit tells the hardware that RO is selected. Therefore, RO is moved to TEMPO by adjusting Switches S4 and *S8*.
(b) The addressing mode is interpreted as a direct address (Mode = 0).
(c) Therefore, the 10 bits of tile address portion in lR are moved to l'vIARby manipulating Switches S9 and SO. (Therefore, MAR now contains 500 in decimal. or 0111110100 in binary.)
(d) A "Read" signal is given.
(e) The data item at memory word location 500 is now put on the data bus.
(1) The data is deposited in MBR by adjusting Switch S 13.
(g) MBR is now copied into TEMP J by adjusting Switches S 1 and S 1O.
(h) The 'opcode' bits in the IR are used as control signals to the instruction decoder. For' ADD' operation, these bits are 0010. The appropriate output signal is generated from the instruction decoder and the "ADD" circuit is activated.
(i) The addition now takes place. TEMP2 now contains die result.
(j) TEMP2 now is copied 10 ROby adjusting Switches S I i and *S4*.
This completes the execution of the instruction.

A question naturally arises in our mind. \Vhy do we st.orethe return address in a stack? Why do we not store it anywhere else, in any other memory location? The answer is: any other memory location also would have been acceptable if there was no nesting of subroutines allowed. In such a case, you could store just one return address and then retrieve it to return to the main routine. The problem arises when a subroutine calls another subroutine, and so on. in such a case, the return addresses have to be stored in a LIFO fashion, if the processing has to take place properly. That is the reason a stack is used, because LIFO is the basic property of a stack,

Other Instructions

The real computers support a number of other instructions, such as AND, OR, NOR, XOR, etc. or COMPARE, Conditional jumps (Jump if equal, etc.), Detailed discussions of these are beyond the scope of the current text, though it should now be fairly easy to imagine how they are executed.

Each of these sub- steps *a, b,* c etc. that comprise an instruction is called a microinstruction. In order to execute one machine instruction, these microinstructions are executed one after the other automatically by the hardware itself. This implementation is relatively faster but it is also more complex and less flexible. Tbis is the latest trend, because the increased complexity due to hardwiring is greatly reduced by simplifying the instructions set itself. The RISC computers have only a limited set of simple instructions. Therefore, the name: "Reduced Instructions Set Computer (RIsqn.

An alternative to this followed in the majority of the non-RISC microprogrammed machines all these days was to have a Read Only Memory (ROM) where a small program was etched for each assembly or machine instruction as studied earlier in steps (a), (b), etc. Eacb machine instruction gave rise to multiple microinstructions. Microinstructions were like any other instruction having their own formats and lengths consisting of several bits, where each bit typically controlled a switch. For instance, a microinstruction 01001 could mean that Switch S0 is to be closed (0), Sl is to be opened (=1), S2 and S3 are to be closed (both 0) and finally Switch S4 is to be opened (=1). When certain switches are manipulated, we know that data transfer between various registers/memory can take place as we have seen earlier. Therefore, a specific substep can be executed by a specific microinstruction and microprogram consisting of several microinstructions would, in effect, execute one machine instruction. This is wby we said that there is a microprogram for eacb machine instruction. This is a simplistic view, but one which is not grossly incorrect.

Thus, in a microprogrammed computer, one can imagine microprograms-one for each machine instruction permanently residing in the ROM. Depending upon the opcode bits, the appropriate microprogram is activated. Thereafter, appropriate control signals are given and appropriate switches are adjusted to execute a series of actions in the desired sequence.

At any time, if there is an interrupt and the current program has to be suspended or put aside, the Context of a Program has to be saved. What is this context? As we know from the hierarcby of languages, an instruction in a 4GL is equivalent to many instructions in ıı 3GL. A 3GL instruction is equivalent to many instructions in a 2GL (assembly or IGL machine language). The context is essentially the values of all the registers plus various pointers to memory locations and files used by that program. When the interrupted program is to be restarted, all these register values have to be reloaded from the Register

Save Area, so that the program can continue as if nothing had happened. The registers have to be saved. because, the new program (Interrupt Service Routine or ISR, in this case) will be using all these registers internally for the execution of its own instructions. Imagine the effect on the interrupted program, if tbe PC is not saved for instance!

~N~iR6PrsFoO"'i2$gS A 41 (Ci .... ·.. ,.,. 21£12& ,p sa ~".""'!9"""'p,-

'"~'"""'|'.",,,,~,,- ....

~.,__..,~::t 't',s,,1' ',n', .h"j"-",,' ·.,_lke.=, .~ «.0..... · '" .. _ '}',o"., ▬ , ·

2.8.1 The Need for Interrupts

The concept of multiprogramming and multiuser Operating Systems is based on the hardware support of interrupts. What is this interrupt? Interrupt is essentially the ability of the hardware to stop the currently running program and turn the system's attention to something else. When you are reading a book line-by- line, and the doorbell rings, what happens? You normally adjust the book's marker, put the book down and go to open the door. In this case, your current program of 'reading a book' is interrupted and a new program or procedure of 'opening a door' starts executing. You open the door and attend to the person (may be a newspaperman or a milkman). This is similar to the Interrupt Service Routine (ISR) in computer jargon. After you finish your IS,R. you resume reading the book.

Imagine that you have opened the door, and are discussing something with tbe person at the door (i.e. you are in the midst of the ISR). At this juncture, if the phone bell rings, you would normally leave the conversation to attend to the phone with due apologies. This is similar to having interrupts within the interrupts or nested interrupts. In the case above, we have assumed that attending a phone call is more important, and therefore, of higher priority than completing the conversation with the person at the door which, in tum, is of higher priority than reading a book. In the same way, in the computer hardware. there can be multiple interrupts and one can set priorities to them so that should they occur at the same time, the computer should know which one to lake up first.

In our lives. we can disable interrupts at certain occasions. A manager busy in a meeting tells the secretary to take all the calls and messages so that the meeting is nOI interrupted. In computer systems too, you can do this. We call this masking a certain interrupt. How does the computer hardware do all th·ıs?.

2.8.2 Computer Hardware for Interrupts

We have seen how a computer program works without the scheme of the interrupts. The PC points to an instruction which is then fetched into the IR and executed, whereupon the PC is incremented to point to the next instruction. When you have a scheme of interrupts, the following happens:

(i) The hardware contains some lines or wires carrying different signals to denote the presence and type of an interrupt. In the simplest case, if the system allows for only one type of interrupt, only one line can indicate the presence of the interrupt (e.g. signal high (= t) indicates interrupts, and signal 0 means no interrupt).

(ii) You cannot predict the exact time when the interrupt will occur. This is obvious. If you could exactly plan their timings. you would perhaps call them by a different name and tackle them differently .

(iii) The hardware is designed in such a way that the current instruction cannot be interrupted in the middle. Only after the current instruction at the machine level is completed, the hardware can attend to the interrupt This is achieved by simply checking by the hardware itself whether an

interrupt has occurred or not (i.e, whether the interrupt line is high or not), only at the *end* of each machine instruction and before the start of the next one. But [here is a problem here. Let us assume that the interrupt occurs halfway during the execution of an instruction. The interrupt line will indicate a high signal. But for how long? By the time the machine instruction is executed. the signal may go low. thereby, causing the interrupt to be lost. Therefore. there is a need to 'store' the interrupt somewhere at least for a short duration by setting some bit in some register on!

(iv) It follows from (ii) and (iii) above that at least for a short duration, the interrupt has to be stored in some register. If there is only one type of interrupt, this register could consist of just bit. Otherwise, it will have to be longer. This register is called Interrupt Pending Register (IPR).

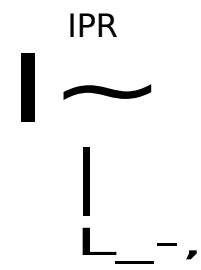
(v) The system normally has a provision to ignore an interrupt for a certain duration. This is called 'disabling' or 'masking' the interrupt. This is achieved by another register called Interrupt Mask Register (GIR). This is of the same length as IPR. In our case, it will be of only I bit, because we have assumed only one type of interrupt.

The IMR is set to 0, if we want to mask the interrupt, It is set to I, if we want to enable it. The fMR and IPR are connected to the AND circuit as inputs, and the output is collected in another register called Interrupt Register. This is shown in Fig. 2.6. Finally. this interrupt register is checked to identify the presence of the interrupt.

Obviously, if IMR = 0, the Interrupt Register will remain 0 even if IPR takes on a value of eitherOor 1. This is how 'masking' is achieved. If ThIR = 1. then the interrupt register will be the same as IPR. Therefore. the interrupt register will be 1 if IPR = I. This is the way interrupts arc enabled. To set/reset IMR are privileged instructions. These are generally used only by the Operating System.

(vi) After completion of every instruction. before fetching the next one, the hardware itself checks the interrupt register and continues with the fetch operation, if this register is 0.

(vii) If this interrupt register = 1, the hardware jumps to a preknown address where the Interrupt Service Routine (ISR) (which is a part of the Device Driver portion of the Operating System) is kept. This is depicted in Fig. 2.7.



Interrupt RegiS1!IJ

-'-'" Fig. 2.6 Interrupt masking and various registers

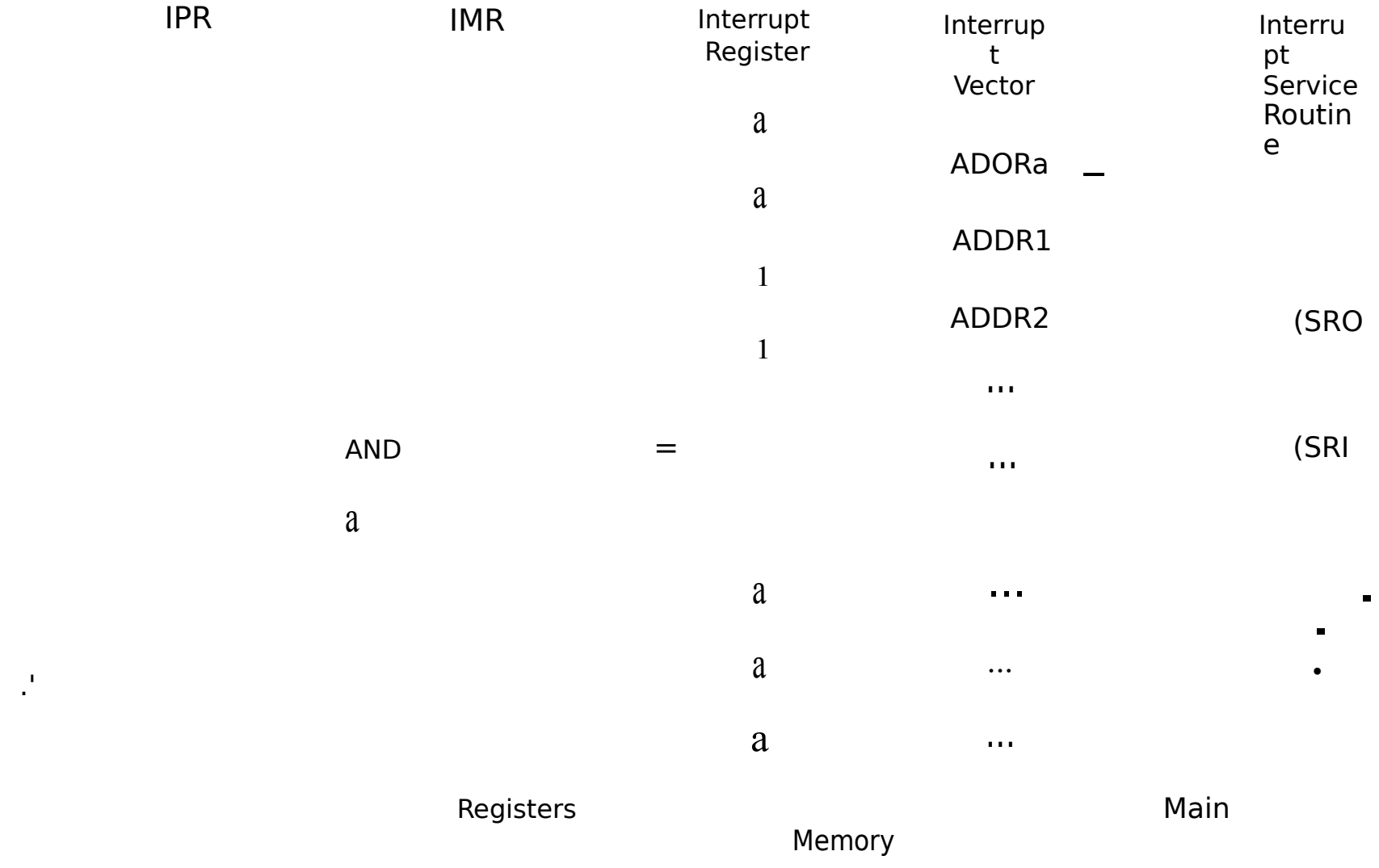
<=-.? Fig. 2.7 Interrupt processing

Therefore, as an example,

IPR = JOIIIOO} and
IMR =01110010 then
Interrupt Register = 00 II 0000

The bits in the Interrupt Register are now put through an OR gate as shown in Fig. 2.8. The output (I bit) indicates whether there is an interrupt of any type at all or not.

(b) There are as many ISR routines stored in the memory as there are interrupts that need to be serviced. Each ISR has a memory address. A list of all these addresses is called **Interrupt** vector. This is shown in Fig. 2.9.



.=., Fig. 2.9 Interrupt vector and ISRs

If there is an interrupt indicated by any non-zero bit in the interrupt register as shown in Figs. 2.8 and 2.9, then the ISR routine within the Operating System corresponding to that interrupt has to be executed. Qne way to achieve this is hy the Operating System actually finding out which interrupt has occurred through the software routine. (A kind of table search to know which bit within the Interrupt Register is 1.)

Having found it, the Operating System can pick up the corresponding address of the ISR and branch to it. This is obviously very slow. Another method. called vectored **interrupt** avoids this 'software table search'. It can directly branch to the correct ISR, because the hardware itself is made to not only signal the presence of the interrupt, but also made to send the actual address of the ISR by hardware lines. This method is obviously more efficient though more expensive. A detailed discussion on this is beyond the scope of the current text and it is also not necessary.

If there are multiple interrupts, then there will be more than 1 bit set to I in the Interrupt Register, and then the Operating System will have t\) service them one by one in a certain sequence. The sequence can be predetermined by setting priorities 'to different interrupts. In fact, during the execution of an ISR, if another interrupt occurs, it could be serviced on a priority basis. All this is possible, but again a detailed discussion on this is unnecessary.

Note that a particular bit in IPR can be set by t\VOways. One is by hardware (Hardware Interrupts), which we have seen up to now. Another way is by executing a software instruction (Software Interrupt) to set or reset a particular bit in IPR. These are obviously privileged instructions and can be used only by the Operating System. Therefore, only the Operating System can set a particular bit in IPR to I to generate a Software Interrupt. Irrespective of how it is set, the hardware must check the IPR before fetching the next instruction. As we will learn later, if a process (i.e. running program) requests the Operating System to read a certain data record on its behalf, the Operating System has to keep the process waiting until the desired *1/0* is over. The Operating System has to 'block' that process. To do this, the Operating System has to cause a software interrupt first and then branch to the corresponding ISR and change the status of that process to "blocked".

After the process is blocked, the Operating System supplies the DMA controller with the relevant details of the data transfer and takes up another process for execution. The DMA completes the I/O and generates a hardware interrupt setting a specific bit in IPR to I. Remember that, at this time, a certain other process would be executing. On encountering this interrupt caused due to the 110 completion of the first process, the CPU is taken away from this second process too! When this interrupt is serviced, the ISR moves the currently running (second) process into the list of 'ready' processes and then finds our which device has caused this interrupt and the reason for the same (i.e. Data read is over). It then fi.nds out for which process this 'Read' operation was done. It then moves this process from blocked to the ready state so that it can now be scheduled.

There is another type of interrupt that devices can generate. This is the "Device Ready" interrupt. This is generated by an idle device ready to accept any data. This is generated regularly at fairly short time intervals. The ISR for that device generally checks if there is any data ready in the memory buffer for that device and which should be sent to that device. If there is, the device *110* is initiated. If there is oo waiting
data, the device 1/0 is put to sleep and the highest priority ready process is dispatched for execution.

Most of the Operating Systems implement the time sharing facility by using a timer clock, which generates a signal over a line after a specific rime interval which is a bigger slice than the intervals at which the device generates the interrupt for "Device Ready". This time slice for the timer can be externally programmed, because, the timer hardware has a register and machine/assembly instruction is available to set this register to a specific value. This is obviously a privileged instruction. After setting this register 10 a specific value, as the time progresses, the value in this register goes Oil decreasing. When this value becomes zero, a timer interrupt is automatically generated.

This timer signal can be used to set on yet another bit in IPR to indicate the interrupt. On recognizing this, the Operating System executes the ISR corresponding to the interrupt caused by this "time up". After saving the context of the then running process, the Operating System switches from that process to the next, after changing the earlier running process to the ready state.

Normally, when tbe Operating System is executing some routines (say, a high priority ISR), it does not want any interruption from any other especially lower priority, interrupts. This can be ensured by adjusting the IMR. The Operating System sets the IMR to the desired value first to disable certain types of interrupts, completes the routines and then enables them again before exiting. All the interrupts that come in during that period change the !PR, but do not take effect due to masking. This more or less completes the picture of computer architecture which is necessary for our understanding of the subject of Operating Systems. It should be noted that what we have described is a simple but nor too unrealistic view of a praetical computer.

True or False Questions

2.1 A 3GL (HLL) is hardwaje independent and 2GL (Assembly Language) is hardware dependent.
2.2 Condition code register is a part of CPU registers.
2.3 Programming in HLL is more cumbersome than programming in AL.
2.4 Assembly language is a structured programming language.
2.5 The assembly language program execution starts from address 0 unless it is specified by 'ORG' instruction.
2.6 Fetch, decode, and execute is the actual tlow of the instruction execution.
2.7 Processing speed of machine is directly proportional 10 the size of the data bus.
2.8 During execution program, counter contains the address of the current instruction to be executed.
2.9 The number of memory loeations that can be accessed depends on the size of control bus.
2.10 TSB or switches works as latches.
2.11 Unlike assembly language, there is one-to-one correspondence between HLL and machine language.

Multiple Choice Questions

2.1 In the memory hierarchy of operation system. is the fastest accessible memory .
(a) CPU register (b) disk (c) main (d) cache memory

2.1 Assembly Language code is converted into Machine Language code by _
(a) compiler (b) assembler (c) simulator (d) emulator

2.3 HLL code is converted into Machine Language by _
(a) assembler (b) simulator (c) compiler (d) emulator

2.4 stores the data temporarily before it is transferred to/from the desired memory location.
(a) CPU register (b) MAR (c) MBR (d) Temporary Register.

2.5 A stack is a data structure which works on property.
(a) LILa (b) LIFO (c) FIFO (d) FILa

2.6 To avoid the interrupt to be lost, register is used to store it for short duration.
(a) IMR (b) lPR (c) lR (d) Interrupt vector

2.7 Stack Pointer gets after pushing the content to the stack.
(a) incremented (b) decremented (c) remains constant.

2.8 is used for transferring data directly to memory without interfering with processor.
(a) CPU register (b) Memory Buffer register
(c) DMA Controller (d) Tri-state buffer

2.9 IMR (Interrupt Mask Register) is used
(a) to set the priorities of interrupt
(b) to disable the interrupt
(c) to enable the interrupt so that programs receive interrupts while execution
(d) to store the interrupt occurred

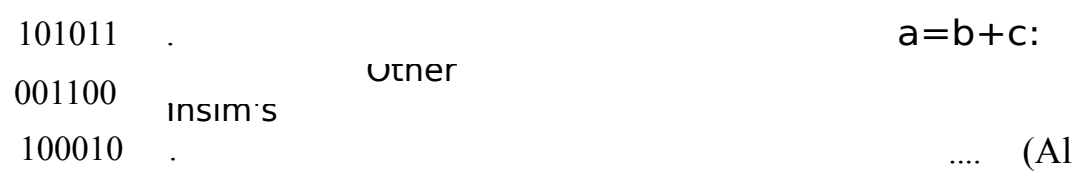
2.10 Instructions used to perform stack operations are _
(a) PUSH and POP (b) JUMP and RET (c) MOVE (d) LDA

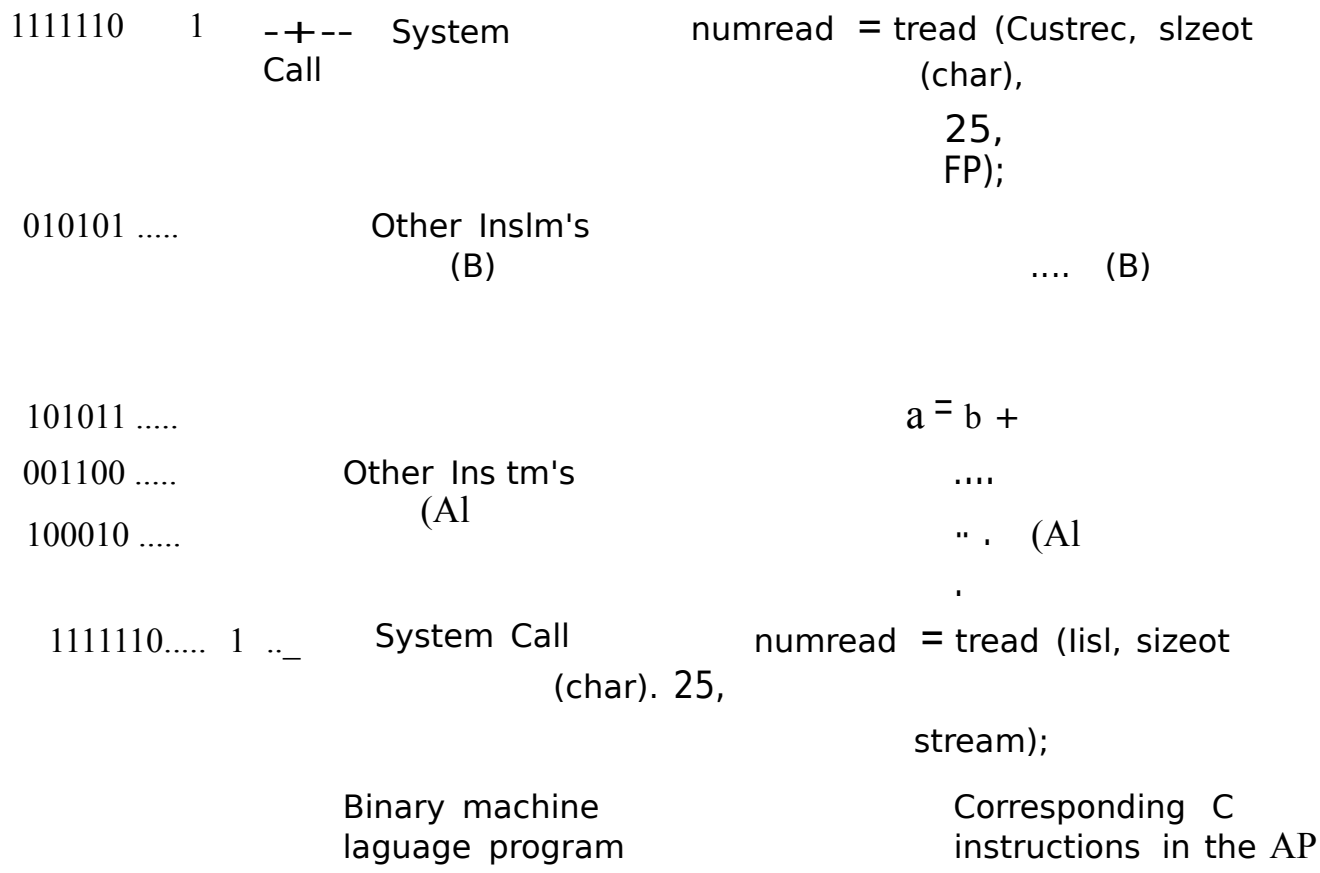
Test Questions

2.1 Why are MAR and MBR registers required? How are they connected to other memory locations?
2.2 What is a data bus? How does the width of the data bus affect the system performance?
2.3 What is an address bus? How many bits must it consist of?
2.4 What is the function of a decoder? How does it do it? .
2.5 What is the relationship amongst the High Level Language, Assembly Language, and Machine Language?
2.6 Discuss the exact functions of Program Counter, Instruction Register, Accumulator and other General Purpose Registers.
2.7 With an example, describe in detail the exact steps involved in the execution of instructions.
2.8 Why are interrupts needed?
2.9 Discuss the need for the Interrupt Pending Register (IPR).
2.10 What is meant by 'masking' an interrupt? How is it done? Why is it required ?
2.11 Describe in detail the procedure involved in the processing of interrupts.
2.12 What are vectored interrupts?

Cor FD section in the case of COBOL)" or a with "Write" instruction which is just the opposite. The Operating System is responsible for the address translation as well as the execution of this instruction.

The COBOL C compiler compiles all other instructions such as MOVEa = b; or COMPUTEa = b + c; (i.e. except instruction such as 110) into the machine instructions, as we have seen earlier, but when it encounters an JJ0 instruction as given above, it substitutes a call to the Operating System, known as a system call, or in ffiM terminology, Supervisory **call** (SVC). This is illustrated in Fig. 3.1.





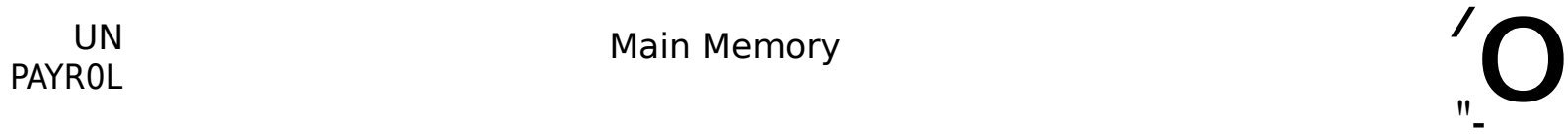
.....J' Fig. 3.1 Binary program and corresponding instructions in C

At the time of the execution of this program, after the instructions shown in Section A of Fig. 3.1 are executed, when this system call is encountered, there is no point in continuing the current program, because, without reading the customer-record, the AP cannot do ally other operation indicated by 'B' in Fig. 3.1. This is because, these operations may be dependent upon the data in that record which is yet to be read. If the program is allowed to continue to execute the instructions in Section B, it may compute something on the data in the last or the previous record, This is why, this program is put aside for the time being (in the Operating System jargon, the process is 'blocked', as we will learn). The Operating System takes over to read the required record on behalf of the AP. and schedules some other program at the same time.

The Operating System, however, will have to know from which file the record is to be read and where in memory it is. This information is supplied in the system call itself in the form of parameters, and passed on to the Operating System call for "Read".

Disk drives are controlled by another hardware unit called, 'disk controller'. This controller is, in fact, a tiny computer which has an instruction set of its own-basically to take care of the 110operations. The controller has its own memory which can vary from one byte to even one track, i.e. maJ1Ykilobytes.

This controller (or small computer) needs an instruction specifying the address of the sector(s) which is to be read and that of the memory locations where it is to be read. Once the Operating System supplies this instruction to the controller, the controller Callcarry out the data transfer OD its own, without the help


UN
PAYR0L
Main Memory
'O
"-

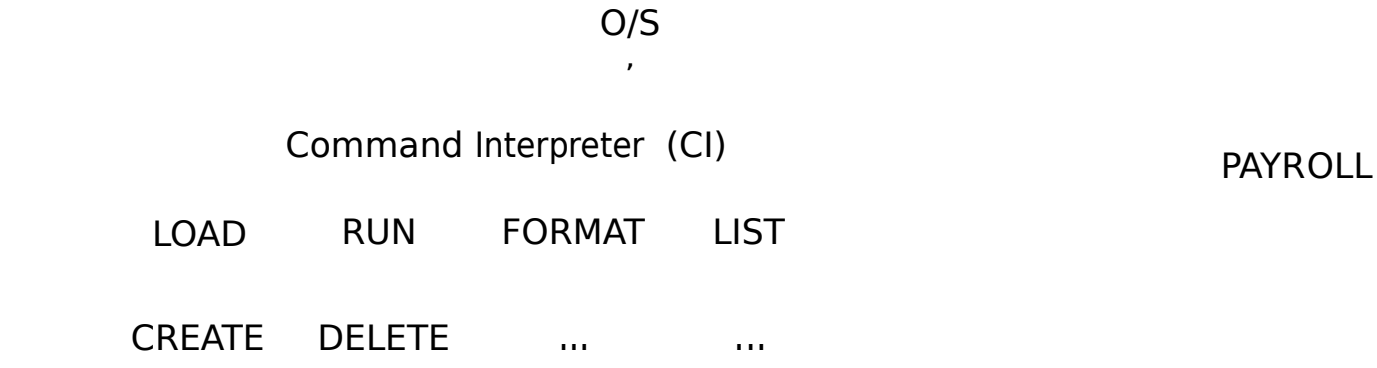


Fig. 3.13 Program loaded in memory

The picture is a little simplistic, but two points become clear from the following discussion:

- The system calls in 1M, PM and MM categories have to work in close cooperation with one another.
- Though the user's or Application Programmer's view of the Operating System is restricted to the CL, the commands are executed with the help of a variety of system calls internally (which is essentially the system programmer's view of the Operating System).

GRAPHICAL USER INTERFACE (GUI)

The latest trend today is to make the user's life simpler by providing an attractive and friendly Graphical User Interface (GUI) which provides him with various menus with colours, graphics and windows, as in Microsoft Windows. Therefore, the user does not have to remember tedious syntaxes of the Command Language, but can point at a chosen option by means of a mouse, However, this should not confuse us.

On the other hand, this layered approach also makes these Operating Systems somewhat less efficient compared to Monolithic Operating Systems, as all requests pass through multiple layers of software before they reach the hardware. For example, if a program needs to read data from a file, it will execute a system calL.The system-call interface layer will trap this call and it will call the fi1e system management layer with arguments like file name and the desired operation, The file system management layer in tum will call memory management layer passing a pointer to the memory location corresponding to the file.

The memory management layer will use the services

of the processor scheduler layer that will finally allocate the processor to carry out the desired operation. As a result, the overheads incurred at each layer may make tbe overall operation slower.

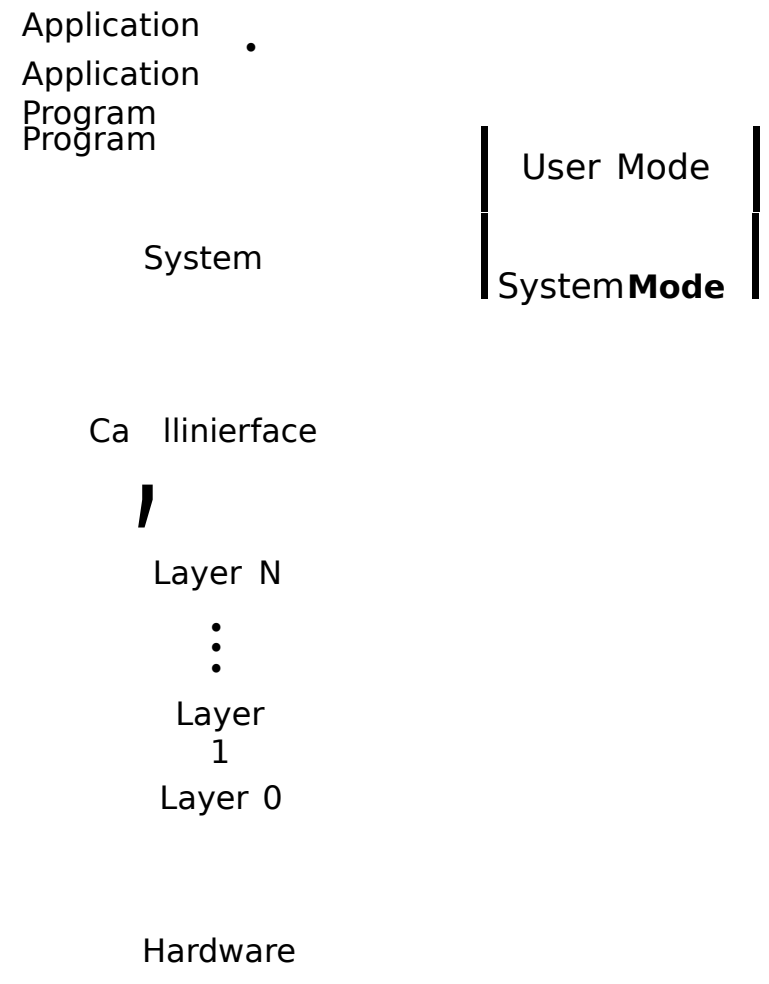
Additionally, at times it may not be possible to decide.;1f.l'Ilasthould go in a panicular layer. There may be overlapping functionalities, or layers that focus on too many or too little features. Thus it may happen that some of the layers do not do much work and others are heavily loaded thus creating performance bottlenecks.

Examples of Layered Operating Systems are VAX/VMS and UNIX.

3.7.3 MicrokemelOperating System

Layered Operating Systems are better organized and more systematic in terms of their design and operation. However, generally they have proven to be less efficient. Moreover, as the designers of the Operating Systems started adding more and more features, the kernel became 'too heavy' and unmanageable. These obstacles gave birth to a totally new approach for designing Operating Systems-the l\.licrokemel approach. Microkernel has become a real buzzword in the world of Operating Systems today with more and more Operating Systems subscribing to this approach.

The basic concept of a Microkernel Operating Syst.ems is very easy to understand. In this approach, the kernel provides only the most essential Operating System functions like process management, communication primitives and low-level memory management. System programs oruser level programs, implemented outside the kernel, provide the remaining Operating System services. These programs are known as *servers*. As a result, the size of the kernel reduces dramatically, making it a Microkemel. The application programs and various servers communicate with each other using messages that pass through the Microkernel, The Microkemel validates the messages, passes them between the various modules of the Operating System and permits access to the hardware. Figure 3.18 illustrates the working of a Microkernel Operating System. The Mach Operating System developed at the Carnegie Mellon University in the 1980s was the first Operating System built using the Microkerncl approach. Andrew Tanenbaum's Minix is another example of a Microkernel Operating System.



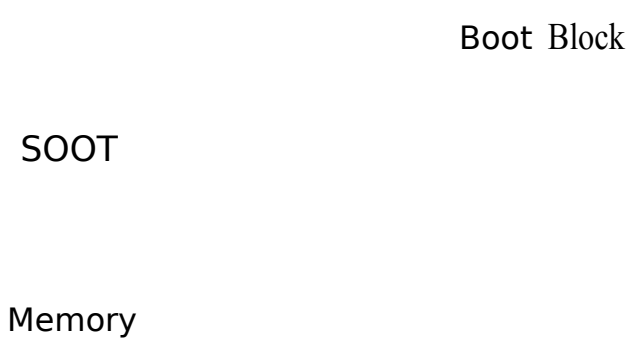
:;..,; Fig. 3.17 Layered Operating System

Operating System; because in order to execute, that program will need to be in the memory first, and then we will need to ask: Who brought that program P in the memory? How was the *1/0* possible without the Operating System already executing in the memory? It appears to be a chicken and egg problem.

In some computers. the part of the memory allocated to the Operating System is in ROM; and therefore, once brought (or etched) there. one need not do anything more. ROM is permanent; it retains its contents even when the power is lost. A ROM-based Operating System is always there. Therefore, in such cases. the problem of loading the Operating System in the memory is resolved.

However, the main memory consists of RAM in most of the computers. RAM is volatile. It loses its contents when the power is switched off. Therefore, each time the computer is switched 'on', the Operating System has to be loaded. Unfortunately. we cannot give a command of the type LOAD Operating System, because such an instruction would be a part of the CI which is a part of the Operating System which is still on the disk at that time. Unless it is loaded, jt cannot execute. Therefore, it begs the question again!

The loading of the Operating System is achieved by a special program called BOOT. Generally this program is stored in one (or two) sectors on the disk with a pre-determined address. This portion is normally called 'Boot Block' as shown in Fig. 3.21. The ROM normally contains a minimum program. When you tum the computer 'on', the control is transferred to this program automatically by the hardware itself. This program in ROM loads the BOOT program in pre-determined memory locations. The beauty is to keep the BOOT program as small as possible, so that the hardware can manage to load it easily and in a very few instructions. This BOOT program in tum contains instructions to read the rest of the Operating System into the memory. This is depicted in Figs 3.21 and 3.22.



---" Fig. 3.21 The *H/W* loads the BOOT routine automatically

The mechanism gives an impression of pulling oneself up. Therefore, the nomenclature bootstrappiDg or its short-form booting.

What will happen if we can somehow tamper with the BOOT sector where the BOOT program is stored on the disk? Either the Operating System will not be loaded at all or loaded wrongly, producing wrong and unpredictable results, as in the case of Computer Vims.

3.6 is used to keep a track of whether any command is issued on Cl or not. (a) DMA controller (b) System Call
(c) Interrupt Service Routine (d) Watchdog Program

3.7 is volatile memory and is nonvolatile memory.
(a) ROM. RAM (b) PROM, ROM (c) RAM, ROM (d) EPROM, RA,M

3.8 The control is transferred to programs in after turning 'oo' the computer system.
(a) RAM (b) ROM (c) EPROM Cd) DRAM

3.9 Syslem/360 Operating System can be considered as
(a) Online Computing OS (b) Timesharing OS
(c) Batch Oriented OS (d) Real time OS

3.10 The innermost layer of the Operating System close 10 the hardware and which control it is the

(a) Application Program
(b) Kernel
(c) Graphical User Interface
(d) Hardware Layer

Test Qu,estiODS

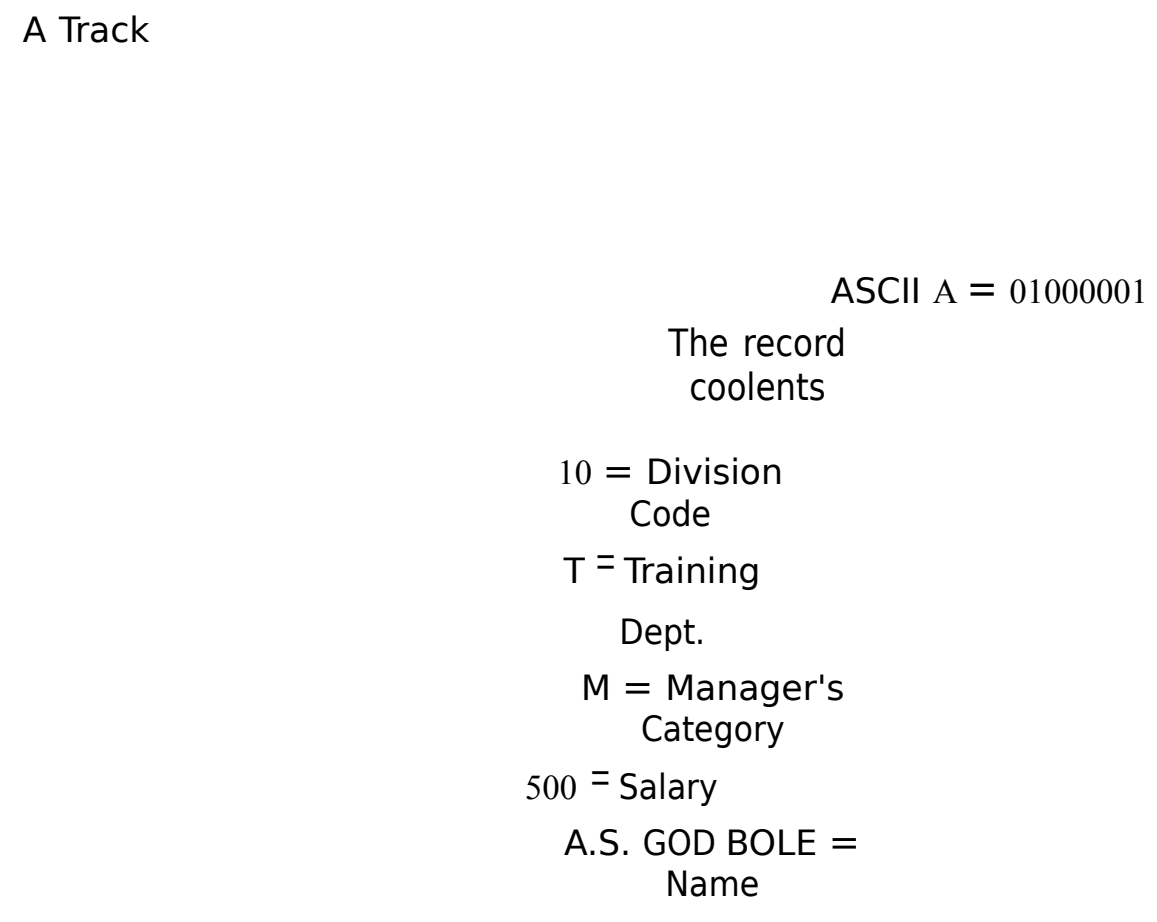
3.1 Why is a user program not allowed to carry out a direct ReadlWrite operations from/to a disk sector?
3.2 What is meant by a System Call? How is it used? How does an Application Program use these calls during execution? How is all this related to the compilation process?
3.3 Which are the three major areas in which the Operating System divides its services? Give examples.
3.4 How does a program become portable? What is meant by object code portability? What is source code portability?
3.5 Describe what is meant by 'User's' view of the Operating System,
3.6 Describe the exact steps involved in running a program sitting at a terminal.
3.7 What is Kernel? Describe briefly the approaches of designing Operating Systems.
3.8 Describe the steps involved in "Booting".
3.9 What are the functions of an Operating System?
3.10 What is a system call? How is it different from a subroutine or a subprogram?
3.11 What is the need for an Operating System?
3.12 Explain the following:

- Batch processing
- Multiprogramming
- Time sharing
- Real time system
- Online system
- Multiprocessing

4.1.1 Disk Basics

A disk constitutes a very important lIO medium that the Operating System has to deal with very frequently. Therefore, it is necessary to learn how the disk functions. The operating principle of a floppy disk is similar to that of a hard disk. Simplistically, in fact, a hard disk can be considered as being made of multiple floppy disks put one above the other. We will study the floppy disks in the subsequent sections but the discussion is equally applicable to the hard disks as well.

Disks are like long play music records except that the recording is done in concentric circles and not spirally. A floppy disk is made up of a round piece of plastic material, coated with a magnetized recording material. The surface of a floppy disk is made of concentric circles called tracks. Data is recorded on these tracks in a bit serial fashion. A track contains magnetized particles of metal, each having a north and a south pole. The direction of this polarity decides the state of the particle. It can have only two directions. Therefore, each such particle acts as a binary switch taking values of 0 or 1, and 8 such switches can record a character in accordance with the coding methods (ASCDJEBCDIC). In this fashion, a logical record which consists of several fields (data items), each consisting of several characters is stored on the floppy disk. This is depicted in Fig. 4.1.



....., Fig. 4.1 Data recording on the disk

A disk can be considered to be consisting of several surfaces, each of which consisting of a number of

tracks as shown in the Fig. 4.2. The tracks are normally numbered from 0 as the outermost track, with the

Controller's Memory — Electronics to collect and shift bits — Data in

— Dataout

Current Error/Status Registers
Other Info. Track No.

Area to store instructions

Start Molor

..... Check

Status .....

'.....

'.....

Seek | Surface # | Track l# | Sector l# | Targel

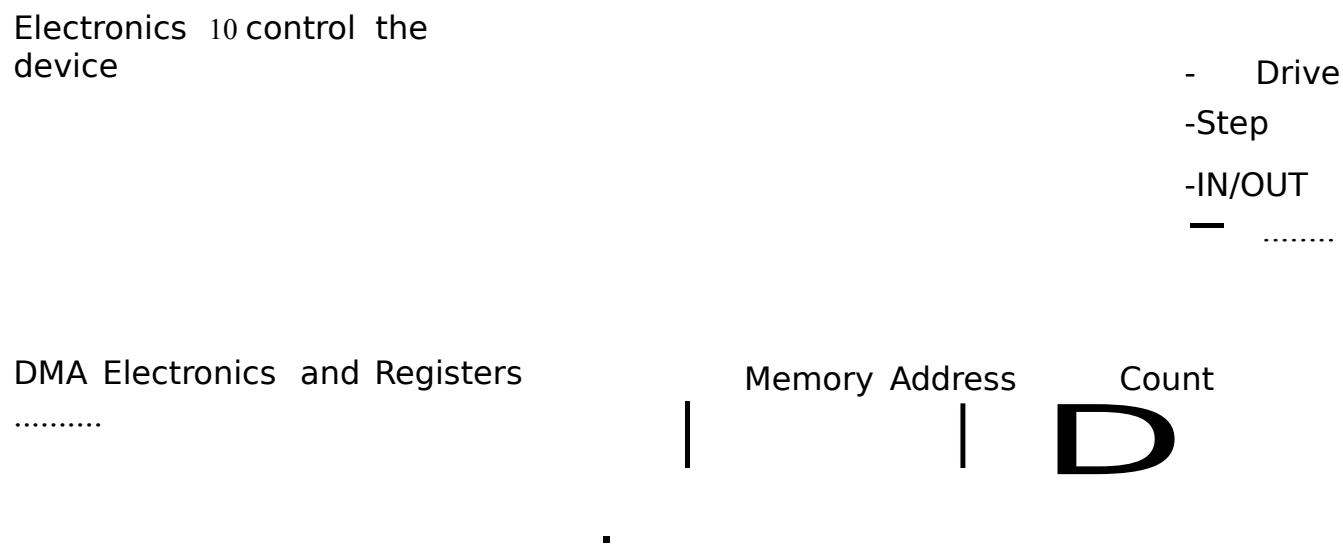
'.....

Set up DMA registers.

Stop Motor.

,

Electronics/Registers (MAR, IR, PC .....) to execute
controller's ins1rUctions.



"""", Fig. 4.10 The schematic of a disk controller

2. *Electronics to Collect and Shift Bits*

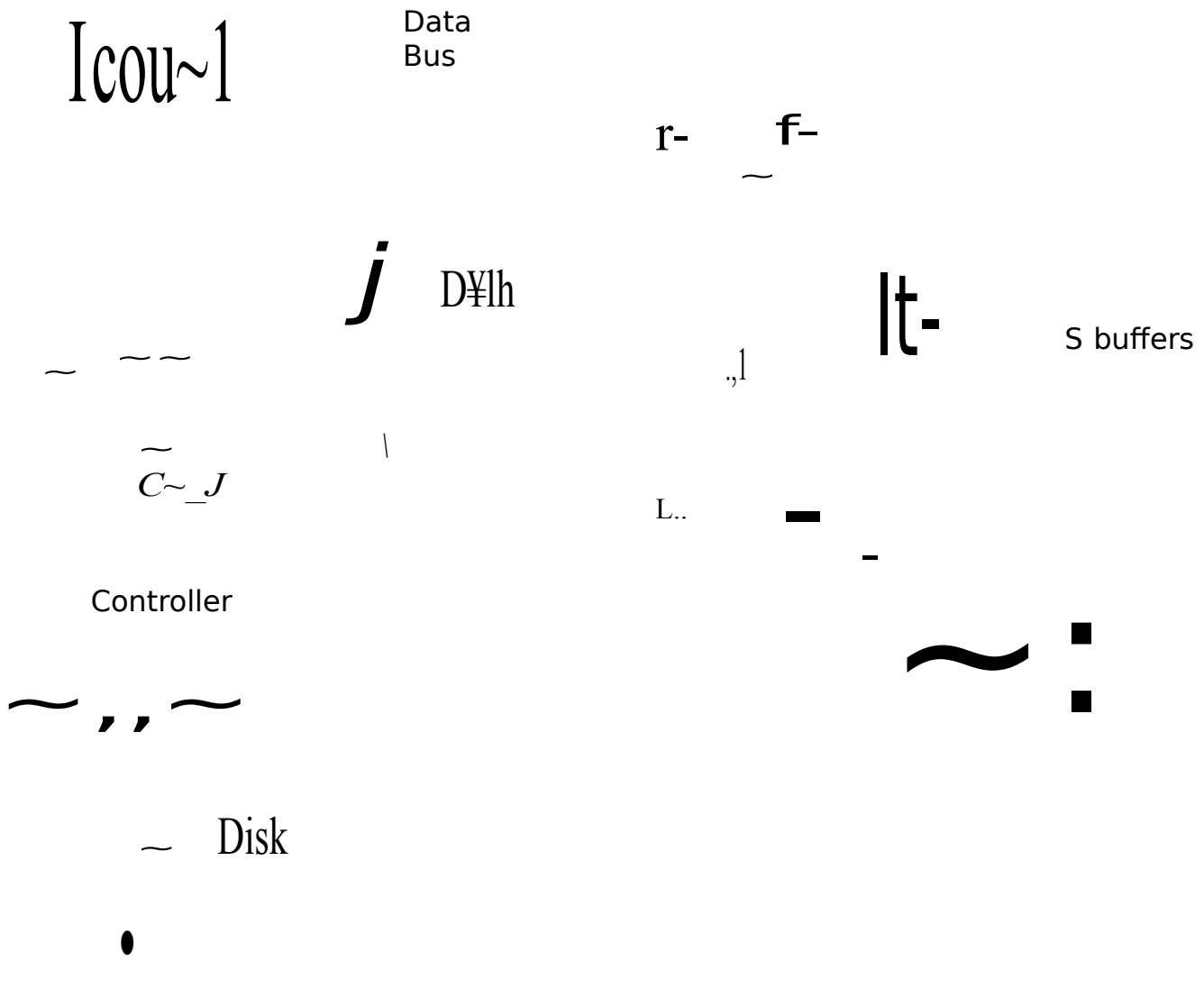
Tbe Data In and Data Out wires are responsible for the data transfer between the disk drive and the controller as shown in Fig. 4.8 and also in Fig. 4.10. As the data arrives in a bit serial fashion, the electronics in the controller collects it and shifts it to make room for the next bit. After shifting eight bits, it also accepts the parity bit, if maintained, depending on the scheme, The bardware also checks tbe parity bit for correctness. The received bytes are then stored in the controller's memory as discussed in I.

When the DMA is transferring the data, the main CPU is free to do the work for some other process. This forms the basis of multiprogramming. But there is a hitch in this scheme. Exactly at the time when the DMA is using the data bus, the CPU cannot execute any instruction such as LOAD or STORE which has to use the data bus. In fact, if the DMA and the CPU both request the data bus at the same time, it is the DMA which is given the priority. In this case, the DMA 'steals' the cycle from the CPU, keeping it waiting. This is why it is called cycle stealing. The CPU, however, can continue to execute the instructions which involve various registers and the ALU while the DMA is going on.

An altemative to the scheme of DMA is called **programmed** 110, where each word from the controller's memory buffer is transferred to the CPU register (MBR or ACC) of the main computer first and then to the target memory word by a 'store' instruction. The software program then decrements the count and loops back until the count becomes 0 to ensure that the desired number of words are transferred. This is a cheaper solution but then it has two major disadvantages. One is that it is very slow. Another is that it ties up the CPU unnecessarily, therefore, not rendering itself as a suitable method for multiprogramming, multi-user systems, We will presume the use of the DMA in OUT discussion.

Figure 4.IO{c) shows both the possible methods of 110. From the data buffer in the controller, the data can be transferred through the Operating System buffers or directly into the memory of the application program.

System Memory



:.. -;; Fig. 4.10(c) DMA and Programmed I/O

(SNs), and the SNs are converted into the physical addresses (cylinder, surface, sector) as discussed earlier. These are then used by Device Management for actually reading those sectors by setting the 'seek' instruction in the memory of the controller accordingly. The controller, in tum, sends the appropriate signals to the disk drives to move the RIW arms and to read the data into the controller's buffer and later to transfer it to the main memory through the DMA. Here too, though other schemes are possible, for logical clarity, we will assume that the data is read in the Operating System buffer first and then transferred to the APs memory (e.g. FD area) by DMA.

Some Operating Systems follow the technique of interleaving. This is illustrated in Fig. 4.13. After starting from Sector 0, you skip two sectors and then number the sector as J, then again skip two sectors and caU the next sector as 2, and so on. We caU this interleaving with factor = 3. Gen- erally, this factor is programmable, i.e. adjustable. This helps in reducing the rotational delay.

The idea here is simple. While processing a file sequentially. after reading a block, the program requesting it will take some time to process it before wanting to read the next one. In the non-interleaving scheme. the next block will have gone past the RIW heads due to the rotation by

4

5

that time. thereby forcing the controller to wait until the next revolution for the next block. In the interleaving scheme, there is greater probability of saving this revolution if the timings are appropriate.

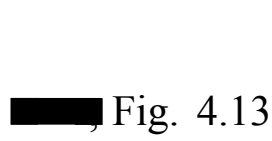


Fig. 4.13 Interleaving with factor = 3

4.2.3 File Support Levels

The Operating System is responsible for the translation from logical to physical level for an Application Program. In earlier days, this was not so. In those days, an application programmer had to specify the actual disk address (cylinder. surface, sector) on the Operating System to access a file. If he wanted to access a specific customer record, be had to write routines to keep track of where that record resided and then he had to specify this address. Hence the application programmer had a lot of work to do, and the scheme had a lot of problems in terms of security, privacy and complexity.

Ultimately somebody had to translate from the logical to the physical level. The point was. who should do it? Should the Application Program do it or should the Operating System do it? The existing Operating Systems have a great deal of differences in answering this question. Some (like UNIX) treat a file as a sequence of bytes. This is one extreme of the spectrum where the Operating System provides the minimum support. In this case, the Operating System does not recognize the concept of a record.

Therefore, the record length is not maintained as a file attribute in the file system of such an Operating System like UNrx or Windows 2000. Such an Operating System does not understand an instruction such as "fread" in C or "Read ...record" in COBOL. It only understands an instruction "Read byte numbers X to *Y".* Therefore, something like the application program or the DBMS has to do the necessary conversion. At a little higher level of support, some other systems treat rues as consisting of records of fixed or variable length. (like *AOSNS).* In this case, the Operating System keeps the information about record lengths, etc., along with the other information about the file.

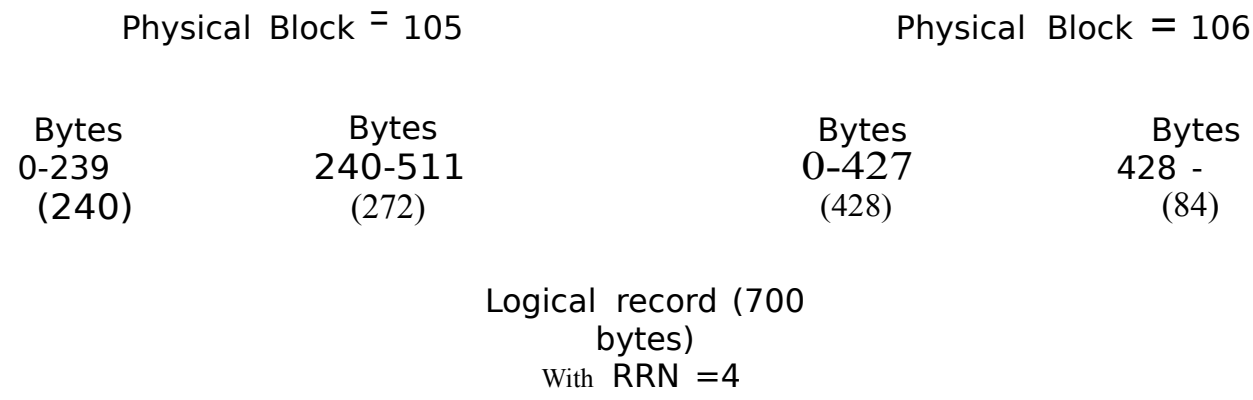
(i) RRN = 4 means RBN = 2800 (refer to Fig. 4.16). This means that the Operating System has to start writing at logical byte number 2800 in the file. We have to now decide which logical block this RBN falls in as per Fig. 4.17. Instead of referring to that figure, we would like a general formula that the Operating System can use.

(ii) Divide RBN by block length, i.e. 2800/512. We get quotient = 5 and remainder = 240. Therefore, 5 becomes the Logical Block Number (LBN). This means that the 2800th byte is the same as the 240th byte in the logical block number 5. Logical block numbers 0 to 4 will occupy RBN = 0-2559 (logical block number 5 starts at RBN = 2560 (Fig. 4.17). RBN = 2800 falls in this block. RBN = 2560 would be the 0th byte in LBN = 5. RBN = 2561 would be byte number I in LBN =

5. Therefore, extrapolating this logic, RBN = 2800 would be byte number (2800-2560) or byte number 240 in LBN = 5. This quite fits into our formula.

(iii) The Operating System now has to translate the logical block number (LBN) into the Physical Block Number (PBN). Logical block number 0 corresponds to physical block number = 100, because physical block numbers 100-1099 are allocated to this file. Therefore, logical block number 5 of this file is the same as physical block number 100 + 5 = 105.

(iv) Therefore, the Operating System knows that it has to start writing from byte number 240 of the 105th physical block of the disk.



~ Fig. 4.18 Writing a logical record onto pbysical blocks

However, there are only (511-239) = 272 bytes left in block number 105 starting from 240. Therefore, the Operating System has to continue writing the logical record into the next block tool It will have to use the remaining (700-272) = 428 bytes (byte numbers 0-427) of block . 106.This is shown in the Fig. 4.18.

Therefore, to write the fifth customer record (RRN = 4), the Operating System will have to write 272 bytes at the end of physical block number 105, and 428 bytes at the beginning of physical block number 106, because 272 + 428 = 700 which is the record length.

(v) Tbe DO portion of the Operating System now translates the physical block numbers 105 and 106 into their physical addresses as discussed earlier, based on the scctorlblock numbering scheme. For instance, given the disk characteristics of 8 surfaces of 80 tracks each, where there are 10 sectors per track, we get the following equation.

Block 105: Surface = 2, Track (or Cylinder) = I, Sector = 5.

Block 106: Surface = 2. Track (or Cylinder) = I, Sector = 6.

The reader should verify this, keeping in mind our numbering scheme. The track number is synonymous with the cylinder number. Cylinder 0 bas blocks 0-79, cylinder 1 has blocks 80-159, and so

Alternatively, the c-ursorcan directly give the RBN itself. In this case, after a record is read, the cursor is incremented by the number of bytes read (i.e. the RL) which is supplied through the system call. All this is done by the compiled AP itself at the time of execution. At this juncture, the compiler generated system call to UNIX for reading the number of bytes equal to the record length. starting from the computed RBN is executed. Therefore. somebody has to finally do the work of this translation if an instruction in an AP, written in a language such as *C/COBOL,* is to be supported. Whether the compiler does it or the Operating System is the main question.

4.2.6 The Relationship between tbe Operating System and DMS

As we have seen, the Operating System can present to the A.P, records in the same sequence that they have been written. i.e. the way they bad been presented by the AP to the Operating System for writing in the first place. If another AP wants to process the records in a different sequence-say by customer number, it is the duty of the AP to ensure that they are presented to the Operating System in that fashion. so tbat they are also retrieved in the same fashion. If that sequence is different, wbat should be done? One way is to sort the original file in the customer number sequence if all the records are to be processed in tbat sequence only.

Alternatively, if only some records are to be selectively processed in that sequence as in a query (given customer number, what is his name, address or balance?), it is advisable to maintain some data structure like an index to indicate where a specific record is written. One advantage of this scheme is that you can access the file in the original sequence as well as the customer number sequence. A sample index on customer number as the key is shown in Fig. 4.22. Notice that RBN is used to indicate the address of that specific record. We will revisit this later.

Customer record Number for thai	RBN of the Customer
COOl	0
C003	2100
COO4	1400
C009	700
..	..
..	..

Fig. 4.22 A Customer Number index

Normally, there is another piece of software to maintain and access these indexes. It is known as the- Data Management Systems (OMS). It is obvious that the index has to be in the ascending sequence of the key. if the search lime is to be reduced. If a new record is to be added to the customer file, it will be written by the Operating System in the next available space on the disk for that file, and therefore, may not necessarily be in the sequence of customer number. However. that does not matter anymore, because, the index is maintained in the customer number sequence.

As soon as a record is added, the Operating System knows at what RBN the record was written. For instance, we have studied in Sec. 4.2.4 that the fifth record, i.e. with RRN = 4 was written at RBN = 2800 if the RL was 700_ In fact, at any time the File directory maintains a field called file size. A new record has to be written at RBN = file size. After the record is written, the file size field is incremented by the RL_The Operating System can pass this to OMS and the OMS can then add an entry consisting of the key and the RBN of the newly added record to the index at an appropriate place to maintain the ascending key sequence, Let us assume that till now. four records are written with RBNs as shown in Fig. 4.22. As is obvious from the RBNs, they have been written in the sequence of COOl, COO9,COO4and

this directory entry given a file name. Therefore, the algorithm for this is very simple, though quite time consuming during execution.

FileName
File Extension
File Size· Current and maximum
File Usage Count
Number of processes having this file open
File Type (binary, ASCII, etc.)
File Record tength (if records are recognized)
File Key length, positions (if ISAM is a part of Operating
System) File Organization (Sequential, Indexed.
Random)
File Owner. Creator (usually the same)
File Access Control Information
File Dales (Creation, last usage.
etc.) Other information
File Address (Block Number) of the first block

..:..:.;,,Fig. 4.25 File Directory Entry or VTOC

The only thing of significance at this stage for address translation is the file address field for each file. This signifies the address (i.e. block number) of the first block allocated to the file. If the allocation is contiguous, finding out the addresses of the subsequent blocks is very easy. If the allocation is chained or indexed, the Operating System has 10 traverse through that data structure to access the subsequent blocks of the same ftle. When you request the Operating System to create a file with a name, say CUST.MAST and request the Operating System to allocate 1000 blocks, the Operating System creates this file directory entry for this file. As in the last example, if blocks 100-1099 are allocated to it, the Operating System also creates the file address within the file directory entry for that file as 100. This is subsequently used by the Operating System for the I/O operations.

4.2.8 OPEN/CLOSE Operations

An AP written in *C/COBOL* or any other HLL has to "Open" a file for reading or writing. As we know, "Open" is an Operating System service. and therefore a compiler substitutes a system call in the place of the "Open" statement in the HLL program.

The system call for "Open" at the time of execution searches for a file directory entry for that file using the file name and copies that entry from the disk in the memory. Out of a large number of files on the disk, only a few may be opened and referred at a given time. The. list of the directory entries for such flies is caUed Active File List (AFL) and this Listin the memory is also arranged to allow faster access (index on AFL using file name, etc.). After copying in the memory. the Operating System ensures that the user is allowed to perform the desired operations on the file using the access control information, As we have seen. for calculating the physical address for every "Read" and "Write" statement, the starting block number (which was 100 in the previous example) in the file directory entry in the memory has to be added 10 the logical block number.

If the AP adds new records 10 a file, the corresponding file size is altered by the Operating System in the ftle directory entry in the memory (AFL). Similarly, any time the file is referred to or modified or

in the sorted list is such an entry. Therefore, blocks 41-47 will be allocated. The resulting two tables. similar to the ones shown in Fig. 4.28 can now be easily constructed. If 10 blocks were requested for a file, we would have to use the second entry of 16 blocks in the sorted li.stand allocate blocks 5 to 14 to the
new file. After this allocation, there would be only 16 - 10 = 6 free blocks left in this hole. As 6 is less
than 8, which is the number of free blocks in the first entry, the list obviously would need resorting. thereby consuming more time.

However. the best fit method claims to reduce the wastage due to fragmentation, i.e, the situation where blocks are free, but the holes are not large enough to enable any allocation. This is because, this method uses a hole just enough for the current requirement. It does not allocate blocks from a larger hole unnecessarily. Therefore, if subsequently, a request for a very large allocation arrives, it is more likely to be fulfilled.

The advocates of worst fit method do not agree. In fact, they argue that after allocating blocks 41-47, block number 48 which is free in the example above cannot be allocated at all. This is because it is far less likely to encounter a tile requiring only one block. Therefore, they recommend that the required 7 blocks should be taken from the largest slot, provided that it is equal to or larger than our requirement (i.e. 7). Therefore, by this philosophy, blocks 2001-2007 will be allocated, thereby leaving the remaining blocks with numbers 2008-6399 still allocable. This chunk is large enough to cater to other large demands. At some point, however, in the end, it is likely to have very few free blocks remaining and those would most probably be unallocable even in the worst fit scenario. But by then, some other blocks are likely to be freed. thereby creating larger usable chunks after coalescing. It is fairly straightforward to arrive at the resulting two tables after the allocation using this philosophy.

In either of these philosophies, the tables have to be recreated/resorted after creation/deletion of any file. In fact. after the deletion of a file, the Operating System has to check whether the adjacent areas are free and if so, coalesce them and create a newly sorted list To achieve this, the Operating System needs both the tables shown in Fig. 4.28. For instance, let us assume that the block allocation to various files is as shown in Fig. 4.28(a) at a specific time. Let us assume that the file named CUSTOMER is now deleted. The Operating System must now follow the following steps.

(i) It must go through the file allocation list as given in Fig. 4.28(a) to find that 52 blocks between 49 and 100 will be freed after the deletion.

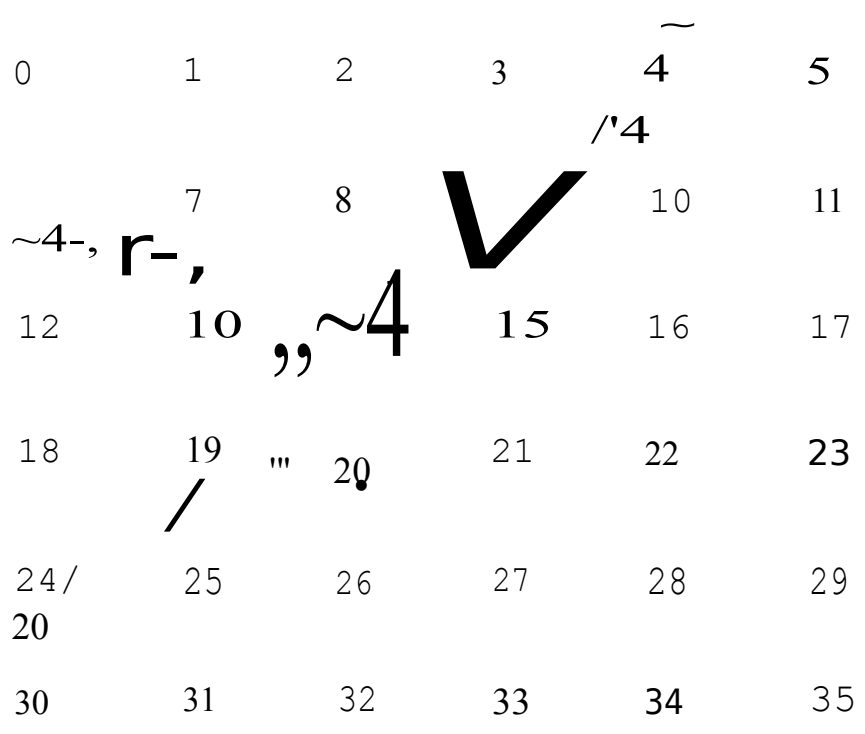
(ii) It must now go through the free blocks list as in Fig. 4.28(b) to find that free blocks 41-48 and 101-200 are free and are adjacent to the chunk of blocks 49-100. Therefore, it will therefore coalesce these three as shown in Figs 4.30 and 4.31 and work out a new free blocks list. This new list is shown in Fig. 4.32.

(iii) It will sort this new free blocks list, as shown in Fig. 4.33. The new list can be used later for best or worst fit algorithms.

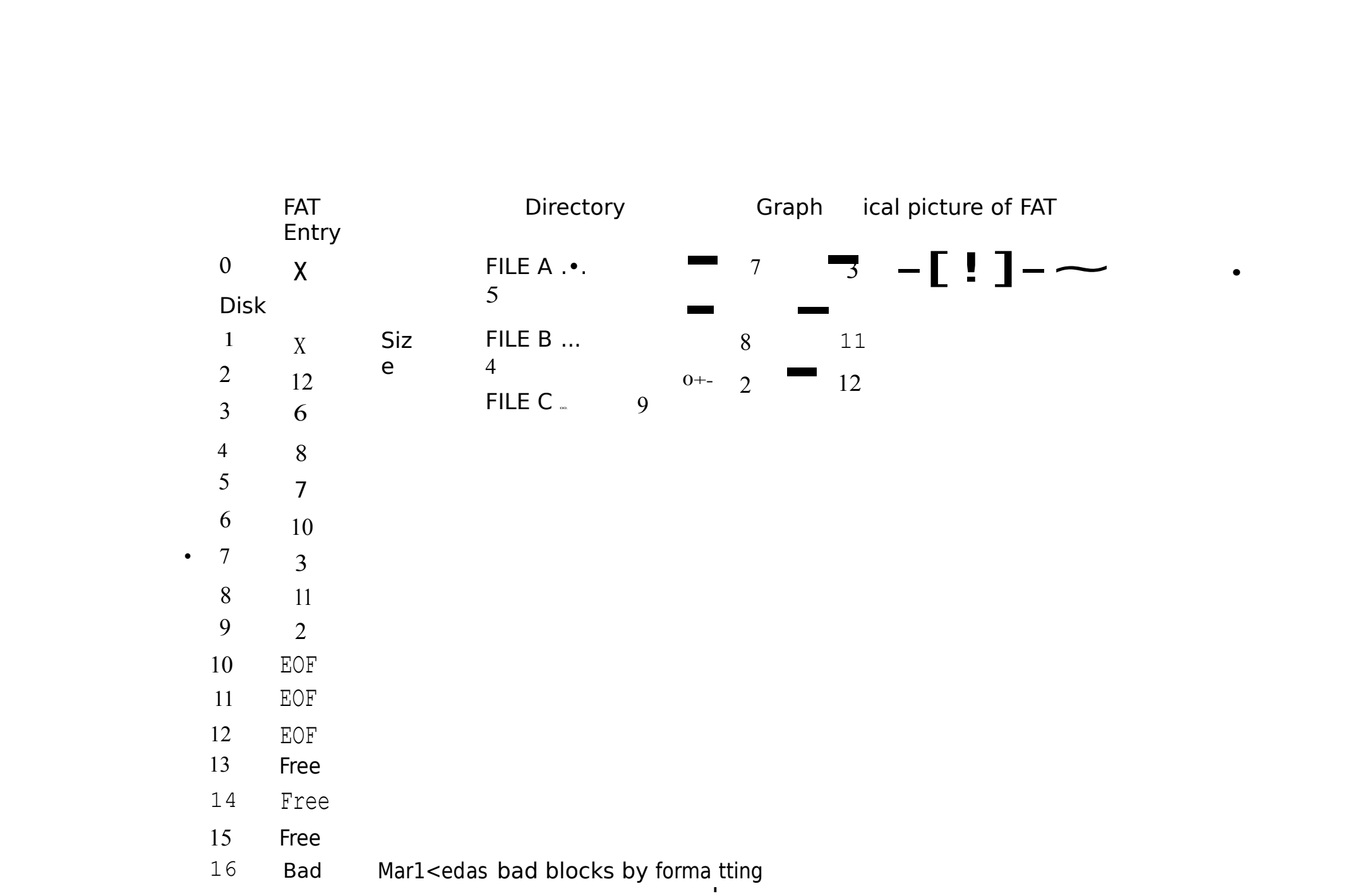
Fig. 4.30 Before coalescing

FIIE. A 4

File Directory Entry



:::::;:.F..ig. 4.36 Chained allocation



~ Fig. 4.37 MS-DOS, Windows 2000 or *OS/2* chained allocation

16 blocks

	0	1	2	..	15
File Name		5	7	3	6	10	.

~ Fig. 4.38 CP/M directory entry

If the file requires more than 16 blocks, the directory entry is repeated as many times as necessary. Therefore, for a file with 38 blocks, there would be 3 directory entries. The first two entries will have all the 16 slots used (16 x 2 = 32). The last entry will use the first 6 slots (32 + 6 = 38) and will have 10 slots unused. In each directory entry, there also is a field called "block count" which, if less than 16, indicates that there are some free slots in the directory entry. Therefore, this field will be 16, 16 and 6 in the 3 directory entry records in our example gi ven above. Figure 4.38 shows this field as 5 because there are only 5 blocks (5, 7, 3, 6 and 10) allocated to this file. This corresponds to FILE A of Fig. 4.37.

When the AP wants to write a new record on to the file, the following happens.

(a) The AP makes a request to the Operating System to write a record.

(b) If in a block already allocated to the file there is not enough free space to accommodate the new record, the Operating System calculates the number of blocks needed to be allocated to accommodate the new record, and then it acquires these blocks in the following manner.

- The Operating System consults the free blocks pool and chooses a free block. Let us say, 3 blocks are needed for this operation.
- It updates'the free blocks pool (i.e, removes those 3 blocks from the free blocks pool).
- It now writes as many block numbers as there are vacant slots in the current directory entry given by (16 - block count). If all block numbers (in this case 3) are accommodated in the same directory entry, nothing else is required. It only writes these block numbers in the va- cant slots of the directory entry and increments the block count field within that entry. How- ever, if, after writing some block numbers, the current directory entry becomes full (block count = 16), then it creates a new directory entry with block count", 0 and repeats this step until all required block numbers have been written.

(c) Now the Operating System actually writes the data into these blocks (i.e, into the corresponding sectors after the appropriate address translations).

Reading the file sequentially in this scenario is fairly straight forward and will not be discussed bere. For online processing, if you want to read 700 bytes starting from RBN =700 as given by the index in the last example in the section on chain allocation under non-contiguous allocation. it effectively means reading logical block numbers I and 2. The File System can easily read the directory entry and pick up the second and third slots (i.e. logical block numbers land 2-whicb correspond to physical blocks 7 and 3, as per Fig. 4.38). In the same way, picking up logical blocks 35 and 36 is not as difficult now as it was in chained allocation. The File System can easily calculates that the logical block numbers 35 and

36 will be available in the slot numbers 3 and 4 of the third directory entry for that file. Therefore, it can directly access these blocks improving the response time.

allocated as an index block. We are faced with the same problem. Which pointer-I 50 or 800--- should be maintained in the directory entry now? There are 2 index blocks, whereas there is space for only one pointer. The Operating System uses the same trick to solve this problem. Another block, say 51 is acquired and is assigned to a higher level index. This block also can contain .128 entries of 4 bytes each. Each entry in this case holds a pointer to the lower level index. The first two entries in block 51 are pointers to lower level indexes (150 and 800, respectively). The remaining J 26 slots in block 51 are unused at this juncture. Now, the file directory entry points [o block 5 I. A closer study of Fig. 4.42 will clarify how this system works.

» Step 4 If two levels of indexes are not sufficient. a third level of index is introduced. AOSNS supports 3 levels of indexes. It is easy to imagine this and therefore. it is neither discussed, nor shown.

Reading a file sequentially is fairly straightforward in this case. If the file is being processed from the beginning, the Operating System does the following to achieve this.

- From the file size maintained as a field in the file directory. the Operating System determines the index levels used. For instance, if file size is less than the clement Size, no index wiU be required. If file size is between I element to 128 elements, there will be one index, and so on. This tells the Operating System the meaning of the pointer in the file directory entry, as to the level of index to which it points.
- The Operating System picks up the pointer in the file directory and traverses down [.0 the required level as gi yen above te reach the data blocks.
- It now can read data blocks 41, 42, 43, 44, and so Oll after appropriate address translations, in thai sequence. Actually, normally, an AP will want to process the logical records sequentially. The translation from logical records to logical blocks and then physical blocks is already discussed earlier.
- After reading the data blocks in the controller's memory, relevant bytes from physical blocks are transmitted to the main memory to form a logical record which is presented to the AP.
- After block 44, the Operating System knows that one element is over, and it has to look at the next pointer in the index (which is 81 in this case).
- By the same logic, when data block 8 which is the last data block pertaining to that index block is read (refer to Fig. 4.42), the Operating System knows that it has to look up the next index block. whose address is given as the second pointer in the higher level index (which is in the block: number 800 in this case).
- By repeating this procedure, the entire file is read.

For online processing, the A.P will wake a request (.0 DMS to read a specific record, given its key. The DMS will lise its key index to extract the RBN and request the Operating System (o read that record. At this stage, *AOSNS* will determine the logical blocks to be read to satisfy that request Given the LBNs and the file size, AOSNS can find out the index level and the element which is to be read. Fer instance, from RBN, if the Operating System knows that LBNs 3 and 4 are to be read, the Operating System can work backwards to find out where these pointers will be. It knows that LBN 0-3 are in element 0 and LBN 47 in element I. Therefore, it wants to read the last block of element 0 + First the block: of element I in order to read the blocks with LBN = 3 and 4. These are physical blocks 44 and 81 respectively, as shown in Fig. 4.41. ·11also knows that elements 0127 are in index block 0. Therefore, the first t\VOentries io this index block (given by block 150 shown in the figure), therefore will give pointers to these two elements, i.e. element 0 and element I. It can read those pointers almost directly and then access the

traverse through the chains or indexes (as per the allocation method) to access the block number allocated to that file before it can do this job.

- It removes the BFD entry for that file.

It should be noted thai all these changes in the BFD are made in the in-memory BFD images first using various data structures described above. Periodically (for better recovery), and finally in the end at the time of shut down, the updated data structures are copied onto the disk at the appropriate locations in the BFD on the disk, so that next time you get the updated values when the system is used again.

An algorithm to create a file under a directory is almost reverse of this and can be easily imagined.

An algorithm for creating a soft link for a tile/directory essentially parses the path name, locates the BFD entry and increments its usage count. It also inserts the file/directory name in the required SFD with the BEN same as for the file being linked.

If the user wants to go up in a directory. the entry with" ." in the SFD can be used. For instance, if we want to traverse from PURCHASING -> COSTING -> BUDGET, using a relative path name ./ BUDGET when we are in PURCHASING directory, the following algorithm is executed.

(i) The Operating System will parse the path name.

(ii) The Operating System will read the " .... entry in the SFD of PURCHASING as shown in Fig. 4.48(d). It gives BEN = 5. Tbis is the BEN of the parent directory (which in this case is the COSTING directory).

(iii) It will access BFD entry with BEN = 5.

(iv) It will know that it is a directory starting at block 75. It will verify its access rights and then read it to get the SFD of COSTING. The contents of the SFD for COSTING are as shown in Fig. 4.48(c).

(v) It will now perform the search for a name BUDGET in the SFD for COSTING and will store its BEN = 10.

(vi) Having located tbe desired file. it will proceed to take any further permissible actions.

The data structures that are maintained in the memory have to take care of various requirements. They bave to take into account the following possibilities.

- One process may have many files open at a time, and in different modes.
- One file may be opened by many processes at a time and in different modes.

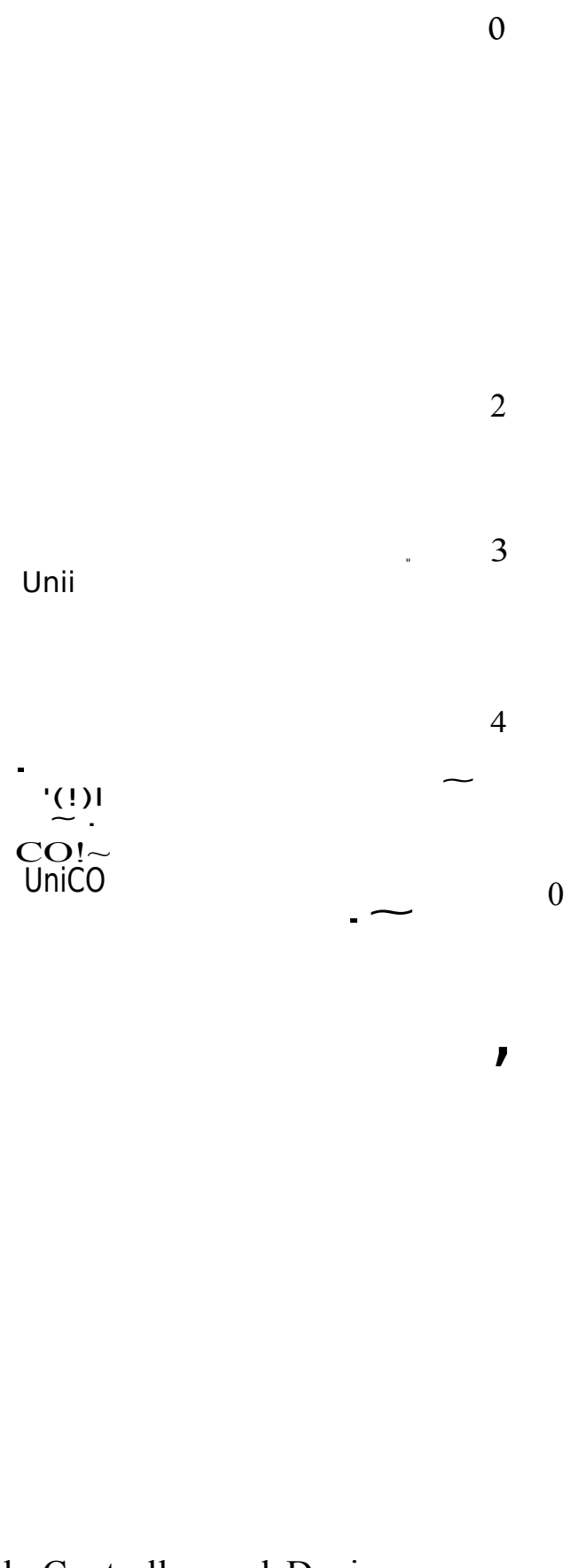
For instance, a Customer file may be opened by three processes simultaneously. One may be printing the name and address labels sequentially. Another may be printing the statements of accounts, again sequentially, and the third may be answering an online query on the balances. It is necessary to maintain separate cursors to denote the current position in the file to differentiate an instruction to read the next record in each case, so that the correct records are accessed.

The exact description of these data structures and algorithms is beyond the scope of this text, though it is not difficult to imagine them. This brings us to the end of our discussions about the File Systems. We need to examine the DDs more closely to complete the picture.

4.3.1 The Basics

We have seen the functions of a disk controller in the previous sections. We have also seen various instructions that this controller understands, i.e. the instruction set of the controller and how the DD uses

In some mainframe computers such as ffiM-370 family (i.e. ffiM 370. 43XX, 30XX, etc.), the functions of a controller are very complex, and they are split into two units. One is called a Channel and the other is called a Control Unit (CU). Channel sounds like a wire or a bus, but it is actually a very small computer with the capability of executing only some specific *1/0* instructions. If you refer to Fig. 4.54, you will notice that one channel can be connected to many controllers and one controller can be connected to many devices. A controller normally controls devices of the same type, but a channel can handle controllers of different types. It is through this hierarchy that finally the data transfer from/to memory/device takes place. It is obvious that there could exist multiple paths between the memory and devices as shown in Fig. 4.54. These paths could be symmetrical or asymmetrical as we shall sec.



~-.~, Fig. 4.54 Channel, Controller and Devices

(ii) Control Unit Control Block (CUCB)

This data structure maintains the following information.

- Control Unit 10.
- Control Un~ Status (busy. notfunclioning etc ..)
- List of devices connected 10 Ihis CU
- List 01 channels connected to this CU
- List 01 processes waiting for this CU
- Current process using this CU
- Some other infonnation

Fig. 4.58 Control unit control block (CUCB)

(iii) Device Control Block (DeB)

This data structure maintains the following information.

- Device 10.
- Device status (busy, not functioning etc.)
- Device characteristics
- Device descriptor
- list of CUs connected to this device
- List of processes waiting for this device
- Current process using this device
- Salne other information

Fig. 4.59 Device control block (DCB)

In the Device Control Block (DCB). we maintain device characteristics and device descriptor to achieve a kind of device independence. The idea is to alJow the user to write the *1/0* routine in a generalized fashion so that it is applicable for any device once the parameters for that device are supplied to that routine. These parameters are the same as the device characteristics in the DCB. Therefore, the idea is that whenever an Operating System wants to perform an *1/0* fOT a device. it reads the DCB for that device and extracts these Device characteristics. It then invokes the "common 110 routine" and supplies these Device characteristics as parameters. The ultimate objective is to have only one common 1/0 routine. However. this is quite an impractical goal. A via media is to have a common routine for the same types of devices at least.

If you study the contents of the DCB. it is easy to imagine how the Operating System would maintain these fields. Most of them could be updated at the time of system ge.neration. But the list of processes waiting for that device changes with time, and is updated at run time.

How does the 1/0 procedure maintain this list? It does this by creating a data structure called Input! Output Request Block (IORB) for each process waiting for that device. It contains the following information.

Operating System. Regardless of the implementation, the algorithms are similar and interesting. Let us therefore study them.

(0) First Come First Sef1led (FCFS)

Figure 4.63 illustrates the FCFS method. The figure depicts four requests, 0, I, 2 and 3. It also shows these requests on the figure at their respective target track numbers. If the requests have arrived in the sequence of 0, 1,2 and 3. they are also serviced in that sequence, causing the head movement as shown in the figure starting with the R/W head position which is assumed to be between the target tracks of requests 2 and 3.

FCFS is a 'just' algorithm, because, the process to make a request first is served first, but it may not be the best in terms of reducing the head movement as is clear from the figure. To implement this in the Operating System. one will have to chain all the JORBs to a DCB in the FIFO sequence. Therefore, the DCB at any time points to the next 10RB to be scheduled. After one 10RB is dispatched, that 10RB is deleted, and the DCB now points 10 the next one in time sequence, i.e. which came later. When a new 10RB arrives, it is added at the end of the chain. In order to reduce the time to locate this 'end of the chain', the DCB can also contain a field "Address of the Last 10RB" for that device. For recovery purposes. normally lORB chains like all others are maintained as two-way chains, i.e. each 10RB has an address of the next IORB for the same device as well as the address of the previous IORB for the same device. We can now easily construct the algorithms to maintain these 10RB queues for this method.

lf▷▷c" Fig. 4.63 FCFS method

(b) Shortest Seek Time First (SSTF)

Figure 4.64 illustrates the SSTF method. This method chooses the next 10RB, where the seek time is the shortest - with respect to the current head position. The figure depicts four 110 requests again. As in the last figure, here too, the boxes denote the target track numbers in the 10RBs for those requests. The requests are serviced in the order of 0, I, 2 and 3 as shown in the figure. In fact we have deliberately numbered the requests in the sequence that they are serviced and not the one in which they have arrived. This is done so that we understand the algorithm better. This is not the sequence in which the requests may have necessarily arrived. It improves the throughput, but it may not be a 'just' method. A request which was made earlier may not get serviced faster. Also, it can delay some requests for a very long time.

~ Fig. 4.64 SSTF method

4.3.6 Device Handler

The device handier essentially is a piece of software which prepares an *IIO* program for the channel or a controller, loads it for them and instructs the hardware to execute the actual *1/0.* After this happens, the device handler goes to sleep. The hardware performs the actual *VO* and on completion. it generates an interrupt. The ISR for this interrupt wakes up the device handler again. It checks for any errors (remember, the controller has an errcr/status register which is set by tbe hardware if there are any errors). U'there are no errors, the device handler instructs the OMA to transfer the data to the memory.

4.3.7 Interrupt Service Routine (ISR)

We have discussed this before and therefore it needs no further discussion.

4.3.8 The Complete Picture

Let us now take an example to study in a step-by-step manner, how all these pieces of software are interconnected. When an AP wants to read a logical record, the following happens.

(i) The AP-I has a system call to read a logical record for an Operating System which can recognize an entity such as a logical record. For systems such as UNIX which treat a file as a stream of bytes, a system call to read a specific number of bytes starting from a given byte is generated in its place. Upon encountering this. AP-l is put aside (blocked) and another ready process, say AP-2 is initiated.

(ii) The Pile System determines the blocks that need to be read for the logical record of AP-l, and requests DO to read the same. We have seen bow this is achieved.

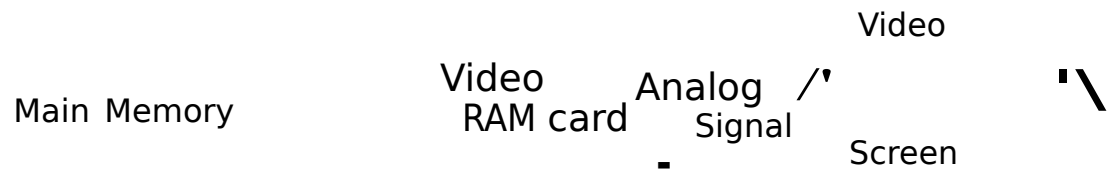
(iii) The *VO* procedure within DD prepares an IORB for this *VO* request.

(iv) The *VO* procedure within DD now establishes a path and chains the IOR.B to the device control block and other units as discussed earlier. The 1/0 procedure actually constructs an 10RB and hands it over to the If0 scheduler. The *If* 0 Scheduler chains the IC)RB in the appropriate manner as per the scheduling philosophy.

(v) Whenever a device is free. it keeps on generating interrupts at reguJar intervals to attract attention ("I want to send some data" or "does anybody want to send anything to me? I am free"). The ISR for that interrupt wakes up the If0 scheduler. The *If* 0 scheduler checks up tbe IORB queues and then services them one by one, as per the scheduling philosophy. When the device is free, the controller may not be. Or even if it is, the channel may not be free at that time. For the actual *IfO.* the entire path bas to be free, and this adds to the complication. The *If* 0 scheduler ensures this before scheduling any IORB. After this is done, it makes a request to the device handler to actu- ally carry out the [f0.

(vi) After an IORB has finally been scheduled, the *If* 0 scheduler now makes a request to the device handler to carry out the actual I/O operation.

(vii) The device bandler within DO DOW picks up the required details from the IORB (such as the source, destination addresses, etc.) and prepares a channel program (CP) and loads it into the channel which in turn instructs the controller. If there is no channel, the device handier directly instructs the controller about the source and target addresses and number of bytes to be read. As we know, the device handler can issue a series of instructions which can be stored in the controller's memory.


Video
Main Memory
Video
RAM card
Analog
Signal
Screen

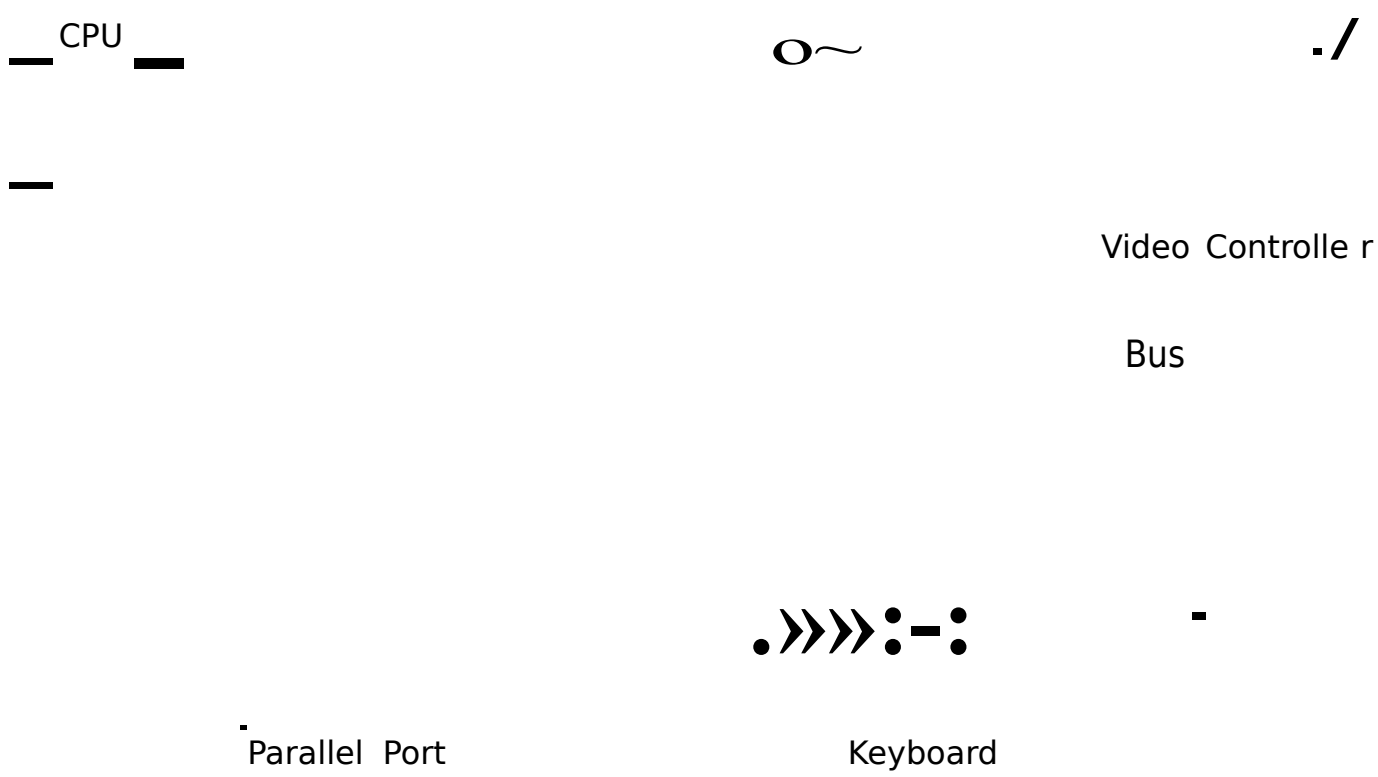


Fig. 4.70 Schematic of the terminals

Therefore, there are multiple memory locations involved in the operation. These are:

- A very small memory within the keyboard itself
- The video-RAM (data and attribute bytes)
- The Operating System buffers
- The *1/0* area of the AP (FD, working storage, etc.)

There are various Operating System routines and drivers to ensure the smooth functioning between the keyboard, the screen, all the memory areas and the Application Program itself. This is the subject of our next section.

4.4.3 **Terminal** Software

Imagine that an Application Program written in HLL wants to display something on the terminal. The compiler of the HLL generates a system call for such a request, so that at the time of execution, the Operating System can pick up the data from the memory of the AP, dump it into its own output buffers first and then send it from these buffers to the terminal. The rates of data transfers between the memory of the AP to the Operating System buffers and finally from there to the terminal are very critical if the user has to see the output continuously, especial! y in cases such as scrolling or when using the Page Upl Page Down facility, etc. What is transferred between the AP to the Operating System and finally to the terminal is the actual data as well as some instructions typically for screen handling (given normally as escape sequences).

Let us say that the AP wants to erase a screen and display some data from its working storage section. The AP will request the Operating System to do this. The Operating System will transfer the data from the working storage of the AP to its own buffers and then send an instruction (in the fonn of escape sequences) to the terminal for erasing the screen first. The terminal's microprocessor and the software running on it will interpret these escape sequences, and as a result, move spaces to the video RAM, so that the screen will be blanked out. The Operating System then will transfer the data from its buffers to

the pointer chains for thatClist as well as for CLF to reflect this change. in the same way. ita Cblock has served its purpose, tbe Operating System delinks it from that Clist and adds it to the pointer chains of CLF at the end of the chain. There is no reason why it should be added anywhere else. This is because, there is no concept of priority or preference in this case. Any free Cblock is as good as any other for its allocation. Our example clarifies this in a step-by-step manner.

Let us now imagine tbat the Operating System bas allocated the buffer for all the Clists for all the terminals put together, such that it can accommodate 25 Cblocks in all, This is obviously too small to be realistic. but it is quite fine to clarify our concepts. Let us assume that the Cblocks are assigned in the manner as shown in Fig. 4.74.

Terminal	Clist	Cbtock numbers
T.o	ClO	0.5.8.17.21
	Cll	1,4,15.20
	Cl2	2.10,22
Tl	—	—
—	CLF	3,6.7,9,11,12,13,14,16.18.19.23.24

Fig. 4.74 Allocation of Cblocks

> Step 1: Initial state: 1'.0 represent this, the Operating System maintains the data structures as shown in Fig. 4.75. These are essentially the readers of the chains,

Terminal Number	Clist Number	Cblock Numbers	
		Startin	Endin
TO	CLO	0	21
TO	ClI	1	20

TO	Cl2	2	22
	CLF	3	24

[redacted],Fig. *4.7S* Terminal Data Structure

The actual Cblocks would be as shown in Fig. 4.76. We can easily verify that if we start traversing the chain, using the starting Cblocks from Fig. 4.75 and the Next pointers in Fig. 4.76, we will gel the same list as shown in Fig. 4.74. Figure 4.76 shows only 5 Cblocks allocated to terminal TO in detail with their data contents. The others are deliberately nor filled up to avoid cluttering and 10 enhance our understanding.

We can make out from the figure that the user has keyed through the terminal TO. the following text "Clist number 0 for the terminal TO has 5 Cblocks." This is stored in Cblocks 0,5,8, 17 and
21. All Cblocks in this example are fully used because, there are exactly 50 characters in this text, and therefore. in this example, the start offset is 0 and last offset is 9 for all those Cblocks. As we know, tbis need not always be the case.

)0 Step 4: It is possible that. at any moment, terminal TO may acquire new Cblocks to its Clists or it may relinquish them. We have, however, assumed that the Cblocks for TO have nOI changed during the period of this data entry for T 1.

We now assume that all the J 6 characters are keyed in. The user then keys in a "Carriage Return (CRY'. AI this juncture, the terminal data structure and the Cblocks will continue to look as shown in Figs. 4.79 and 4.80, respectively (and therefore, not repealed here). except that Cblock 6 would be different and would look as shown in Fig. 4.81. Notice that the "last" offset has now been updated 10 5. This means that a new character should be entered at position 6.

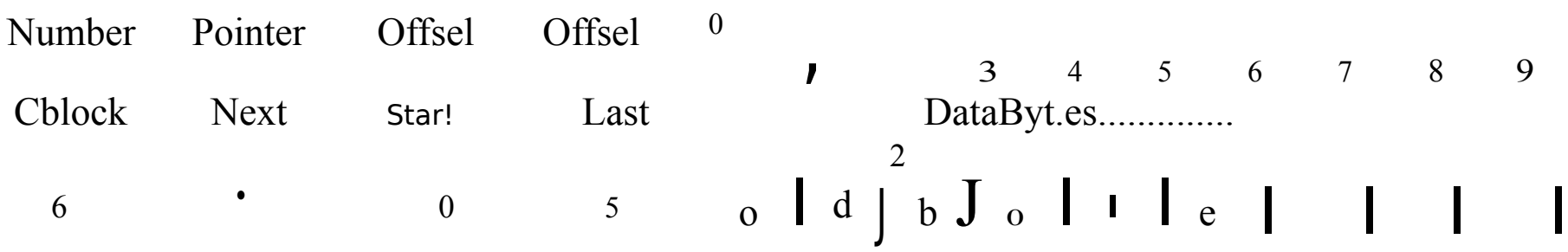


Fig. 4.81 Cblock 6 after tbe entry of aU the cbaractcfli in the naDle string

As soon as "CR" is hit, the Operating System routines for the terminals understand that it is the end of the user input At this juncture, a new routine memory; is invoked which moves all the 16 char-deters into Cblocksforthatterminal; the AP' 5 memory in a field appropriately defined (e.g. char[l7] in C or PIC X(16) in the ACCEPT statement or screen section of COBOL). After this. Cblock 3 and Cblock 6 are released and they are chained in the list of free Cblocks again. We also know that, if the name is to be displayed as each character is entered at that time itself, it wiU have been sent to the video RAM, from which it would have been displayed. After all the cbaracters are keyed in, the terminal data structure and the Cblocks again look as shown in Figs 4.75 and 4.76, respectively, This is where we had started from.

Begin
"Input char = 'CR'
then franster the entireClistto AP's Free the Clistand Else insen a character Inthe Clist
End"
'--_En_d .-'

FIg. .82 Input a character

We will now study some of the algorithms associated with Clists and Cblocks.

In these. "free the Clisr and Cblocks for that terminal" can be further exploded easily. We give below the algorithm for "insert a character" which is a little complex.

```
BegIn
        It the terminaldoes nothave a
            Cllst then Acquirea
            Clistnumber;
              Acquirea freeCblock;
                  Create a Clistentryinthe data
                          structure:
        Endll

        Access the "EndingCblockNumber"(say ECB)inthe Clist
                  tor that terminaltromthe data
                          structure;
        Access the Cblockwithnumber" ECB;
        Access the tast offset(LO)inthe
        Cblock: II Last Offset(LO)" 9
                  then Acquirea free
                      Cblock
        Endll IncrementLOby 1;
        MoveInputcharacterIntoByteno"
        LO
End.
```

.....;;F. ig. 4.83 Insert a character

(a) Application Program 10 Terminal 'Enter Customer Number'

- Step 1: The C/COBOL compiler would have generated the necessary code to push two parameters onto the stack, viz. the file descriptor terminal, as a is also treated as a file in UNIX, and memory address of the string to be displayed which in this case is "Enter Customer Number" which must have been already stored in the memory. The compiler would have also generated the code to call the library procedure for write (printfIDISPLA Y).
- Step 2: The library code pushes some constant on the stack (which depends upon UNIX imple- mentation) to indicate to the kernel that it is the write system call that needs to be executed. A mode switch from user to kernel is then done. The method to do this varies from CPU to CPU (On the 680 x 0 CPU, a mode switch is achieved by executing a trap instruction). The program is now executing in the kernel mode.
- Step 3: A table look up is done for the system call to determine validity, number of parameters etc. The arguments are then fetched from the stack. Depending on the arguments, the pertinent device driver is called (in this case, the terminal driver).
- Step 4: The character is read in by the terminal dri ver from the Application Program's address space to the kernel address space. Finally these characters are passed to the line discipline.
- SI.ep 5: The line discipline (canonical) places the characters on a Clist, Necessary processing is done (expansion of tabs to spaces, etc.). When the number of characters on the Clist becomes large or if the Application Program requests flushing. or the Application Program desires input from the keyboard, the line discipline invokes the driver output procedure. In this case. no pro- cessing is required in our text 'Enter customer number'. Therefore, the line discipline merely moves it to the Clist. As the AP desires input, the line discipline now invokes the driver.
- Step 6: The driver outputs the characters to the device 110 of the device (the terminal). All char- acters are sent (including escape sequence) to the terminal. The Application Program then goes to sleep.
- Step 7: Device *1/0* software within the terminal displays the characters (doing escape sequence processing for instructions like erase screen, display bold, display with underline, etc.). Position- ing of characters. wrapping long lines, etc. are done by this device 110 software running inside the terminal. This step sets up the video RAM with the appropriate data and attribute bytes and then the video controller comes into action for actually displaying the characters.
- Step 8: Upon completion of the display operation, the video controller interrupts the UNIX Operating System. If the process is sleeping on this device, it is awakened in the lSR.
- Step 9: The system call returns, the mode switches back to user mode and the Application Pro- gram continues.

(b) User Enters on the Keyboard 'OOOI<ENTER>' in response to AP's prompt.

- Step 1: We assume that the *C/COBOL* compiler would have generated the code necessary to push 2 parameters onto the stack (file descriptor of tbe terminal and the memory address of tbe variable where the data entered by the user, e.g, 0001 is to be moved). We also assume that the compiler would have generated the necessary code to call the library procedure for read (scanf! ACCEPT).
- Step 2: The library code pushes some constant (which depends upon the UNIX implementation) to indicate to the kernel that it is the read system call that needs to be executed. A mode switch from user to kernel is then done as seen before. The program is now executing in the kernel mode.

The Yellow book has defined two modes. Mode- 1 takes the form as shown in the figure Here, data part is 2048 bytes and error correction part is 288 bytes. However, not all applications need such strin- gent error correction mechanisms. For instance, audio and video applications should preferably have more data bytes. Mode-2 takes care of this. Here. all the 2336 bytes are used for data. Note that we are talking about three levels of error correction: (a) Single-bit errors are corrected at the byte-level, (b) Short burst errors are corrected at the frame level, and (c) finally any other remaining errors are corrected at the sector-level.

The definition of CD-ROM data format bas since been extended with the Green Book, which added graphics and multimedia capabilities in 1986.

4.5.3 DVD-ROM

The Digital Versatile Disk Read Only Memory (OVD·ROM) uses the same principle as a CD-ROM for data recording and reading. The main difference between the two. however, is that a smaller laser beam is used in case of a DVD. This results into a great advantage that data can be written on both the surfaces of a DVD. Also, the laser beam is sharper. This adds to the advantage of extra storage capacity: the tracks on a DVD are closer and hence pack more data. It is not possible in case of a CD. The typical capacity of each surface of a DVD is about 8.5 gigabytes (GB). Hence, together, they can accommodate
17GB! This is equivalent to the storage on 13
CDs.

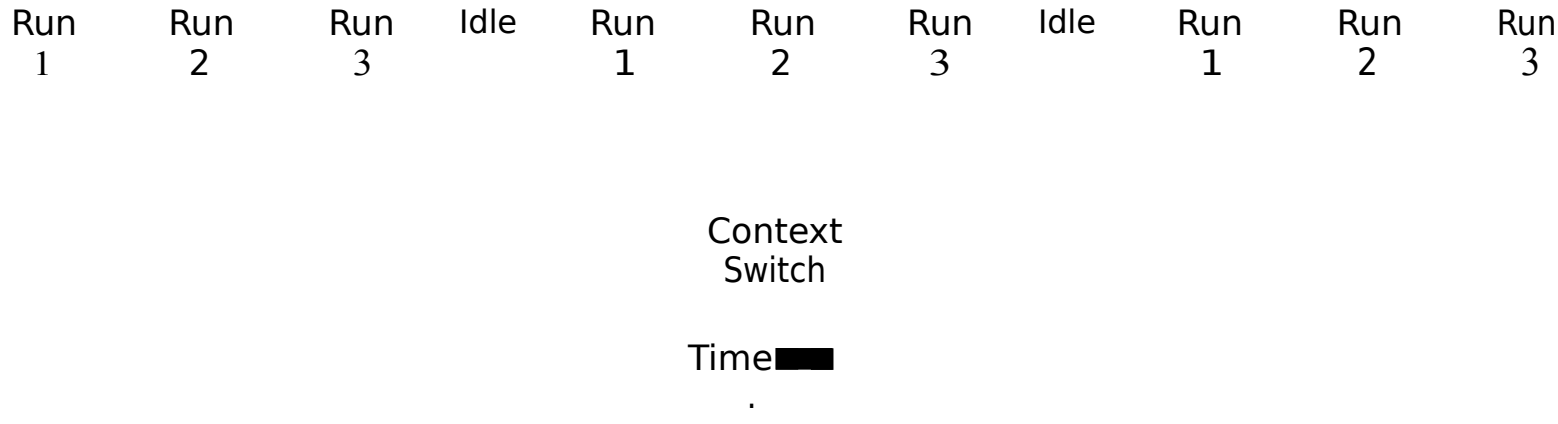
The main technical differences between a CD-ROM and a DVD are:

Characteristic	*CD-ROM*	*DVD·ROM*
Pit size	0.8 microns	0.4 microns
Spiral	1.6 microns	0.74 microns
Laser	0.78 microns	0.65 microns

- Access Control List
- Access Rights
- Active File Number
- Address CRe
- Bad blocks/sectors
- Best Fit
- Block
- Block Number
- Capability List
- Chained Allocation
- Channels
- Character List
- Coalescing
- Contiguous Allocation
- Control Unit
- Controller
- Count Data
- Current Address
- Cursor
- Access Control Matrix
- Active File List
- Address
- Address Marker
- Basic File Directory
- Bit Map
- Block Allocation List
- C-scan
- Cblock Number
- Channel Control Block (CCB)
- Character Block
- Cluster. Cluster Size
- Complete Path Name
- Contiguous area
- Control Unit Control Block (CUCB)
- Cooked/Canonical mode
- Count Key Data
- Current Directory
- Cycle Stealing

4.10 Describe first fit, best. fit and worst fit strategies for disk space allocations with their merits! demerits.
4.11 Why is it necessary to split the file directory entry into two directories, viz. Basic File Directory (BPD) and Symbolic File Directory (SFD)? How is sharing made possible due to this?
4.12 Describe broadly the device management functions within the Operating System.
4.13 How is path management done? What data structures help in this task?
4.14 Explain how the 110 requests for a busy device are queued up and scheduled.
4.15 Discuss all the 110 (disk arm) scheduling methods with their merits/demerits.
4.16 What is an Interrupt Service Routine (ISR)?
4.17 Using the File System and Device Driver, construct a complete picture as to how an Operating System can perform the read/write operations on behalf of the applications program.
4.18 Discuss the mechanism of data storage on a CD-ROM.
4.1.9 Explain benefits of DVD-ROM over CD -ROM.

process (e.g. register-to-register transfers or ALU calculations). However, between the bursts of data transfer, when there is no traffic on the data bus, the CPU can execute any instruction for the other process. This is the basis of multiprogramming.



~ Fig. 5.4 CPU utilization with three processes

The number of processes running simultaneously and competing for the CPU is known as the degree of multiprogramming. As this increases, the CPU utilization increases, but then each process may get a delayed attention, hence, causing a deterioration in the response time.

5.4 CONTEXTS~CHING

How is this context switching done? To answer this, we must know what is meant by the context of a process. If we.study how a program is compiled into the machine language and how each machine language instruction is executed in terms of its fetch and execute cycles, we will realize that at any time, the main memory contains the executable machine program in terms of Os and I s. For each program, this memory can be conceptually considered as divided into certain instruction areas and certain data areas (such as 110 and working storage areas). The data areas contain at any moment, the state of various records read and various counters and so on.

Normally, today modem compilers produce code which is reentrant, i.e. it does not modify itself. Therefore, the instruction area does not get modified during execution. But the Operating System cannot assume this. If we interrupt a process at any time in order to execute another process, we must store the memory contents of the old process somewhere. It does not mean that we have to necessarily dump all these memory areas on the disk, It is sufficient but not necessary to keep their pointers. Also, all CPU registers such as PC, IR, SP, ACC and other general purpose registers give vital information about the state of the process. Therefore, these also have to be stored. Otherwise restarting this process would be impossible. We will not know where we had left off and therefore, where to start from again. The context of the process precisely tells us that, which comprises both the entities mentioned above. If we could store both of these, we have stored the context of the process.

Where does one store this information? If the main memory is very small, accommodating only one program at a time, the main memory contents will have (o be stored onto the disk before a new program can be loaded in it. This will again involve a lot of *110* operations, thereby defeating (he ve.ry purpose of multiprogramming. Therefore. a large memory to hold more than one program at a time is almost a pre-requisite of multiprogramming. It is not always true, but for the sake of the current discussion, we will assume that the memory is sufficient to run all the processes competing for the CPU.

This means that even after the context switch, the old program will continue to be in the main memory. Now what remains to be done is to store the status of the CPU registers and the pointers to the memory

Figure 5.6 shows the way a process typically changes its states during the course of its execution. We can summarise these steps as follows:

(a) When you start executing a program, i.e. create a process, the Operating System puts it in the list of new processes as shown by (i) in the figure. The Operating System at any time wants only a certain number of processes to be in the ready list to reduce competition. Therefore, the Operating System introduces a process in a new list first, and depending upon the length of the ready queue, upgrades processes from new to the ready' list This is shown by the 'admit (ii)' arrow in the figure. Some systems bypass this step and directly admit a created process to the ready list.

(b) When its tum comes, the Operating System dispatches it to the running state by loading tbe CPU registers with values stored in the register save area. This is shown by the' dispatch (iii)' arrow in the figure.

(c) Each process is normally given certain time to run. This is known as time sHce. This is done so that a process does not use the CPU indefinitely. When the time slice for a process is over, it is put in the ready state again, as it is not waiting for any external event. This is shown by (iv) arrow in the figure.

(d) While running, if the process wants to perform some 110 operation, denoted by the *1/0* request (v) in the diagram, a software interrupt results because of the *1/0* system call. At this juncture, the Operating System makes this process blocked, and takes up the next ready process for dispatching.

(e) When the 110 for the original process is over, denoted by 110 completion (vi), tbe hardware generates an interrupt whereupon the Operating System changes this process into a ready process. This is called a wake up operation denoted by (vi) in tbe figure. Now the process can again be dispatched when its tum arrives.

(f) The whole cycle is repeated until tbe process is terminated.

(g) After termination, it is possible for the Operating System to put this process into the halted state for a while before removing all its details from the memory as shown by the (vii) arrow in the figure. The Operating System can however bypass this step.

The Operating System, therefore, provides for at least seven basic system calls or routines. Some of these are callable by the programmers whereas others are used by the Operating System itself in manipulating various things. Tbese are summarised in Fig. 5.7.

(I) Enter (Procees-ld) .... New

(II) Admit	(Procees-Id)	New	....	Ready
(III) Dispatch	(Proces&-Id)	Ready	....	Running
(Iv) TImeup	(Proc:ess-ld)	Running	....	Reedy
		Running	....	Blocked
(v) Block	(Process-ld)			
(vi) Wakeup	(PYocess-ld)	Blocked	_'	Reedy
(vII) Halt	(Process-ld)	Running	....	Halted

---" Fig. 5.7 System calls for Process state transitions

For each system call. if the process-id is supplied as a parameter, it carries out the process state transition. We will now study how these are actually done by the Operating System.

Process
Management

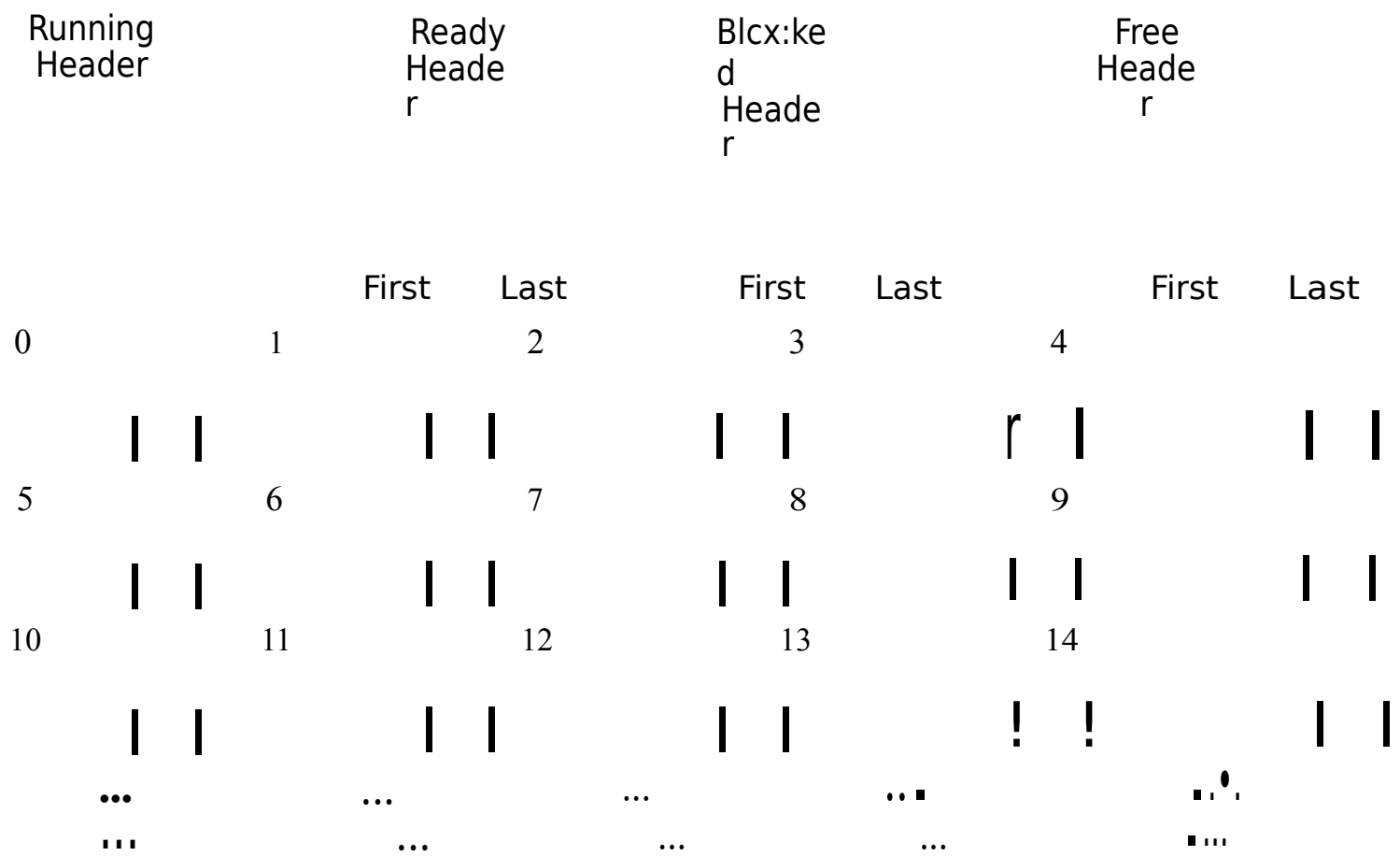


Fig. *S.9* PCBs in the memory

Whenever a process terminates, the area for that PCB becomes free and is added to the list of free PCBs. Any lime a new process is created, the Operating System consults the list of free PCBs first, and then acquires one of them. It then fills up the PCB details in that PCB and finally links up that PCB in the chain for ready processes.

We assume that. to begin with, at a given time, process with Pid = 3 is in running state. Processes with

Pid = 13.4,14 and 7 are in ready stale. Processes with Pid = 5, 0, 2. 10 and 12 are in the blocked state. PCB slots with PCB number = 8, 1,6, 11,9 are free (we have shown only Pids 0-14). The same is"'fttown in Fig. 5.10.

10 the PCB list of blocked processes or that of the free PCBs, there is no specific order or sequence. A list of free PCBs grows as processes are killed and PCBs are freed, and there is no specific order in which that will necessarily happen. The Operating System can keep the blocked list in the sequence of process priorities. But that rarely helps, because that is not the sequence in which their 110 will be necessarily completed 10 move them to the ready state. On the other hand, the ready processes are normally maintained in a priority sequence. For instance, a process with Pid = 13 is the one with the highest priority and the one with Pid = 7 is with the lowest priority in the list of ready processes shown in Fig. 5.10.

In Stich a case, at the time of dispatching the highest priority ready process, all that the Operating System needs to do is to pick up the PCB at the head of the chain. This can be easily done by consulting the header of the list (which gi ves the PCB with Pid = 13 as shown in Fig. 5.10) and then adjusting the beader to point to the next ready process (which is with Pid = 4 in the figure). U the process scheduling philosophy demanded the maintenance of PCBs in the ready list in the FIFO sequence as in the Round

Process Managemenz

• Create a process	(Pid
• Killa process	)
• Dispatch a process	(PId)
• Change the priority 01 a process	(Pld)
• Block a process	(Pld. New
• Time up a process	Priority) (PId)
• Wake up a process	(Pid)
	(Pid)

Fig. *5.14* Common operations on the processes

In the subsequent sections, we will COli sider these one by one, each time giving a step by step procedure and the accompanying process state transitions. We will assume that in the system, there are many processes in different states, as depicted in Fig. 5.10 to begin with. We will construct an imaginary sequence of events to study the process state transitions more closely. While doing that, we will create a process, kill a process, dispatch a process, change tbe process' priority, block a process due to an *YO* request, dispatch yet another process, time up a process and wake up a process. Essentially, we are trying to simulate a realistic example.

When you sit at a terminal and give a command to the CI to execute a program or your program gives a call to execute a sub-program, a new child process is created by executing a system call. The Operating System follows a certain procedure to achieve this which is outlined below,

1. The Operating System saves the caller's context . If you give a command to the CI, then the CI is the caller. If a sub-program is being executed within your program, your program is the caller, In both the cases, the caller process will have its PCB. Imagine that you are executing process A, as shown in Fig. 5.15.

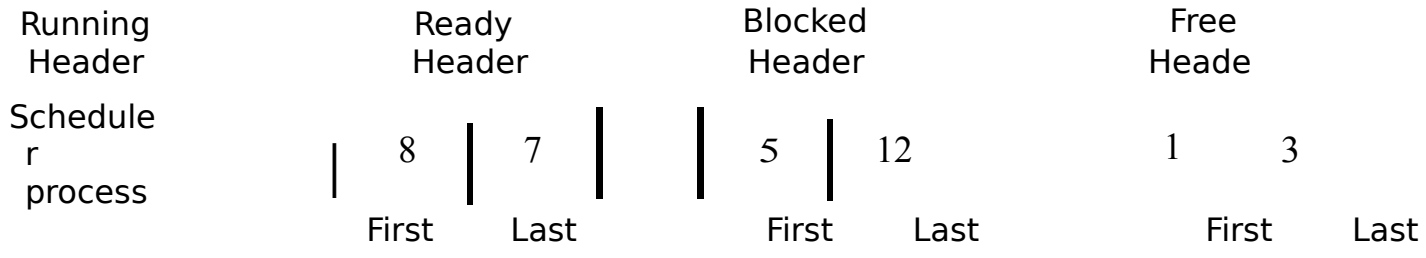
```
Begin
    0          Read
           cust·file
              Add ...
                 .
7              Divide
           ...
              8
       Call'SORr'
          9
       Add...
End
```

Fig. *S.lS* A program and a sub-program

After the divide instruction (instruction number 7), the program calls another sub-program at instruction 8, after the completion of which the main program must continue at instruction 9. At the time of execution, instruction 8 gives rise to the creation of a child process. The point is

that after the child process is executed and terminated, the caller process must continue at the proper point (instruction 9 in this case).

As we know, while executing instruction 8, the program counter (PC) will have already been incremented by 1. Hence, it will already be pointing to tbe address of instruction 9. Hence, .ithas to be saved so that when it is restored, the execution can continue at instruction 9. This is the


Running Header
Ready Header
Blocked Header
Free Heade
Schedule r process
8
7
5
12
1
3
First
Last
First
Last
First
Last

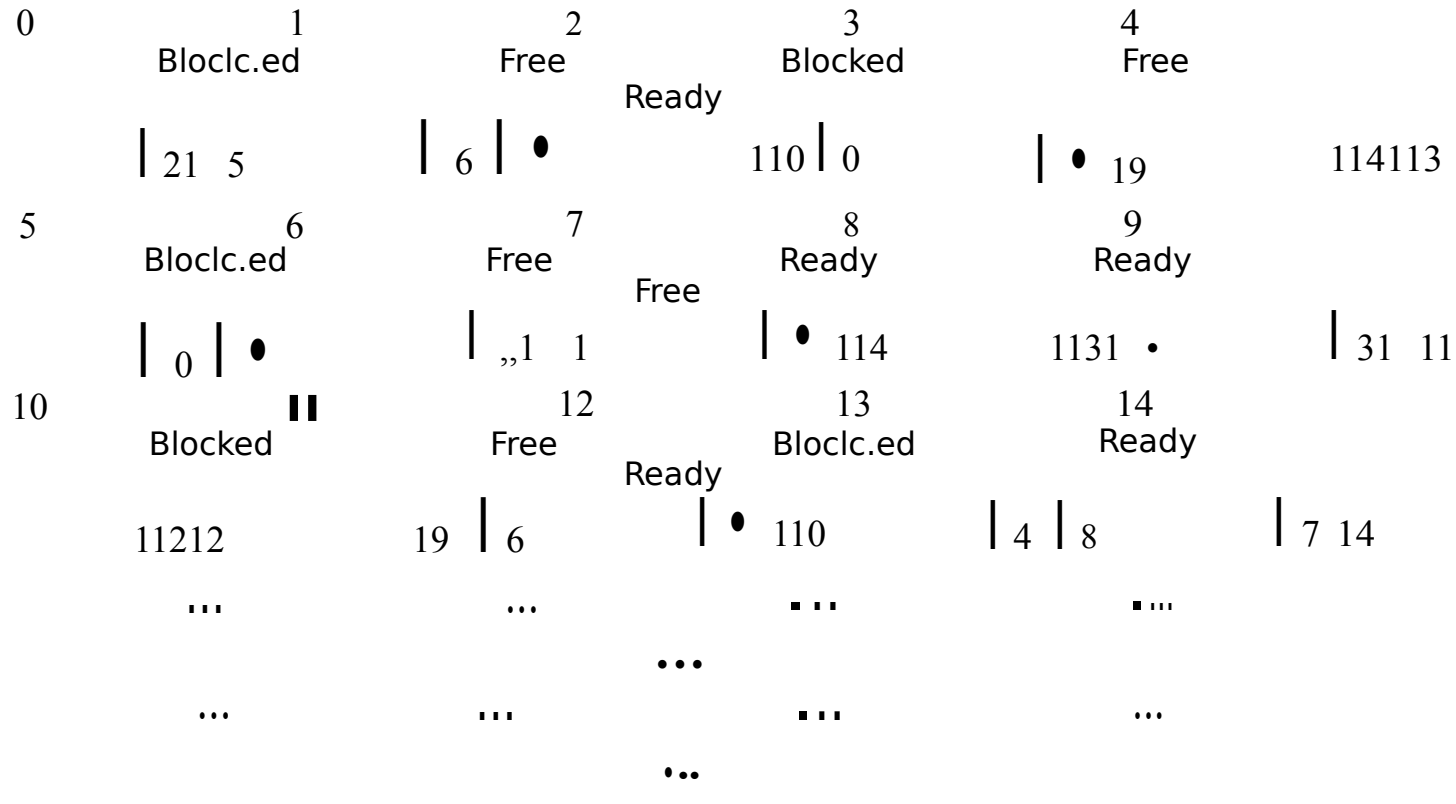


Fig. *S.18* PCBs after killing a process with Pid = 3

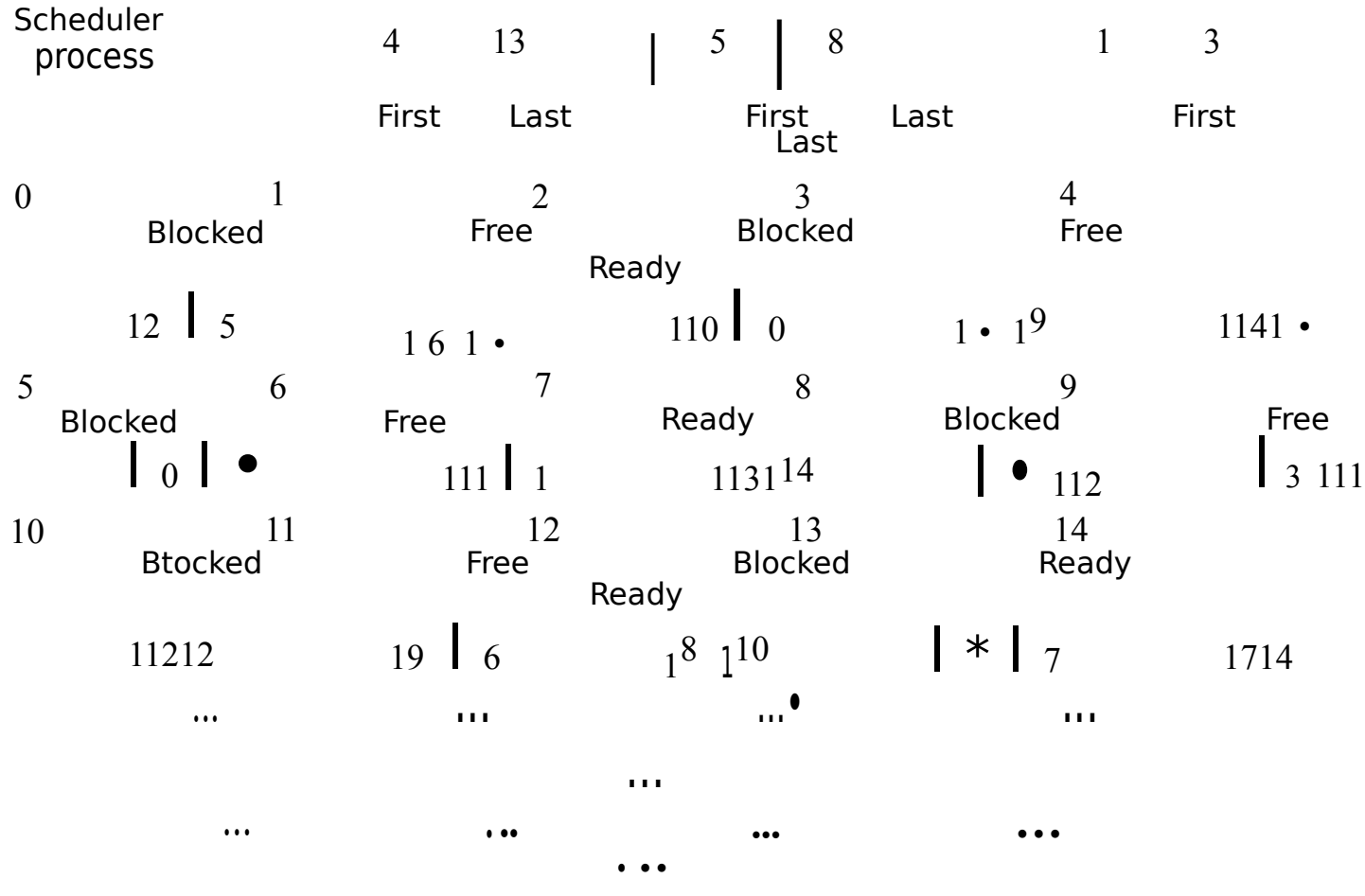
5.1

7 7. tit 77 b777 727'7' 777'. _"T .>7' 3!!C ~;:=*'

We will simply assume that the currently running process gets over (e.g. process with Pid = 3 as in the last example) and therefore. there is a need to dispatch a new process. We will assume that the Operating System has fmished the kill process procedure as outlined in Sec. S. LI. The PCBs will be in a state as shown in Fig. 5.18.

I. The Operating System accesses the ready header and through it, it accesses the PCB at the head of the chain, In this case, it will be the PCB with Pid = 8.
2. It removes PCB 8 from the ready list and adjusts the ready header. It changes the status of PCB 8 to running. The PCBs will look as shown in Fig. 5.19.
3. The Operating System updates the running header to Pid = 8.
4. The Operating System loads all the CP'U registers with the values stored in the register save area of PCB 8.
5. The process with Pid=8 now starts executing where it had left before or from the first executable instruction if it has just started executing.
6. The master list of known processes as shown in Fig. 5.17 is also updated

2. It now updates the status of that PCB to ready. It may be noted that the process is not waiting for any external event, and so it is not blocked.
3. The process with Pid = 13 now is linked to the chain of ready processes. This is done as per the scheduling philosophy as discussed before. Meanwhile, Jet us assume that, externally, the priorities of all other ready processes have been increased more than that of 13, and hence, the .PCB with Pid = 13 is added at the end of the ready queue. The ready header is also changed accordingly.
4. The running header is updated to denote that the scheduler process is executing.
5. The master list of known processes, as shown in Pig. 5.17 is now updated to reflect this change. The PCBs now look as shown in Fig. 5.22.



~ Fig. S.22 Time up a process with Pid =
13

When tbe I/O for a process is completed by hardware, before the execution of tbe wake up system call. the following things happen:

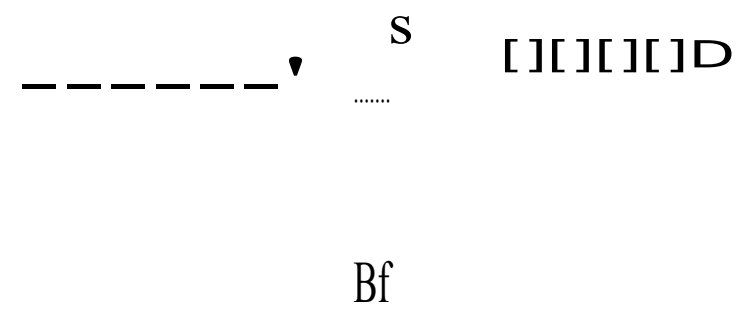
(i) The hardware itself generates an interrupt.

(ii) The device which has generated tbis interrupt and the corresponding interrupt service routine (ISR) are identified first (directly by hardware in the case of vectored interrupts .or by software routine otherwise).

(ill) The ISR for that device is executed.

Fairness refers to being fair to every user in terms of CPU time that he gets. Throughput refers to the total productive work done by all the users put together. Let us consider traffic signals as an example (Fig. 5.26) to understand these concepts first and then see how they conflict.

There is a signal at the central point S whicli allows traffic in the direction of AB, BA or CD and DC. We assume the British method of driving and signals in our examples. Imagine that there are a number of cars at point S, standing in all the four directions. The signalling system gives a time-slice for traffic in every direction. This is common knowledge. We define throughput as the total number of cars passed in all the directions put together in a given time. Every time the signal at S changes the direction, there is some time wasted in the context switch for changing the lights from green to amber and then subsequently to red. For instance, when the signal is amber, only the cars which have already started and are half way through are supposed to continue. During this period, no new car is supposed to start (at least in principle) and hence, the throughput during this period is very low.



II&- Fig. 5.26 Traffic signals

If the time slice is very high, say 4 hours each, the throughput will be very high, assuming that there are sufficient cars wanting to travel in that direction. This is true, because there will be no time lost in the context switch procedure during these 4 hours. But then, this scheme will not be fair to the cars in all the other directions at least during this time. If this time slice is only 1 hour, the scheme becomes fairer to others but the throughput falls because the signals are changing direction more often. Therefore, the time wasted in the context switch is more. Waiting for 1 to 4 hours at a signal is still not practical. If this time slice is 5 minutes, the scheme becomes still fairer, but the throughput drops still further. At the other extreme, if the time slice is only 10 seconds, which is approximately equal to the time that is required for the context switch itself, the scheme will be fairest, but the throughput will be almost 0. This is because, almost all the time will be wasted in the context switch itself. Hence. fairness and throughput are conflicting objectives. Therefore, a good policy is to increase the throughput without being unduly unfair.

The Operating System also is presented with similar choices as in the case of street signals. When the Operating System switches from one process to the next, the CPU registers bave to be saved/restored in addition to some other processing. This is clearly the overhead of the context switch. and during this period, totally useless work is being done from the point of view of the user processes . If the Operating System switches from one process to the next too fast, it may be more fair to various processes, but then the throughput may fall. Similarly, if the time slice is far much, the throughput will increase (assuming there are a sufficient number of processes waiting and which can make use of the time slice), but then. it may not be a very fair policy.

Let us briefly discuss the meaning of other objectives. CPU utilization is the fraction of the time that the CPU is busy on the average executing either the user processes or the Operating System. If tbe time slice is very small. the context switches will be more frequent. Hence, the CPU will be busy executing the Operating System instructions more than those of tbe user processes. Therefore, the throughput will be low, but the CPU utilization will be very high, as this objective does not care what

is being executed. and whether it is useful. The CPU utilization will be low only if the CPU remains idle.

(b) At any time, the main memory of the computer is limited and can bold only a certain number of processes. If the availability of the main memory becomes a great problem, and a process gets blocked. it may be worthwhile to swap it out on the disk and put it in yet another queue for a process state called swapped out and blocked which is different from a queue of only blocked processes. hence, requiring a separate PCB chain (we had not discussed this as one of the process states to reduce complications, but any Operating System has to provide for this).

The question tbat arises is as to what happens when the IJO is completed for such a process and if the process is swapped out? Where is the data requested by that process read in? The data required for that process is read in the memory buffer of the Operating System first. At this juncture, the Operating System moves the process to yet another process state called swapped out but ready slate. It is made ready because it is not waiting for any IJO any longer. This also is yet another process state which will require a separate PCB chain.

One option is to retain the data in the memory buffer of the Operating System and transfer itto the 110 area of the process after it gets swapped in. This requires a large memory buffer far the Operating System because the Operating SystCIDbas to define these buffers for every process as a similar situation could arise in the case of every process. Another option is to transfer tbe data to the disk in the process image at the exact location (e.g. 110 area), so that when the process is swapped in, it does so along with the data record in the proper place. After this, it can be sched-
uled eventually. This requires less memory but more 110 time.

When some memory gets freed. the Operating System looks at the list of swapped but ready processes, decides which one is tobe swapped in (depending upon priority, memory and other resources required, etc.) and after swapping it in, links that PCB in the chain of ready processes for dispatching. This is the function of the medium term scheduler as sbown in Fig. 5.27. It is obvious that this scheduler has to work in close conjunction with the long term scheduler. For instance. when some memory gets freed, there could be competition far it from the processes managed by these two scbedulers.

(c) The short term scheduler decides which of the ready processes is to be scheduled or dispatched next.

These three scheduling levels have to interact amongst themselves quite closely to ensure that the computing resources are managed optimally. The exact algorithms for these and the interaction between them are quite complex and are beyond the scope of this text. We will illustrate the scbeduling policies only for the short tern} scheduler in the subsequent section.

5.19.5 Scheduling Policies (For Short Term Scbeduling)

We will now discuss some of the commonly used scheduling policies-belonging to both pre-emptive and non-preemptive philosophies and using either a concept of priority or time slice or both. It should be fairly easy to relate these policies to the kind of PCB chains for ready processes that will be needed for implementing them.

Round Robin Policy

This is the simplest method which holds all the ready processes in one SIngle queue and dispatches them one by one. Each process is allocated a certain time slice. A context switch occurs only if the process consumes the full time slice (i.e. CPU bound jab doing a lot of calculations) or if it requests for *YO* during the time slice. If the process consumes the full time slice, the process state is changed from

211 may be quite unnecessary for the other processes. You need a policy whereby the Operating System can
increase the rime slice only for certain types of processes (which are CPU bound). But then, in order to be fair, it should allocate this larger time slice less frequently to it, i.e. it should reduce its priority. For an 110 bound processes, the actions should be just the reverse. Hence, if you want to be fair and also increase the throughput, for CPU bound processes, you need to:

- Increase the time slice, and
- Reduce the priority.

Therefore. this process should gel more time slice, but less frequently.

Similarly, for 110 bound processes, you need to:

- Decrease the time slice, and
- Increase the priority.

Hence, an IJO bound process should get less time slice each time but it should get it more frequently. The reason is that such a process cannot utilize a bigger time slice anyway!

Notice the fairness in the policy. For both CPU bound and I/O bound processes, the total time a.I.Iocated is more or less equitable if not exactly the same.

How should this policy be implemented? When the process is initialed, the Operating System does not know whether it is 110 bound, or CPU bound. What we need is a heuristic approach for the Operating System which will monitor the performance of the process in terms of the frequency of ItO calls *(1/0* boundness) and then change the priority and the time slice of that process accordingly. This is normally implemented using i\'luJtilevel Feedback Queues (MFQ) as shown in Fig. 5.31.

The scheme works as follows:

(a) The list of ready processes is split up into many queues with levels from 0 to *n* (in the figure shown, we have assumed ıt = 3). At each level, the PCBs are chained together as before.

(b) Each level corresponds to a value of time slice. For instance level 0 has time slice = '0' level I has time slice = *t,* and so on. *These* time slice values are stored by the Operating System. When it wants to dispatch a process belonging to a specific queue, it loads the corresponding value of the lime slice into the timer, so that there will be a 'time up' interrupt generated after ihat much time, as we have studied earlier.

This is organised in such a way ihat as you go down the level. i.e. from level 0 to level 3, the time slice increases, i.e. *t3* >'2>" > *to'* In practice if,o = *x* milliseconds, J could be *2(,1*$_2$ could
be *4x* and ~ could be 8x.

(c) As you go down the level, the priority decreases. This is implemented by having the scheduler search through the PCBs at level 0 first, ihen level 1. then level 2 and so on for choosing a process for dispatching. Hence, if a PCB is found in level 0, the scheduler will schedule it wiihout going to level I implying thereby that level 0 has higher priority than level 1. It will search for the queue at level I only if ihe level 0 queue is empty. The same philosophy applies to all the levels below. Hence, as we traverse from level 0 to level 3, the time slice increases and the priority decreases. After studying the pasi behaviour at the regular interval. now ihe kernel needs to somehow keep pushing the 110 bound processes at the upper levels and push the CPU bound processes to the lower levels. Let us now see how this is achieved.

(d) A new process always enters at level 0 and it is allocated a time slice l_0,

(e) If the process uses the time slice fully (i.e. if it is CPU bound), it is pushed to the lower level, thereby increasing the time slice but decreasing the priority. This is done for all levels, excepting if it is already at the lowest level in which case it is reintroduced at the end of the same (lowest) level only, because, obviously, it cannot be pushed any further.

- At any given time, the CPU can execute only one instruction belonging to only one of the programs residing in the memory. Therefore, the Operating System will have to allocate the CPU time to various users based on a certain policy. This is done by the Process Management (PM) module.
- A process can be defined as a program under execution. which competes for the CPU time and other resources.
- To reduce the idleness of CPU, concept ofmultiprogram.mlng is arisen. When one process waits for *110.* some other process can be presented to CPU to execute and vice versa
- Wben CPU turns its attention from one process to another, it is said to be context switching and usually some time is lost in the process.
- The number of processes running simultaneously and competing for the CPU is known as the degree of multiprogramming.
- If a process is interrupted. its memory contents (or pointers to them) and CPU register contents are saved before switching to other process so that restarting the process should be possible. These memory contents and CPU registers together called as context of the process.
- The context of the process is saved by the operating system in a specific memory area called Register Save Area. Each process has ilS separate Register Save Area.
- To manage the switching between the processes. the operating system defines three process states as Running. Ready, and Blocked.
- Each process is normally given certain time to run. This is known as time slice.
- The Operating System maintains the information about each process in a record or a data structure called Process Control Block (PCB).
- Operating System has to provide system calls or services for following operations on a process:
  (I) Create a Process
  (2) Kill a Process
  (3) Dispatch a Process
  (4) Change the priority of a Process
  (5) Block a Process
  (6) Time up a Process
  (7) Wake up a Process.
- Fairness refers to being fair to every user in terms of CPU time that he gets.
- Througbput refers to total productive work done by all the users put together.
- CPU utilization is the fraction of the time that the CPU is busy on the average executing either the user processes or the Operating System.
- Turnaround time is the elapsed time between the time a program or a job is submitted and the time when it is completed.
- Waiting time is the time a job spends waiting in the queue of the newly admitted processes for the Operating System to allocate resources to it, before commencing its execution.
- The response time means the time to respond with an answer or result to a question or an event.
- Priority of a process can be of two main types-Global or external priority and Local or Internal priority.
- Shortest Job First (SJF) is an algorithm to set the internal priority of a process in which the operat- ing system sets higher priority for shorter jobs.
- Some Operating Systems use concept of time sUet to schedule the processes instead of using the concept of priority.
- There are two scheduling philosophies either of which is used by any OS: Non-preemptive phUoso- phy and Preemptive pbilosophy.
- The OS has 3 types of schedulers one for eacb level of scheduling. They are Long term scheduler, Medium term scheduler and Short term scheduler.
- The different scheduling policies for dispatching a ready process are: Round Robin policy, Priority-based scheduling. Priority class and Heuristic scheduling.
- A thread can be defined as an asynchronous code path within a process.
- In Operating Systems which support multithreading, a process can consist of multiple threads. which can run concurrently.
- The OS maintains a Thread Control Block (TCB) for each thread within a process.

Inter Process Communication

6

This chapter describes the problems encountered in the Inter Process Communications (lPC) by taking an example of Producer-Consumer algorithms.
It discusses various sotutions that have been proposed 50 far, for mutual exclusion.
The chapter concludes with a detailed discussion on semaphores and classic problems in LPC.

.1 THE PRODtJCER-CONSUMER"pROBLEMS

In practice, several processes need to communicate with each other simultaneously. This requires proper synchronisation and use of shared data residing in shared memory locations. We will illustrate this by what is called a **Producer-Consumer problem.**

Let us assume that there are multiple users at different terminals running different processes but each one running the same program. This program prompts for a number from the user and on receiving it, deposits it in a shared variable at some common memory location. As these processes produce some data, they are called **Producer** processes. Let us also imagine that there is another process which picks up this
number as soon as any producer process outputs it and prints it. This process which uses or consumes the data produced by the producer process is called **Consumer** process. We can, therefore, see that the producer processes communicate with the consumer process through a sbared variable where the shared data is deposited (Fig. 6.1).

(i) Initially flag = 0
(ii) PA executes instruction P.O and falls through to P.I, as the flag = O.
(iii) PA sets flag to 1 by instruction P.l.
(iv) The time slice for PA is over and the processor is allocated to another Producer process .PB.
(v) PB keeps waiting at instruction P.O because flag is now = 1. This continues until its time slice also is over, without doing anything useful. Hence, even if the shared data item (i.e. the number in this
case) is empty, PB cannot output the number. This is clearly wasteful, though it may not be a serious problem, Let us proceed further.
(vi) A consumer process CA is now scheduled. It will fall through C.O because flag = 1. (It was set by PA in step (iii».
(vii) CA will set flag to 0 by executing instruction C.I.
(viii) CA will print the number by instruction C.2 before the producer has output it (may be the earlier number will get printed again!). This is certainly wrong!

Therefore, just preponing the setting of the flags does not work. What then is the solution?

Before going into the solution, let us understand the problem correctly. The portion in any program which accesses a shared resource (such as a shared variable in the memory) is called as Critical SeetloD or Critical Region. In our example, instructions P.I and P.2 of producer process or instructions C.I and C.2 of consumer process constitute the critical region. This is because, both the flag and the data item where the number is output by producer process are shared variables. The problem that we were facing was caused by what is called race condition. When two or more processes are reading or writing some shared data and the outcome is dependent upon which process runs precisely when, the situation can be called race condition. We were clearly facing this problem in our example. This is obviously undesir- able, because, the results are unpredictable. What we need is a highly accurate and predictable environ- ment. How can we avoid race conditions? .

A closer look will reveal that the race conditions arose because more than one process was in the critical region at the same time. One point must be remembered. A critical region here actually means a critical region of any program. Itdoes not have to be of the same program. In the first example (Fig. 6.4), the problem arose because both P A and PB were in the critical region of the same program at the same time. However. PA and PB were two producer processes running the same program. In the second exam- ple (Fig. 6.5), the problem arose even if processes PA and CA were running separate programs and both were in their respective critical regions simultaneously. This should be clear by going through our exam- pIe with both alternatives as in Figs 6.4 and 6.5. What is the solution to this problem then?

If we could guarantee that only one process is allowed to enter any critical region (i.e. of any process) at a given time, the problem of race condition will vanish. For instance, in any of the two cases depicted in Figs 6.4 and 6.5, when PA has executed instruction P.I and is timed out (i.e. without completing and getting out of its critical region), and if we find some mechanism to disallow any other process (producer or consumer) to enter its respective critical region, the problem will be solved. This is because no other producer process such as PA would be able to execute instructions P.I or P.2 and no other consumer process such as CA would be allowed to execute instructions C.I or C.2. After PA is scheduled again, only PA would then be allowed to complete the execution of the critical region. Until that happens, all the other processes wanting to enter their critical regions would keep waiting. When PA gets out of its critical region, one of the other processes can now enter its critical region; and that is just tine. Therefore, what we want is mutua.1 exclusion which could tum out to be a complex design exercise. We will outline the major issues involved in implementing this strategy in the next section.

competing for the CPU time, it would be useful. Allocating a time slice to a process which is going to waste it in 'busy waiting' anyway is quite unproductive. If we avoid this, the CPU would be free to be scheduled to other ready processes. The blocked consumer process would be made ready only after the flag status is changed. After this, the consumer process could continue and enter its critical region.

This is exactly what we had wanted. This effectively means that we are treating this flag operation as an *1/0* operation as far as process states are concerned, so that a process could be blocked not only while waiting for an *110* but also while waiting for the change in the flag. The problem is that none of the solutions, inc luding Peterson's, avoids busy Waiting. In fact, if this was put down as a sixth condition, in addition to the five conditions listed at the end of Sec. 6.1. to make the solution as an acceptable one. But none of these solutions would be acceptable, however brilliant it may be.

A way out of all this had to be found. In 1965, E. W. Dijkstra suggested a solution using a 'semaphore', which is widely used today. We will discuss some of these earlier solutions and their shortcom- ings in the following sections. At the end, we will discuss Dijkstra's solution.

6.2.S Alternating Policy

This was the first attempt to arrive at the mutual exclu- sion primitives. It is based on the assumption that there are only two processes: A and B, and the CPU strictly alternates between them. It first schedules A, then B, then again A, and so on. The algorithms for programs run by processes A and B are outlined below. We assume that the variable Process-Ilr contains the name of the process such as A or B. This is a shared variable between these processes and is initially set up by the Operating System for them. Figure 6.9 depicts this alternating policy.

```
    Begin
A.O     Whll.   Process-ID = "B"  do; /"Wait
A.1             Critical Region-A;
A.2             Set Process-ID-"B"
    End

                Proce ••
                    A

    Begin
B.O     While   Process-ID = .'''''  do; /"Wail
B.1             Critical Region-a;
B.2             Set Process·IO""'·
    End

                Proc ...    B
```

..... Fig. 6.9 Alternating policy

We can verify that mutual exclusion is guaranteed. A.O and A.2 are the instructions which encapsulate the critical region and therefore, functionally play the role of the primitives for mutual exclusion. This is true about instructions B.O and .B.2 also. Let us see how this works.

(i) Let us assume that initially Process-Ilz is set to "A" and Process A is scheduled. This is done by the Operating System.

(ii) Process A will execute instruction A.O and fall through to A.I, because Process-ID = "A".

(iii) Process A will execute the critical region and only then Process-ID is set to "8" at instruction A.2. Hence, even if the context switch takes place after A.O or even after A.I but before A.2, and if Process B is then scbeduled (remember we have assumed that there are only two processes I), Process 8 will continue to loop at instruction B.O and will not enter the critical region. This is because Process-Hi is still "A". Process B can enter its critical region only if Process-ID = "8". And this can happen only in instruction A.2 which in tum can happen only after Process A has executed its critical region in instruction A.1. This is clear from the program for Process A as given in Fig. 6.9.

(iii) Instruction 0 is executed and then through instruction I, Enter-Critical-Region is called.
(iv) Instruction EN.O is executed. Now ACC becomes "F', but INO becomes "N",
(v) Instruction EN. I recognises that ACC has the value "F' and therefore, prepares to go into critical region by executing EN .3, i.e. returning to the caller (in this case, instruction 2 of PA).
(vi) Let us assume that PA loses control to PB due to the context switch at this juncture, just before the critical region is actually entered. .
(vii) PB now executes instruction 0 of process PB.
(viii) PB executes instruction I and therefore, calls Enter-Critical-Region of process PB.
(ix) EN.O is executed for PB, ACC now becomes "N" (In step (iv),INO had become "N", which is now moved to ACC) and INO continues to hold the value "N".
(x) EN.I now is executed for PB, and because of unequal comparison. it loops back and therefore, docs not get out of the Enter-Critical-Region routine. Thus, PB cannot reach instruction 2 and enter its critical region. This is because PA is in.its critical region.
(xi) Eventually PA is again scheduled. It gets into the critical region (executes instruction 2 of the main program). While executing the critical region (instruction 2) for PA, if a context switch takes place again, and if PB is again scheduled, PB will srillioop because both IND and ACC still continue to be UN" from step (ix). (Busy Waiting!) Let us assume that PA completes instruction 2 and gets out of its critical region.
(xii) PA executes instruction 3 of the main program and calls the Exit-Critical-Region routine.
(xiii) EX.O is now executed for PA where IND becomes "P' again. PA gets out of instruction 3 of the main program by executing EX. I.
(xiv) Let us assume tbat now PB is again scheduled, where it executes EN.O once more. (It was looping in Enter-Critical-Region. remember") After EN.O, ACC becomes "P' (because INO had become "P' in step (xiii) and INO is moved to ACC) and IND is changed to UN" by the TSL instruction.
(xv) At EN. 1, it now finds that ACC is equal to "P' (due to step (xiv) and therefore, it goes to EN.3 and returns to the main program, i.e. instruction 2.
(xvi) PB now can enter its critical region. Thus. we can see that PB could enter its critical region only after PA came out of its critical region.

We leave it to the reader to ensure that this solution works in all cases. The algorithm shown in Fig. 6.12 does not specify whether it is for a producer process or a consumer process. It is valid for any process. In short, if any process which uses critical region is written in this fashion, race conditions can be avoided. The solution, however, is not without demerits. In the first place, it uses special hardware, and therefore, cannot be generalised to all the machines. Secondly, it also is based on the principle of busy waiting and therefore, is not the most efficient solution.

Finally, Dijkstra in 1965 found a new method using the concept of semaphores. It can tackle most of the problems mentioned above, depending upon its implementation. We will now study this method.

6.2.8 Semaphores

Concepts

Semaphores represent an abstraction of many important ideas in mutual exclusion. A semaphore is a protected variable which can be accessed and changed only by operations such as "DOWN" (or P) and "UP" (or V). It can be a Counting Semapbore or a General Semaphore. where it can take oo any positive value. Alternatively, it can be a Binary Semaphore which can take on the values of only 0 or 1. Semaphores can be implemented in software as well as hardware.

(xiv) It comes out of the "If" statement by "Endif" at
0.4. (xv) It enables interrupts again at 0.5.
(xvi) As PB is no more a "Running" process, let us assume that the scheduler schedules PC and dispatches it.
(x vii) PC will go IhIOUgh the same steps as PB, and its PCB will also gel added to the semaphore queue, because S is still equal to O. We know that S can become 1 only in the UP(S) operation, which takes place after the execution of the critical region portion of any process. We also know that only PA, wben rescheduled, can achieve this, since no other process can even enter it so long as PA has nor come Out of its critical region. AU wiU only continue getting added to the semapbore queue. Only UP(S) instruction can again set S to L; but UP(S) can get executed only after PA has come out of its critical region.
(xviii) Let. us assume that PA is rescheduled eventually and il resumes where il had left off last time at step (vii).
(xix) PA completes instruction 2 of Fig. 6.13, i.e. its critical
region. (xx) PA calls the UP(S) routine at instruction 3 of Fig.
6.13.
(xxi) UP(S) disables interrupts at U.O.
(xxii) [t increments S by I at U.l. Now S becomes 1.
(xxiii) It checks the semaphore queue at U.2 and finds that it is NOT empty.
(xxiv) It releases PB, i.e. moves it from the semaphore queue to the ready queue. PC still continues in the semaphore queue.
(xxv) It executes U.4 and U.5 and comes out of UP(S) after enabling the interrupts again.
(xxvi) PA starts executing instruction 4 of Fig. 6.13. Let us assume that during the execution of 4 (actually instruction 4 is a sci of instructions), PA's time slice gets over, and PO gets scheduled (May be PO had a higher priority than PB!).
(xxvii) PO executes instruction 0 of Fig. 6.1.3.
(xxviii) PO calls tbe "~OWN" routine at instruction I of Fig. 6.13.
(xxix) The "DOWN" routine goes through the instructions exactly as discussed in steps (iii), (iv) and (v). It wiU decrement S to 0 and allow PO 10 enter its critical region.
(xxx) Let us assume that PO finishes instruction 2 of Fig. 6.13, i.e. critical region.
(xxxi) Also assume that the time slice for PO is over after it has finished instruction 2 of Fig. 6.13 (i.e. critical region) but before it executes instruction 3 of Fig. 6.13 (i.e, UP(S) call).
(xxxii) Let us assume that PB gets scheduled. (It can now be scheduled, because, it was made "Ready" in step (xxivj).
(xxxiii) PB executes instructions 0 and I of Fig. 6.13. Because S = 0, PB is added to the semaphore queue agam.
(xxxiv) Let us assume that PO is scheduled again, and it completes the UP(S) operation to set S to L. Also, because the semaphore queue is NOT empty, it will release PC into the ready
queue. (xxxv) PC can now be scheduled.

This procedure is repeated for all the processes until all are over. Two interesting points emerge out of this discussion.

(a) When a process enters a critical region., and a process switch takes place beforeit completes the UP(S) instruction, what is the use of scheduling any other process? What is the purpose of picking up another process from the ready queue and moving it to the semaphore queue, when, at a later

```
1.  Main()                          r This is the main lasko!  every philosopher"

    Repeat continuously Think
                        ();         r Philosopher  is thinking for some time "
                        Take_forks  r Acquire both the forks, or wail ~ Ihis is not possible "
                        (); Eat (); I' Eat spaghetti "
                        PuL!orks  (
                        );          t: Put down both the forkS on the table "
```

End

2. Take_forkS ()

```
DOWN (mutex);                  r Enter critical region (obtain e.xclusive access to State) "
State = HungI}';               r Signify that the philosopher is hungry .,
Test                           t: Try to acquire both the forkS"
(Philosopher);                 r Exit critical region (release exclusive access to
UP (mutex);                    State) '/ I' Block if forkS could not be acquired "
DOWN
(Philosopher);
```

3. Put_forkS ( )

```
DOWN                           I' Enter Ctilical region (obtain axclusive access to
                               State) " I' Signify that the philosopher has finished
                               eating"
(mutex);                       r Check H the left neighbor can noweal"
State = Thinking;              I' Check if Ihe right neighbor can now eat '/
Test (LEFT);                   r Exil critical region (obtain exclusive access 10 State) "
Test
(RIGHn; UP
(mutex);
```

4. Tesl ( )

```
If the philosopher Is hungry and both his neighbors are
not eating. Then sellhe  Stale of this philosopher =
Eating;
UP (Philosopher);                  I' Philosopher can be tested by other philosophers '/
```

~",..;> Fig, 6.19 Solution to the dining philosophers problem

In which practical situations would the dining philosophers problem apply? Clearly, it is useful when the number of resources (such as 110 devices) is limited, and many processes compete for exclusive access over those resources. This problem is unlikely in the case of a database access, for which the next problem is applicable.

2, The Readers and Writers Problem

Imagine a large database containing thousands of records. Assume that many programs (or processes) need to read from and write to the database at the same time. In such situations, it is quite likely that two or more processes make an attempt to write to the database at the same time, Even if we manage to take care of this, while a process is writing to the database, no other process must be allowed to read from the database to avoid concurrency problems. But we must allow many processes to read from the database at the same time.

(i) A proposed solution tackles these issues by assigning higher priorities to the reader processes, as compared to the writer processes.

(ii) When the first reader process accesses the database, it performs a DOWN on the database. This prevents any writing process from accessing the database.

(iii) While this reader is reading the database. if another reader arrives, that reader simply increments a counter RC, which indicates bow many readers are currently active.

```
1.  Main ()

    Repeat continooously
                    Think ();
                    Take_lorks 0;
                    Eat 0;
                    PuLlorks ( );

r This Is the main task 01every philosopher"


r Philosopher is thinkIng for some time"
r Acquire both the lorks. or watt if this is not possible"
r Eat spaghetti "
t: Put down both the lorks on the table "
```

```
    End
2.  Take_forks  ()
    State =
    Hungry.
    Test (Philosopher);
    IIState  not
    Eatinglhen
         w8It(Philosophe
         r);                    r Slgn~y that tl1e philosopher  is hungry  "
                                t: Try to acquire  both the lorks  "
                                r  Wait lor other philosopherto   release  lorks  "
                                r Signify that the philosopher  has finiShed eating .,
3.  Put_forks  ()               t: Check  il the left neighbor  can now  eat "
    State  =                    r  Check  If the right neighbor  can noweat"
    Thinking;
    Test(LEFT    );
    Test  (RIGHT);
4.  Test  ()
    IIthe philosopher  is hungry  and both his neighbors  are not eating
        Then  set the Stale of this philosopher  = Eating;
        signal                  r Wake up other philosophers  waiting  to eat',
        (Philosopher);
```

....., Fig, 6,22 Solution to dining philosophers problem using monitors

Of course, one argument in the favour of semaphores as against the monitors is that semaphores, by virtue of their basic low-level interface, provide more granular control. This is always true even in the case of assembly language, which provides a fine-grained control to the application programmer, as compared to a high-level language. However, if the programmer does not want to use such low-level features, monitor is a better choice.

6,3,3 Message Passing

The need for message passing came about because the techniques such as semaphores and monitors work fine in the case of local scope. In other words, as long as the processes are local (i .e. on the same CPU), these techniques work perfectly. However, they are not intended to serve the needs of processes, which are on physically different machines. Such processes, which communicate over II network, need some mechanism to perform communication with each other. and yet be able to ensure concurrency. Message passing is the solution to this problem.

Using the technique of message passing, one process (sender) can safely send a message to another process (destination). without worrying if the message would reach the destination process. This is conceptually similar to the technology of Remote Procedure Calls (RPC), the difference being that mes- sage passing is an Operating System concept. whereas RPC is a data communications concept.

In message passing, two primitives are generally used: *send* and *receive.* The sender uses the *send* call to send a message. The receiver uses the *receive* call. These two calls take the following form;

```
send (destination,  &message);
receive  (source,  &message);
```

?IO Process in which programs running at the same time waits for some event to happen without performing any useful task is called

(a) Starvation
(b) Mutual exclusion
(c) Busy waiting
(d) IPC

Test Questions

6.1 What is meant by race conditions? How do they occur?

6.2 Give an example of Producer-consumer problem. indicating the reasons for inconsistency that can arise due to race conditions.

6.3 What is meant by the terms critical region and mutual exclusion?

— List fi ve conditions that can make the solution to the problem of race conditions acceptable.

6.4 \Vhy cannot a process disable the interrupts to avoid context switch while in the critical region to

6.5 avoid the race conditions?

Discuss the concept of mutual exclusion primitives and different ways of implementing them.

6.6 What is meant by Busy waiting? What is wrong with it?

6.7 Describe the importance and the relevance of Test and Set Lock (TSL) instruction for mutual

6.8 exclusion.

Give the algorithm for Alternating policy. Discuss its merits and demerits.

6.9 Discuss Peterson's algorithm with its merits and demerits.

6.10 Give the algorithm for implementing exclusion primitives using the hardware TSL instruction.

6.11 Does it avoid busy waiting?

\What are semaphores? How Co.t}hey implement mutual exclusion? What are different types

6.12 of semaphores? How can they be implemented?

Describe how semaphores can be used for block/wakeup synchronization between processes.

6.13 Discuss anyone of the classical JPC problems.

6.14

processes are waiting for. In order to detect a deadlock, the Operating System can give some imaginary coordinates to the nodes, Rand P. Depending upon the relationships between resources and processes (i.e. directions of the arrows), it can keep traversing, each time checking ifit has returned to a node it has already travelled by, to detect the incidence of a deadlock.

What does the Operating System do if it finds a deadlock? The only way out is to kill one of the processes so that the cycle is broken. Many large mainframe computers use this strategy. Some systems do not go through the overhead of constructing a DRAG. They monitor the performance of all the processes. Ifnone finishes for a very long time, the Operating System kills one of'the processes. This is a crude but quicker way to get around the problem.

What causes a deadlock? Coffman, Elphick and Shoshani in 1971 have shown that there are four conditions all of which must be satisfied for a deadlock: to take place. These conditions are given below.

(i) Mutual Exclusion Condition

Resources must be allocated to processes at any time in an exclusive manner and not on a shared basis for a deadlock to be possible. For instance, a disk drive can be shared by two processes simultaneously. This will not cause a deadlock. But printers, tape drives, plotters, etc. have to be allocated to a process in an
exclusive manner until the process completely finishes its work with it (which normally happens when the process ends). This is the cause of the trouble.

(ii) Wait for Condition

Even if a process holds certain resources at any moment, it should be possible for it to request for new ones. It should not have to give up the already held resources to be able to request for new ones: If this is not true, a deadlock can never take place.

(iii) No Preemption Condition

If a process holds certain resources, no other process should be able to take them away from it forcibly. Only the process holding them should be able to release them explicitly.

(iv) Circular Wait Conditio"

Processes (P I, P2, ...) and Resources (RI, R2, ...) should form a circular list as expressed in the form of a graph (DRAG). In short, there must be a circular (logically, and not in terms of the shape) chain of multiple resources and multiple processes forming a closed loop as discussed earlier.

It is nece-ssary to understand that all these four conditions have to be satisfied simultaneously tor the existence ofa deadlock. If any one of them does not exist, a deadlock can be avoided.

Various strategies have been followed by different Operating Systems to deal with the problem of a deadlock. These are listed below .

- Ignore it .
- Detect it

cated (if he did, a new resource type should have to be created). The Operating System then goes through the resourcewise table to see if there is any free resource of that type, and if there is any, allocates it to the process. After this, it updates both these tables appropriately.

If no free resource of that type is available, the Operating System keeps that process waiting on one of the resources for that type. (For instance, it could add the process to the waiting queue for a resource, where the wait list is the shortest). This also will necessitate updating of both tables. When a process releases a resource, again both tables will be updated accordingly.

(v) At any time, the Operating System can use these tables to detect a circular wait or a deadlock. Typically, whenever a resource is demanded by a process, before actually allocating it, the Operating System could use this algorithm 10 see whether the allocation can potentially lead to a deadlock or not.

It should be noted that this is by no means the most efficient algorithm of deadlock detection. Modem research has C{lIDeout with a number of ingenious ideas, which are being discussed and debated. Some of tbese are implemented too! What we present here is a simplified, accurate (though a little inefficient) method to clarify the concepts. The algorithm would simulate the traversal along the DR.AG to detect if the same node is reached, i.e. the circular wait.

The working is as follows:

(a) Go through the resourcewise table entries one by one, each time storing the values processed. This is useful in detecting a circular wait, i.e. in finding out whether we have reached the same node or not.

(b) Ignore entries for free resources (such as an entry for ROO.inFig. 7.6).

(c) For all other entries, access the process to which the re.source is allocated (e.g. resource ROI is allocated to process PI in Fig. 7.6). In this case, store the numbers ROI and PI in separate lists called resource list and process list respectively.

(d) Access the entry in the processwise table (Fig. 7.7) for that process (P I in this case).

(e) Access one by one the resources this process (PI) is waiting for. For example, PI is waiting for resource R20. Check if this is the same as the one already encountered, i.e. ifR20 is the same as ROI stored in step (c). 10 short, check ifcircular wait is already encountered. lfyes, the deadlock is detected. If'no, store this resource (e.g, R20) in the resource list. This list will now containROl and R20. The process list still contains only PI. Check from Fig. 7.7 whether there is any other resource apart from R20, that process P I is waiting for. If there is any, this procedure will have to be repeated. In this example, there is no such resource. Therefore, the Operating System goes to the next step.

(f) Go to the entry in the resourccwise table (Fig. 7.6) for the next resource in the resource list after ROI. This is resource R20, in this case. We find that R20 is allocated to P5.

(g) Check if this process (i.e. PS) is the one already encountered in the process list (e.g. ifPS is the same as PI). If it is the same, a deadlock is confirmed. In this case, PS is not the same as PI. So only store PS after PI in the process list and proceed. Tbe process list now contains Pl and PS. The resource list is still ROI, RlO as in step (e). After this, the Operating System will have to choose RIO and R23 as they are the resources process P5 is waiting for. It finds that RIO is allocated to PI. And PI already existed in the process list. Hence, a deadlock (PI ~ R20 ~ P5 ~ RIO ~ Pt) has been detected.

Therefore, the Operating System will have to maintain two lists-one list of resources already encountered and a second list of all the waiting processes already encountered. Any time the

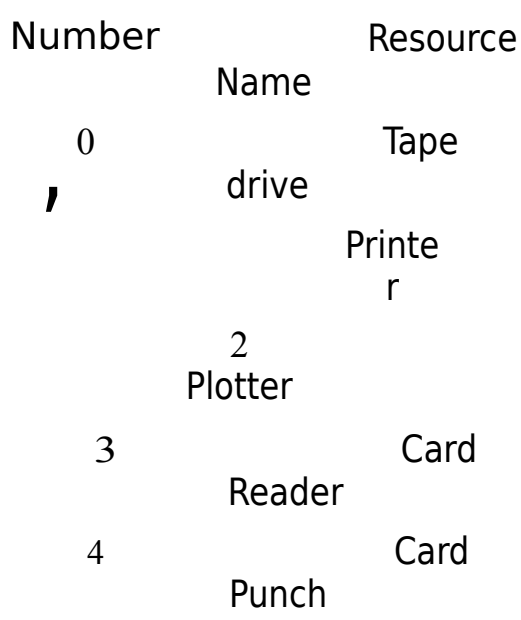
definitely be an unacceptable situation due to the problems of mounting/dismounting, positioning, and so on. With printers, the situation is worse.

(iv) Circular Wait Condition

It is obvious that attacking the first three conditions is very difficult. Only the last one remains. If the circular wait condition is prevented, the problem of the deadlock can be prevented too.

One way in which this can be achieved is to force a process to hold only one resource at a time. If it requires another resource, it must first give up the one that is held by it and then request for another. This obviously has the same flaws as discussed above while preventing condition (iii). If a process P I holds R I and wants R2, it must give up R I first, because another process P2 should be able to get it (RI). We are again faced with a problem of assigning a tape drive to P2 after P I has processed only half the records. This, therefore, is also an unacceptable solution.

There is a better solution to the problem, in which all resources are numbered as shown in Fig. 7.10.



"~ Fig. 7.10 Numbering of resources

A simple rule can tackle the circular wait condition now. Any process has to request for all the required resources in a numerically ascending order during its execution, assuming again that grabbing all the required resources at the beginning is not an acceptable solution. For instance, if a process PI requires a printer and a plotter at some time during its execution, it has to request for a printer first and then only for a plotter, because I < 2.

This would prevent a deadlock. Let us see how, Let us assume that two processes PI and P2 are wanting a tape drive and a plotter each. A deadlock can take place only if P I holds the tape drive and
wants the plotter, whereas P2 holds the plotter and requests for the tape drive, i.e. if the order in which the resources are requested by the two processes is exactly opposite. And this contradicts our assumption. Because 0 < 2, a tape drive has to be requested for before a plotter by each process, whether it is PI or P2. Therefore, it is impossible to get a situation that will lead to the deadlock.

What holds true for two processes also is true for multiple processes. However, there are some minor and major problems with this scheme also.

Imagine that there are two tape drives Tl and T2 and two processes PI and P2 in the system. If PI holds TI and requests for T2 whereas P2 holds T2 and requests for Tl, the deadlock can occur. What numbering scheme should then be followed as both are tape drives? Giving both tape drives the same number (e.g, 0) and aUowing a request for a resource with a number equal to or greater than that oftbe previous request, a deadlock can still occur as shown above.

This minor problem however could be solved by following a certain coding scheme in numbering

the resources. The first digit denotes the resources type and the second digit denotes the resource number

(1) For each request for any resource by a process, the Operating System goes through all these trial or imaginary allocations and updations, and if it finds that after the trial allocation, the state of the system would be 'safe', it actually goes ahead and makes an allocation after which it updates various matrices and tables in a real sense. The Operating System may need to maintain two sets of matrices fOTthis purpose. Any time, before an allocation, it could copy the first set of matrices (the real one) into the other, carry out aU trial allocations and updations in the other, and if the safe state results, update the former set with the allocations.

Two examples to understand this algorithm clearly are presented here.

EXAMPLE 7., Suppose process P I requests for I tape drive when the resources allocated to various processes are given by Fig. 7.11. The Operating System has to decide whether to grant this request or not. The Banker's algorithm proceeds to determine this as follows:

- If a tape drive is allocated to PI, F will become 011 and the resources still required for PI in Matrix B will become 010. After this, the free resources arc such that only process PI can com- plete, because each digit in F, i.e, Oil is equal to or more than the individual digits in the row fOTrequired resources for PI in Matrix B, i.e. 010. Therefore, hypothetically, ifno other process demands anything in between, the free resources can satisfy PI's demands and lead it to comple- tion.
- If PI is given all the resources it needs to complete, the row for assigned resources to PI in Matrix A will become 120, and after this allocation, F will become 00 I.
- At the end of the execution ofP I, aU the resources used by P I will become free and F will become 120 + 00 I = 121. We can now erase the rows for PI from both the matrices, indicating that, this is how the matrices will look if PI is granted its first request of'a tape drive and then is allowed to go 10 completion.
- We repeat the same steps with the other rows. For instance, now F = 121. Therefore, the Operat- ing System will have sufficient resources to complete either PO or P3 but not P2. This is because P2 requires 2 tape drives to complete, but the Operating System at this imaginary juncture bas only I. Let us say, the Operating System decides to allocate the resources to PO (it does not matter which one is chosen). Assuming that all the required resources are allocated to PO one by one, the row for assigned resources to PO in Matrix A will become 300 and that in Matrix B will obviously become 000. Fat this juncture will have become 121-100 = 021. If PO is now allowed to go to completion, all the resources held by PO will be returned to F. Now, we can erase the rows fOTPO from both the matrices. F would now become 300 + 021 = 321.
- Now either P2 or P3 can be chosen for this 'trial allocation'. Let us assume that P3 is allocated. Going by the same logic and steps, we know that resources required by P3 are III. Therefore, after the trial allocation, F will become *321* - III = 210, and resources assigned to P3 in Matrix A would become 212. When P3 completes and returns the resources to F, F will become 212 + 210 = 422.
- At the end, P2 will be allocated and completed. At this juncture, resources allocated to P2 will be 332, and F would be 442 - 211 = 211. In the end, all the resources will be returned to the free pool. At this juncture, F will become 332 + 211 = 543. This is the same as the total resources vector T that are known to the system. This is as expected, because after these imaginary alloca- tions and process completions, F should become equal to the total resources known to the system.

- ThC Operating System does aU these virtual or imaginary calculations before granting the first request of process P I for a tape drive. All it ensures is that if this request is granted, it is possible

Memory Management

8

ChcqJw ObjecrtNe4T 1 ----------

This chapter discusses various contiguous and non-contiguous memory allocation schemes.

Contiguous allocation schemes consist of Single Contiguous Memory Management Partitioned Scheme, Fixed Memory Management Scheme and Variable Partitioned Memory Management Scheme.

Non-contiguous schemes consist of Paging, Segmentation. Combined Schemes and Virtual Memory Management Systems.

Por all these schemes, it states the support tbat the Operating System expects from the hardware and then goes on to explain in detail the way the scheme is implemented.

It discusses various page replacement policies and compares their performances. It also discusses fIFO anomaly.

>Ne will now discuss the last portion of the Operating System, viz. the functions of **Memory**
~aDagemeDt (MM). As mentioned earlier, the topic discussed in this chapter assumes special mportance when a number of users share the same memory. In general, this module performs the allowing functions:

(a) To keep track of all memory locations-free or allocated and if allocated, to which process and how much.
(b) To decide the memory allocation policy, i.e. which process should get how much memory, when and where.
(c) To use various techniques and algorithms to allocate and deallocate memory locations.

Normally this is achieved with the help of some special hardware.

There is a variety of memory management systems. Figure 8.1 lists them.

Because it contains an address, it is as big as MAR. For every memory reference, when the final resultant address (after taking into account the addressing modes such as indirect, indexed, PC relative, and so on)' is in MAR, it is compared with the fence register by the hardware itself, and the hardware can detect "any protection violations. (For instance, in Fig. 8.3, if a user process with mode bit = 1 makes a reference to an address within the area for the Operating System which is less than
or equal to P, the hardware itself will detect it.)

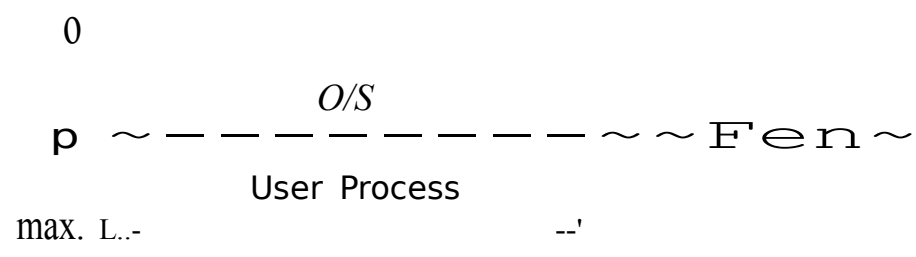


Fig. 8.3 Fence register

Sharing of code and data in memory does not make much sense in this scheme and is usually not supported.

Evaluation

This method does not have a large wasted memory (it cannot be used even ifit were large anywayl). This scheme has very fast access times (no Address Translation is required) and very little time-complexity. But its use is very limited due to the lack of multiuser facility.

8.3.1 Intreduction

Most operating systems such as *OS/360* running on ffiM hardware used the Fixed Partitioned Memory Management method. In this scheme, the main memory was divided into various sections called , partitions'. These partitions could be of different sizes, but once decided at the time of system generation, they could not be changed. This method could be used with swapping and relocation or without them.

In this method, the partitions are fixed at the time of system generation. (System generation is a process of tailoring the Operating System to specific requirements. The Operating System consists of a number of routines for supporting a variety of hardware items and devices, all of which may not be necessary for every user. Each user can select the necessary routines depending upon the devices to be used. This selection is made at the time of system generation.) At this time, the system manager has to declare the partition size.

To change the partitions, the operations have to be stopped and the Operating System has to be generated (i.e. loaded and created) with different partition specifications. That is the reason why, these partitions are also called 'static pardtiens'. On declaring static partitions, the Operating System creates a Partition Description Table (PDT) for future use. This table is shown in Fig. 8.4. Initially all the entries are marked as "FREE". However, as and when 8 process is loaded into one of the partitions, the status entry for that partition is changed to "ALLOCATED". Figure 8.4 shows the static partitions and their corresponding PDT at a given time.

If the Operating System had a "lookahead intelligence" feature, it could have possibly known that the partition with size = 2K is likely to get free soon. In this case, choosing the next process of SK for loading in the partition of size = 5K could bave been a better decision. Almost immediately after this, the 2K partition would get free to accommodate the 2K process with higher priority than the one with size = SK.

The highest priority process with size = 7K will have to wait until the partition with size = 8K gets (Tee. There is no alternative to this. This kind of intelligence is not always possible and it is also quite expensive.

If the Operating System chooses a simple but relatively less intelligent solution and loads the process with size =: 2K in the partition with size =: SK, the process with size = SK keeps waiting. After a while, even if the 2K partition gets free, it cannot be used, thus causing memory wastage, This is called external fragmentation. Contrast this with internal fragmentation in which there is a memory wastage within the partition itself. Imagine a partition of2K executing a process of l.SK size. The O.SK of the memory of the partition cannot be utilized. This wastage is due to internal fragmentation.

This discussion shows that the MM and the PM modules are interdependent and that they have to cooperate with each other.

8.3.3 Swapping

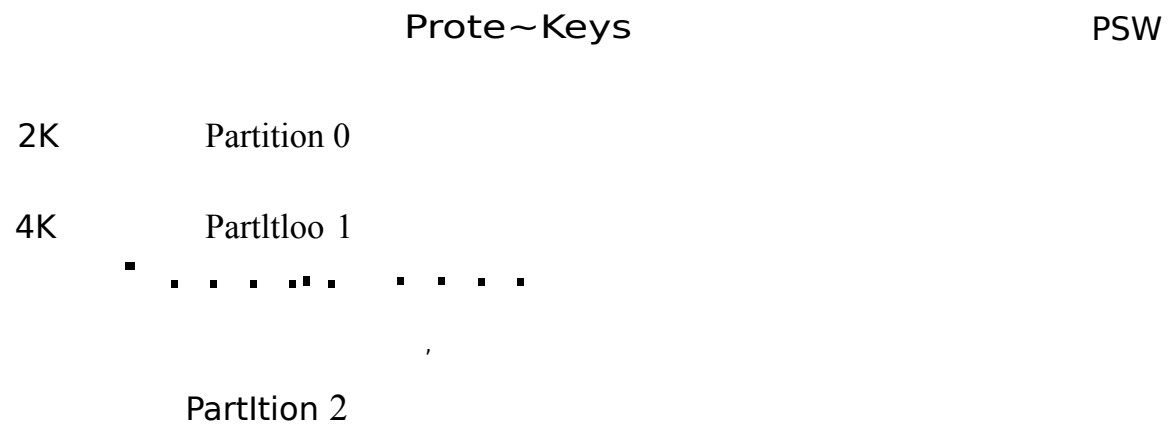
One more way in.which the partitioned memory management scheme is categorized is based on whether it supports swapping or not. Lifting the program from the memory and placing it on the disk is called swapping out. To bring the program again from the disk into the main memory is called swapping in. Normally, a blocked process is swapped out to make a room for a ready process to improve the CPU utilization. If more than one process is blocked, the swapper chooses a process with the lowest priority, or a process waiting for a slow I/O event for swapping out. As discussed earlier, a running process also can be swapped out (in priority-based preemptive scheduling).

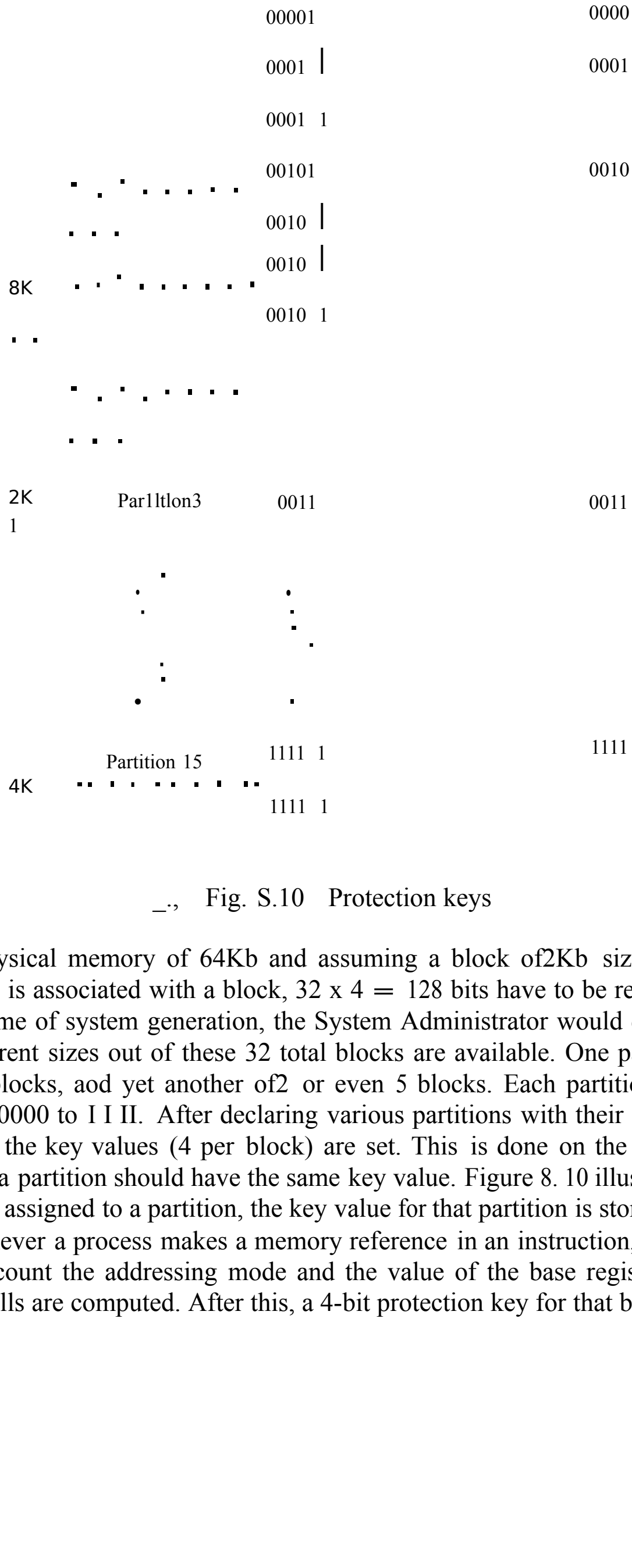
Swapping algorithm has to coordinate amongst Information, Process and Memory Management systems. If the Operating System completes the I/O on behalf of a blocked process which was swapped out, it keeps the data read recently in its own buffer. When the process is swapped in again, the Operating System moves the data into the 1/0 area of the process and then makes the process 'ready'. In demand paging, some portion of the memory where the record is to be read can be 'locked' or 'bound' to the main memory. The remaining portion can be swapped out if necessary. In this case, even if the process is
'blocked' and 'swapped out', the IJO can directly take place in theAP's memory. This is not possible in the scheme of 'fixed partition' because, in this case, the entire process image has to be in the memory or swapped out oo the disk.

The Operating System bas to find a place on the disk for the 'swapped out' process image. There are two alternatives. One is to create a separate swap file for each process. This method is very flexible, but can be very inefficient due to the increased number of files and directory entries thereby deteriorating the search times for any *1/0* operation. The other alternative is to keep a common swap file on the disk and note the location of each 'swapped' out process image within that file. In this scheme, an estimate of the swap file size has to be made initially. If a smaller area is reserved for this file, the Operating System may not be able to swap out processes beyond a certain limit, thus affecting performance. The medium term scheduler has to take this into account.

Regardless of the method, it must be remembered that the disk area reserved for swapping has to be larger in this scheme than in demand paging because, the entire process image has to be swapped out, even if only the "Data Division" area undergoes a change after the process is loaded ill the memory.

All the blocks associated with a partition allocated to a process are given the same key value in this scheme. If the number of partitions is 16, there can be maximum 16 user processes at any moment in the main memory. Therefore, a 4-bit key value ranging from 0000 to 1111 serves the purpose of identifying the owner of each block in the main memory. This is shown in Fig. 8.10.





_., Fig. S.10 Protection keys

Considering a physical memory of 64Kb and assuming a block of2Kb size, there would be 32 blocks. If a 4-bit key is associated with a block, 32 x 4 = 128 bits have to be reserved for storing the key values. At the time of system generation, the System Administrator would define a maximum of 16 partitions of different sizes out of these 32 total blocks are available. One partition could be of 1 block, another of3 blocks, aod yet another of2 or even 5 blocks. Each partition is then assigned a protection key from 0000 to I I II. After declaring various partitions with their different sizes, all the 128 bits reserved for the key values (4 per block) are set. This is done on the principle that aU the blocks belonging to a partition should have the same key value. Figure 8. 10 illustrates this.

When a process is assigned to a partition, the key value for that partition is stored in Program Status Word (PSW). Whenever a process makes a memory reference in an instruction, the resulting address (after taking into account the addressing mode and the value of the base register) and the block in which that address falls are computed. After this, a 4-bit protection key for that block is extracted from the 128-

The shaded area in Fig. 8,,12 shows the free area at any time. Notice that the numbers and sizes of processes are not predetermined. It starts ..v..ith only two partitions (Operating System and the other) and at stage (vi), they are 6 partitions. These partitions are created by the Operating System at the run time, and they differ in sizes.

The procedure to be followed for memory allocation is the same as described for fixed partitions in steps (~(vii) of Sec. 8.3.1, excepting that the algorithms and data structures used may vary. We will not repeat these steps here. An interested reader can go through that section to refresh the memory.

8.4.2 AUocation Algorithms

The basic information needed to allocate/deallocate is the same as given in the PDT in Fig. 8.4. However, because the number of entries is uncertain, it is rarely maintained as a table, due to the obvious difficulties of shifting all the subsequent entries after inserting any new entries. (Try doing that for our example in the previous section shown in Fig. 8.12.)

Therefore, the same information is normally kept as bitmaps or linked lists much in the same way that you keep track offree disk blocks. To do this, like a block on the disk, the Operating System defines a chunk of memory (often called a block again). This chunk could be I word or 2Kb, or 4Kb or whatever. The point is that for each process, allocation is made in multiples of this chunk. In a bit-map method, the Operating System maintains I bit for each such chunk denoting ifit is allocated (I) or free (=0). Hence,

if the chunk is a word of 32 bits, I bit per 32 bits means about 3.1 % of memory overhead, which is p.retty high. However, the memory wastage is minimal. This is because, the average wastage, as we know, is (chunk size--l)/2 per process. **If**the chunk size is high, the overhead is low but the wasted memory due to internal fragmentation is high. In a linked list, we create a record for each variable partition. Each record maintains information such as:

Allocated/free (F = Free, A =
Allocated) Starting chunk number
Number of chunks
Pointer (i.e. the chunk number) to the next entry.

Figure 8.13 depicts a picture of the memory at a given time. Corresponding to this state, we also show in band c, a bit-map and a linked list, where a shaded area denotes a free chunk. You will notice that the corresponding bit in the bit map is 0. The figure shows 29 chunks ofmemory-(Q-28), of which 17 are allocated to 6 processes.

As is clear, the bit-map shows that the first 4 chunks are free, then the next 3 are allocated, then, again, the next 2 are free, and so on. We can ensure that the linked list also depicts the same picture, essentially, any of these two methods can be used by the Operating System. Each has merits and demerits.

In this scheme, when a chunk is allocated to a process or a process terminates, thereby freeing a number of chunks, the bit map or the linked list is updated accordingly to reflect these changes.

8.4.3 Swapping

The swapping considerations are almost identical to those discussed in Sec. 8.3.3 for fixed partitions, and therefore, need no further discussion.

8.4.4 Relocation and Address Translation

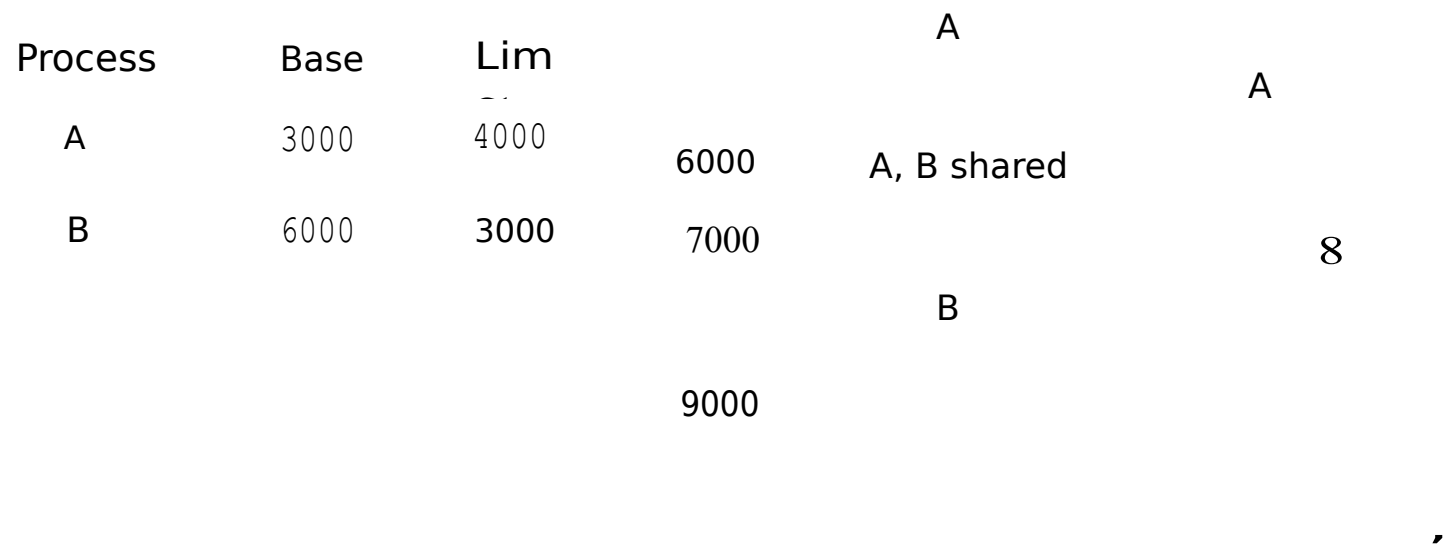
This is substantially the same as in fixed partitions. This scheme also depends upon the base register which is saved-in and restored and from the PCB at the context switch, The physical address is calculated by adding the base register to the virtual address as before, and the resulting address goes to MAR for decoding. After swapping or compaction operations, if the processes change their memory locations, these values also need to be changed as discussed earlier. .

8.4.S Protection and Sharing

Protection is achieved with the help of the limit register. Before calculating the resultant physical address, the virtual address is checked to ensure that it is equal to or less than the limit register. This register is loaded from the PCB when that process is dispatched for execution. As this value of limit register does not undergo any change during the execution of a process, it does not need to be saved back in the PCB at the context switch. "

Sharing is possible only to a limited extent by using overlapping partitions as shown in Fig. 8.15.

300
0



m. Fig. 8.1S Overlapping partitions

The figure depicts that process A occupies locations with addresses 3<>00-6999, and process B occupies locations with addresses 6000- 8999. Thus, in this case, locations with addresses between 6000 and 6999 are overlapping, as they belong to both the partitions. This is possible only because the partitions were variable and not fixed.

0= 0, address 101 would correspond to P= 1and 0 = I, and so on. Therefore, address 107 would be that of the location number = 7 in page number I. Therefore, P = 0 I, 0 = 000 III in binary, if we reserve 2 bits for P and 6 for O.lfwc concatenatethe two, we get the two-dimensional address as 01000111, as against a one-dimensional address of 0 II 0 I0 II.

Notice that these two are different. The address translation at the tune of execution will pose difficulties, if the compiler produces a one-dimensional address which has no correlation with a two-dimensional one.

This problem can be easily solved if the page size is a power of2, which is normally the case. Assume in this case that the page size is 32. Thus, locations ~31 are in page 0,32-63 in page 1,64-95 in page 2 and 96-127 in page 3. Therefore, location 96 means P = 3 and 0 = 0, location 97 means P = 3 and D = I. We can, therefore, easily see tbat the address 107 will mean P = 3 and 0 = 11. Hence, the two-dimensional address for decimal 107 in binary is P = 0 II and O = 0 I0 II. If we concatenate tbe two, we will get 0 II0 I0 II, which is exactly sante as the one-dimensional address in binary that the compiler produces.

An interesting point is worth noting. Even if'the page size were 64 instead of32, the two-dimensional address would remain the same. in this case, page 0 would have addresses 0-63 and page I would have 64-127. Hence, location 107 would mean location 43 in page I. Therefore, a two-dimensional address for 107 would be page (P) = 0 I and displacement (0) = 10I 0 11 in binary. Concatenating the two, we still get 0 II 010 II which is the same as the one-dimensional address in binary address that the compiler produces.

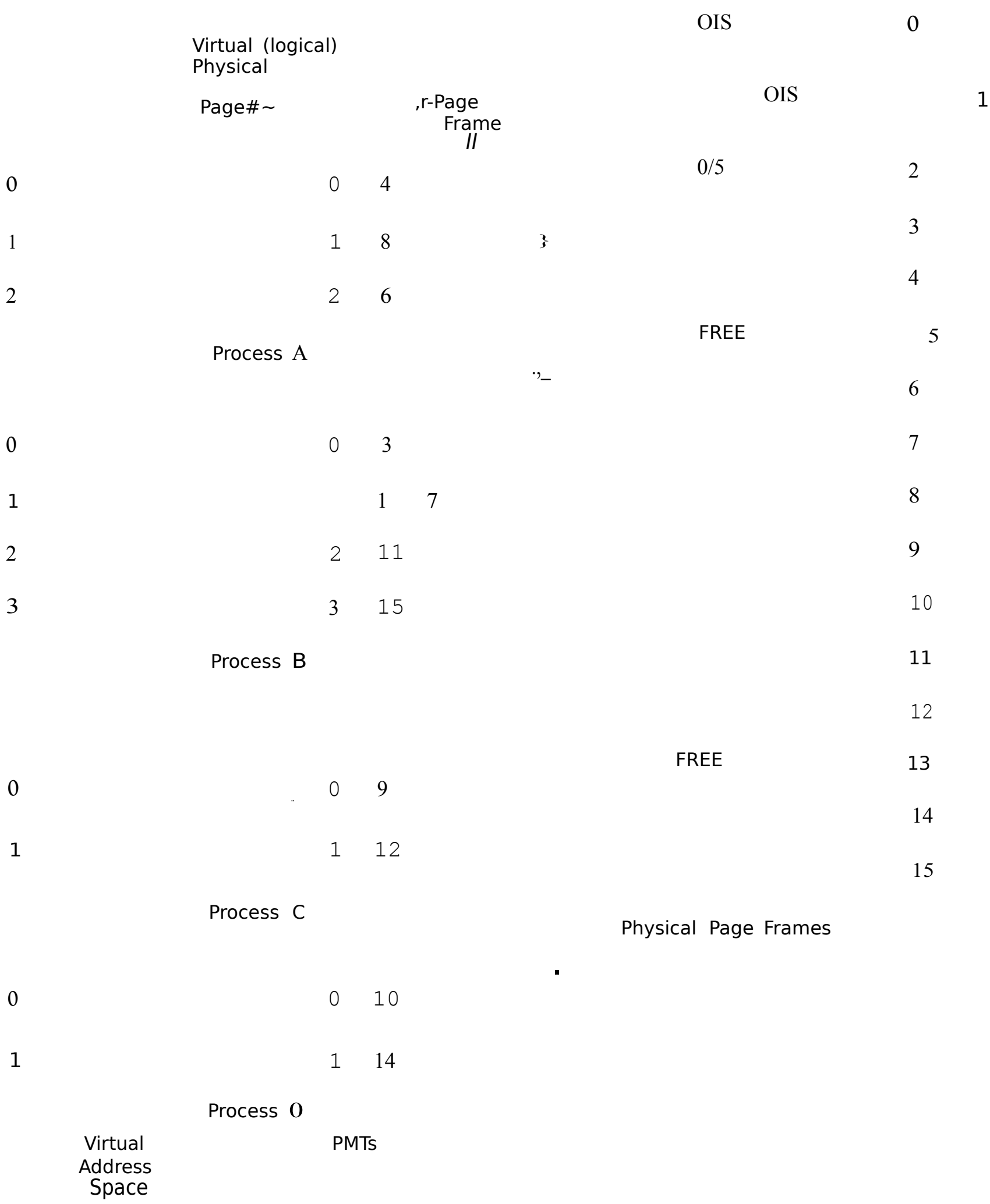
Therefore, the compiler does not have to produce different addresses, specifically because it is going to be treated as a two-dimensional address. The compiler compiles addresses, as if they were absolute addresses with respect to 0 as the starting address. These are the same as single-dimensional addresses. At the time of execution, the addresses can be separated as page number (P) quite easily by considering only a few high order bits of the address and displacement (0) by considering the remaining low order bits of the address. This is shown in Fig. 8.17. The point is: How many bits should be reserved for P and how many for D1 The answer to this depends upon the page size which determines O and maximum number of pages in a process image which determines O and maximum number of pages in a process image which determines P. Given that the total size of the process image = page size x number of pages which is a constant, a number of possibilities can arise. For a process image of 256 bytes, we can have:

- Page size = 128, number of pages = 2 (P has I bit, O has 7) or
- Page size = 64, number of pages = 4 (P has 2 bit, D has 6) or
- Page size = 32, number of pages = 8 (P has 3 bit, O has 5) or
- Page size = 16, number of pages = 16 (P has 4 bit, O has 4), and so on.

The decision of page size is an architectural issue, which has an effect on performance. We will study this later. The point is: the beauty of binary system is such that, whatever the page size may be, the one dimensional address is same as the two-dimensional one.

Normally, in commercial systems, the page size chosen varies from 512 bytes to 4 Kb. Assuming that I Megabyte or I MB (= I024Kb) of memory is available and page size as well as the page frame size is = 21(, we will require 1024/2 = 512 page frames numbering from 0 to 511 or from 000000000 to 1I111I1I1 in binary. Hence, the 9 high order bits of the address can be reserved to denote the page frame number. Each page has 2K (2048) locations numbering from 0 to 2047; thus, requiring II bits for displacement O. (512 requires 9 bits, 1024 would require 10 and 2048 would require 11 bits). Thus, the total address would be made up of 9 bits for page number + II bits for displacement = 20 bits.

Memory Manogement



~ Fig. 8.20 After allocation of memory to process D

There is one PMT tor each process and the sizes of different PMTs are different. Study the free page frames list before and after the allocation. The page frames list need not always be in the sorted order of frame numbers. As and when the frames are freed by the processes which are terminated or swapped out,

Memory Managemeru

JOJ

					0
	P	O	F	O	1
CPU	0011	01011	0010	01011	2

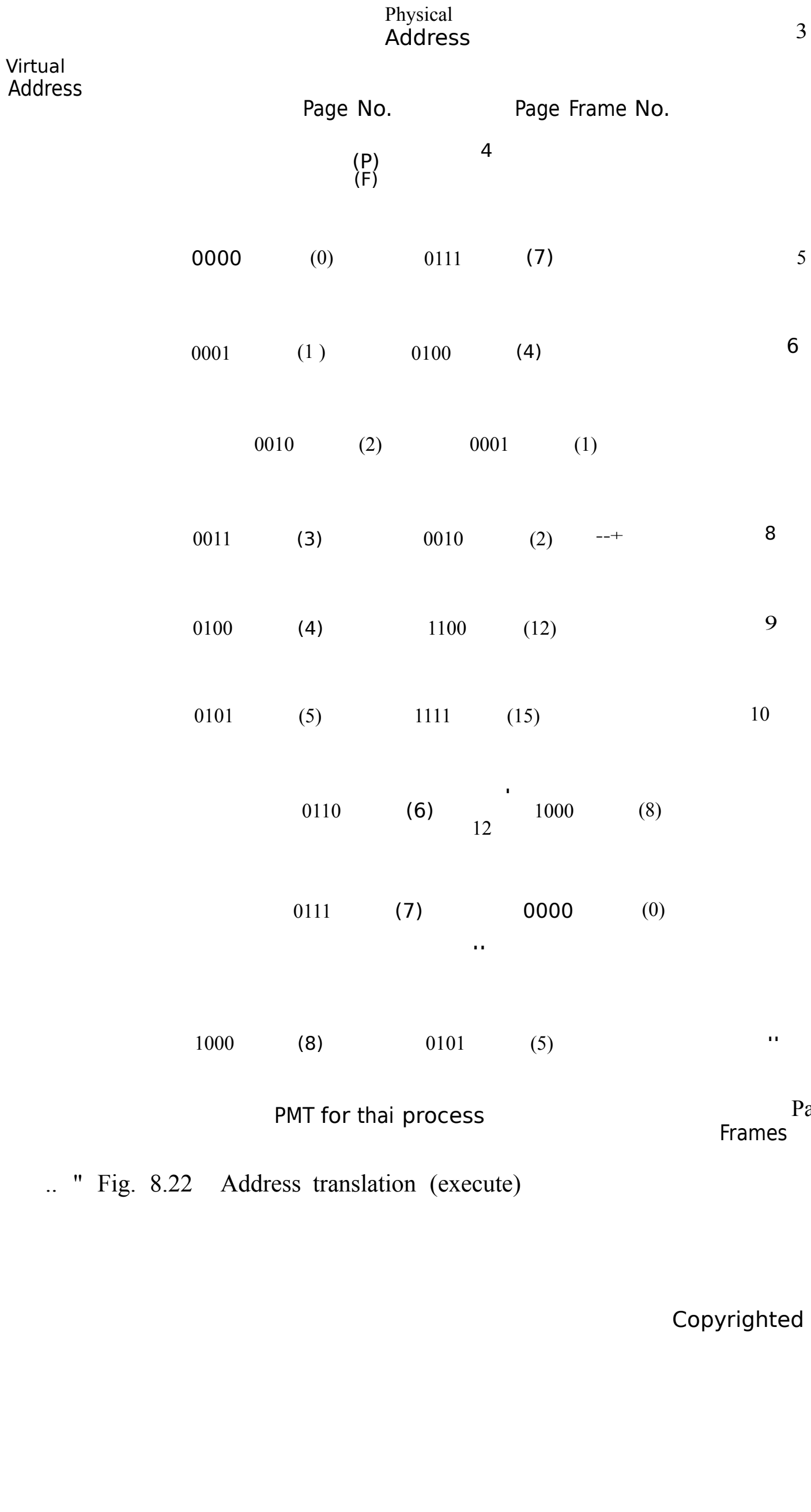


Fig. 8.22 Address translation (execute)

(iii) Hybrid Method The hybrid method provides such a mechanism. In tbis method, associative memory is present, but it consists of only 8, 16 or some other manageably SInaUnumber of registers. This reduces the cost drastically. Only the pages actuaJJy referenced frequently are kept in the associative memory with the hope that they will be referenced more frequently.

The Address Translation in the hybrid method (Fig. 8.25) is'carried out in following fashion:

(a) The virtual address is divided into two parts: page number (P) and displacement (0) as discussed earlier. This is a timeless operation.

(b) The page number (P) is checked for its validity by ensuring that P <= PMTLR. This takes virtu- ally no time, as this comparison takes place in the hardware itself .

- Found in associative memory

(c) If P is valid, a check is made to see if P is in the associative registers, and if it exists there, the corresponding page frame number (F) is extracted directly from the associative registers. Tbis operation takes some time $-I._{ma}$ as seen earlier. This time is required regardless whether P exists in associative registers or not.

(d) The original displacement (0) is concatenated to F, to get the final physical address F + D. This, again, takes virtually, no time.

(e) Using this address, the desired item in the main memory is finally accessed. This requires the time t_{nla} as discussed earlier.

Thus, if found in associative memory, the total time required = tama + (Illa' as in pure hardware

method. If P is not found associative registers, the method followed is the same as the pure soft-

ware method. The steps for this are as follows:

- Not found in associate memory

(f) In the software method, P is used as an index into PMT. PMTBR is added to P (requiring very negligible time) to directly find out the desired entry number ofPMT.

(g) The selected entry of PMT is fetched into the CPU register. This operation requires a memory access time = *tlllJV* became full PMT is in the main memory.

(h) The page frame number (F) is extracted from the selected PMT entry brought into the CPU regis- ter. This, again, is almost a timeless operation.

(i) The original displacement (0) is now concatenated to F requiring negligible time to get the final physical address F + J?

(j) Using this physical address, the desired data item in the main memory is now accessed. This requires time Ima as seen before.

Thus, the total time required if P is not found in associative registers = $(..{}_{na} + 21_{m''}$.

Assuming the hit ratio *(h)* = the fraction of times that P is found in the associative registers, we can fmd out the Effective Access Time (EAT) as follows:

$$EAT = h(/'''',+ t'''')+(I-h) \quad (t\bullet me +$$

211lJQ) Let us assume that $I._{ma}$ = 40 ns, $t_{n'',}$ = 800 ns and *h* = 0.8

We get

$$EAT = 0.8\ (40 + 800) + 0.2\ (40 + 1600) = 1000\ ns$$

Therefore, we see the following degradations:

- Pure time to reference a memory location (if this were possible) = 800 ns.
- Time to reference a memory location using pure hardware method = 40 + 800 = 840 ns (5% degradation.)
- Time to reference a memory location using pure hybrid method with hit ratio = 0.8 = 1000 ns (25% degradation).
- Time to reference a memory location using pure software method with hit ratio = 800 + 800 = 1600 ns (100"10degradation).

entry will give the page frame number (F) corresponding to the virtual page number (PI + P2). The offset or displacement (0) can now be concatenated to this page frame number (F) to form the final resultant physical address. This address is then put on the address bus for accessing the desired memory locations.

The address translation in this scheme is carried out with the help of hardware, though it is a bit more complicated because of the levels involved. For instance, the hardware will have to separate the bits for P I first, use them as an index to access the correct entry of the first level page table, and so on.

The main advantage of this scheme is that all the page tables need not be kept in the main memory. Consider a process needing 12MB of memory where the bottom 4MB of memory is for the text, the next
4MB is for data and the top 4MB is for the stack. In this scenario, there is a large gap or hole in between the data and the stack. Normally in such a scenario, a very large PMT would have been necessary to be kept in memory. But, due to the multilevel page tables, only one first level page table and a few second level page tables are needed to be kept in the main memory.

This discussion is only an example. In practice, there are different levels of paging that are possible. PDP-II for instance, uses one level paging, the VAX has two levels paging, the SUN SP ARC has three levels of paging and M68030 bas four level paging. In fact, M68030 can have a level of paging which is programmable. The levels can be 0-4 and the Operating System controls these levels. The virtual address space is divided into as many parts as there are levels. An interesting idea used in M68030 is that the number of bits reserved in the virtual address for each level is also programmable. This complicates the hardware and algorithms involved in Address Translation, but it provides a lot of flexibility to the whole operation.

Protection and Sharing

Protection in a paging system is achieved mainly by the PMTLR register already discussed. This register ensures that the reference is made in the address space of that process only. If you want to have protection at the level of a page, you can have access protection bits added to each PMT entry and also the corresponding hardware to test these bits. For each page then, you could define access rights as Read Only (RO), Read Write (RW), etc. denoted by different access bits (e.g. 010 = Read only, 011 = Read! Write). Thus, in addition to checking whether a process is allowed to aecess that page or not, the Operating System can also detect if a process is trying to update some data in the page when not allowed to do so.

This scheme is theoretically useful, but in practice not very much so, because a page is a physical entity comprising several bytes. It is not a logical entity as in segmentation. Normally, a programmer would like to use the access rights to logical entities rather than physical ones. For instance, it would be very difficult for a programmer to know in advance where a specific routine will start in a page, and at which instruction it will start with a new page. Therefore, it is very difficult for a programmer to assign these access protection bits at the page level.

Protection in paging can also be achieved by protection keys as discussed earlier. If you recall, protection keys are defined fordifferentmemory blocks as in IBM systems. If the page size is same as the memory block size, protection keys can be used for protection of different pages in a process against unauthorized access as discussed earlier.

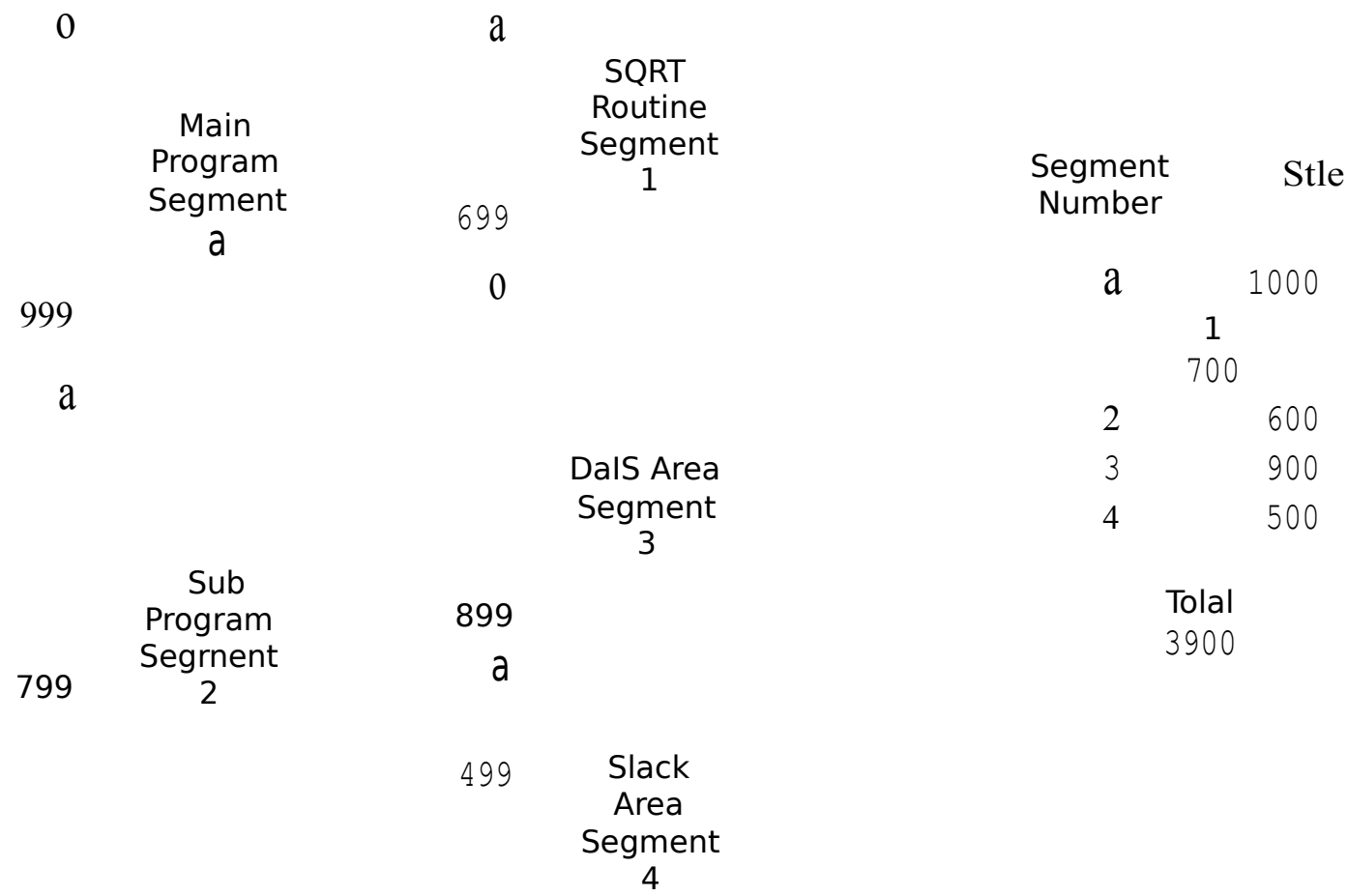
SEGMENTA

8.7.1 Introduction

Segmentation and paging share a lot of common principles of operations, excepting that pages are physical in nature and hence, are of fixed size, whereas segments are logical divisions ofa program and hence, are normally of variable sizes.

For instance, each program in its executable form can be considered to be consisting of three major segments: code, data and stack. Each of these can be divided into further segments. For example, in a program, you normally have a main program and some subprograms. These can be treated as separate segments. A program can use various functions such as "SQRT". In this case, a routine for "SQRT" is prewritten and precompiled. This becomes yet another segment.

Let us say that we have a program with the segments as shown in Fig. 8.29. The figure shows various segments along with their sizes which are obviously found out at the time of compilation, and stored in the executable file for future use.



~ **Fig. 8.29** An example of segments

Each segment is compiled with respect to 0 as the starting address. The segment numbers and their sizes are shown in the table within the figure.

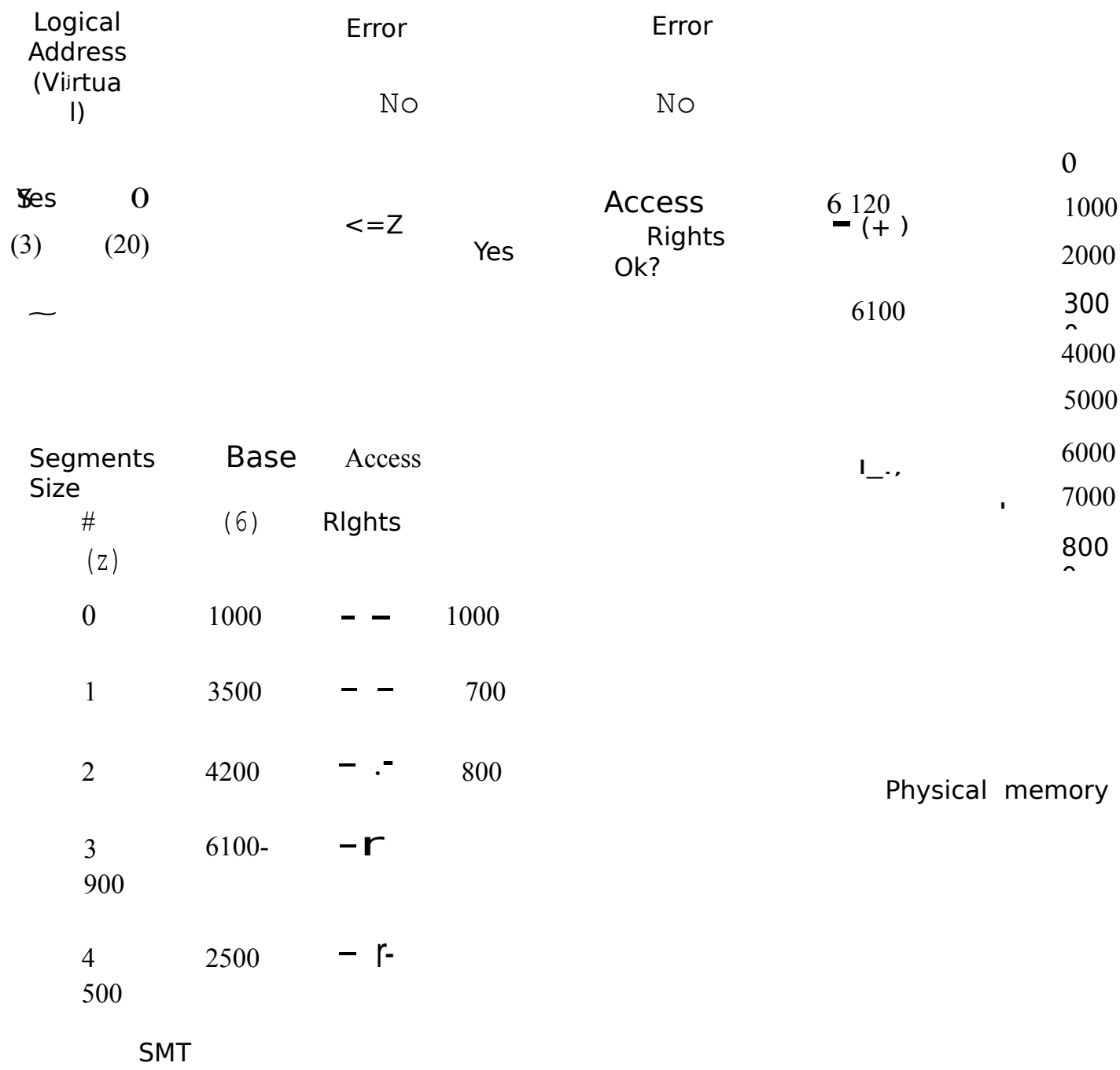
An application programmer does not necessarily have to declare different segments in his program explicitly. If the programmer defines different overlays or segments explicitly, it is fine, but otherwise the compiler/linkage editor does it on its own. In any program, it is not ve.ry difficult to recognize the main program, the "called" subprograms, the common routines, the data area and the stack area. The compiler/linlcage editor thus does the following:

(iii) We know that displacement (D) is a virtual address within that segment Hence, it has to be less than the segment size. Therefore, the displacement (0) is compared with Z to ensure that D <= Z. If not, the hardware itself generates an error for illegal address. This is evidently a protection requirement. In our example, D which is 20, is less than Z which is 900, which is acceptable.

(iv) If the displacement (i.e. D) is legal, then the Operating System checks the access rigbts and ensures that these are not violated. We will discuss the access rights under protection and sharing, later.

(v) Now, the effective address is calculated as (B + D), as shown in the figure. This will be computed as 6100 + 20 = 6120 in our example.

(vi) This address is used as the actual address and is pushed into the address bus to access the physical memory.



~ Fig. 8.33 Address Translation in Segmentation

Various problems and their solutions in this scheme are presented briefly, as they have been already discussed in detail in earlier sections.

(a) Each process has one SMT. Thus, the Operating System will have to reserve a large memory space merely to store all these SMTs. This also means that when a process is 'dispatched', we need to start referring to the correct SMT. This is facilitated by the Segment Map Table Base Register (SMTBR) similar to the PMTBR in paging systems. Similarly, a logical segment number (S) can be validated against another Segment Map Table Limit Register (Sl\lTLR) before getting into the SMT as an additional protection. The SMTLR stores the maximum number of segments in a process. The compiler keeps this information in the header of the executable file for

It will also be noticed that only the data segments are differently mapped for obvious reasons. (Each program will work with its own records and data.)

As in paging, when one of the processes terminates (say, user A finishes his editing), only the data segment portion, i.e. locations 2000 to 2999 are freed and added to the free memory pool. The code segments are released only when all the processes using that editor have terminated. The Operating System has to keep some kind of usage count for this in the same way as was seen earlier in the Basic File Directory (BFD) of the File Systems.

In fact, sometimes we want only certain segments ofa program to be shared (e.g. a common routine such as SQRT - to find out the square root). This poses a problem. All addresses in segmentation have to specify the segment number (S) and displacement (D). If in SQRT routine itself, there is a reference to a location in the same routine (say, Jump to an address within the same routine), how should that address be generated? What segment number should be assumed? If this precompiled routine is added to program A. as segment 4 and to program B as segment 5 at the time oflinldng, how can the scheme of Address Translation work, if only one copy ofSQRT in the pbysical memory is to be shared by both A and B?

One solution to this problem is to use only PC-relative addressing in sharable segments. A more general scheme is to use as many registers as there are segments, and use relative addressing to those registers. This has repercussions on the computer architecture, the compiler and the Operating System. MUL TICS implemented on GE645 used this scheme. A detailed discussion of this is, however, beyond the scope of this text.

Protection is achieved by defining access rights to each segment such as Read Only (RO) or Reaell Write (RW), etc. and by defining certain bit codes to denote those e.g. 01 = RO, 10= RW. These are the access rights bits in any SMT entry. When a user logs on and wants to execute a process, the Operating System sets up these bits in the SMT entries while creating the SMT, depending upon the user privileges. As discussed before, if there are 4 processes sharing a segment, all the 4 SMTs for those 4 processes will have a common entry for that segment. In all the entries for a shared segment, base (B), size (Z) will be the same, but access rights can be different in different SMTs for different sharing processes. Because of this scheme, the shared segments also can be well protected. For instance, a shared segment can have only access rights "RO" for process A and can have "RW" for process B.

Protection in terms of restricting the accesses to only the address space of tbat process is achieved during Address Translation as shown in Fig. 8.33 using the "size *(Z)*" field in the SMT entry, and ensuring that the displacement (D) is not more than Z. Also, the segment number S is checked to ensure that S =< SMTLR as shown in Fig. 8.34.

The combined systems are systems which combine segmentation and paging. This requires a three-dimensional address consisting ofa segment number (S), a page number (P) and displacement (D). There are two possible schemes in these systems.

- Segmented paging as in IBM *360/67* or Motorola 68000 systems is one such scheme. In this scheme, the virtual address space of a program is divided into a number of logical segments of varying sizes. Each segment, in tum, is divided into a number of pages of the same size .
- Paged segmentation as in GE645IMUL TICS.

In either of the above cases, two memory accesses for the final Address Translation are required thus, slowing down the process. The resulting reduction of the effective memory bandwidth by two-thirds may be too high a price to pay. In such a case, the hardware support for Address Translation in terms of

not valid, you could not throw any pages out on the disk from the memory without a possible severe degradation in the performance.

Poge Fault Inmany systems, when a process is executing with only a few pages in memory, and when an instruction is encountered which refers to any instruction or data in some other page which is outside the main memory (i.e. on the disk), a page fault occurs. At this juncture, the Operating System must bring the required page into the memory before the execution of that instruction can restart. In this fashion, the number of pages in the physical memory for that process gradually increases with the page
faults. After a while, when a sufficient number of required pages build up, the pages are normally found in the memory and then the frequency of page fault reduces.

Working Set At any time, a process has a number of pages in the physical memory. Not all of these are actively referred to at that point in time, according to the locality of reference. The set of pages in the physical memory actively referred to at any moment is called working set. This has a significant bearing on the policy of bringing in pages from the disk to the main memory, if the Operating System follows the working set model for this purpose. As discussed earlier, there exist different working sets in housekeeping, during the main routine and during the End-of-lob routines; In this scheme, the Operating System maintains a list of actively referred pages as a working set at any given time. If due to low priority, the entire process is swapped out and swapped in again, the pages corresponding to the working set can be directly swapped in, instead of having to start from the scratch. The working set, therefore, introduces the knowledge about the current page references and, therefore, has to be stored at the context switch.

Page Replacement Policy As the number of processes and the number of pages in the main memory for each process increase, at some point in time, all the page frames become occupied. At this time, if a new page is to be brought in, the Operating System has to overwrite some existing page in the memory. The page to be chosen is selected by the page replacement poli~ as discussed earlier. There are a number of ways in which the Operating System selects a page for overwriting. The Operating System designer chooses amongst one of many such policies and writes a corresponding algorithm for it. We will study these possible algorithms in later sections.

Dirty Page/Dirty Bit Before overwriting a page in the memory, the Operating System has to check if that page has been modified after it was loaded frOID the disk. For instance, there may be a page containing the *UO* area where an employee record is to be read. Initially, that page will be loaded with the *1/0* area as blank. After reading the record, that page gets modified. Such a modified page is called cUrly page. Normally, these days the compilers produce reentrant code which does not modify itself. Hence, the pages reserved for the code portion of a program normally never get dirty, but the ones for the data portion can. The Operating System maintains 1 bit for each physical page frame to denote whether
a page has became dirty or not. This bit is called dirty bit.

The hardware is normally designed in such a fashion that if a page is modified, the dirty bit for that corresponding page frame is set to I automatically; otherwise, it is maintained at 0. The Operating System supporting the virtual memory scheme expects this support from the hardware. The hardware knows the "write" instruction used for updating a page from the opcode bits in the IR. It also knows the page frame modified by the resultant address derived after the address translation. The hardware, therefore, can set this bit corresponding to the modified page to 1. This bit is called dirty bit because it indicates whether a page is modified or not, and therefore, whether it needs to be saved on the disk before getting overwritten.

OIS
Page Faull
. Routine

~

#

Disk
Address

0

1

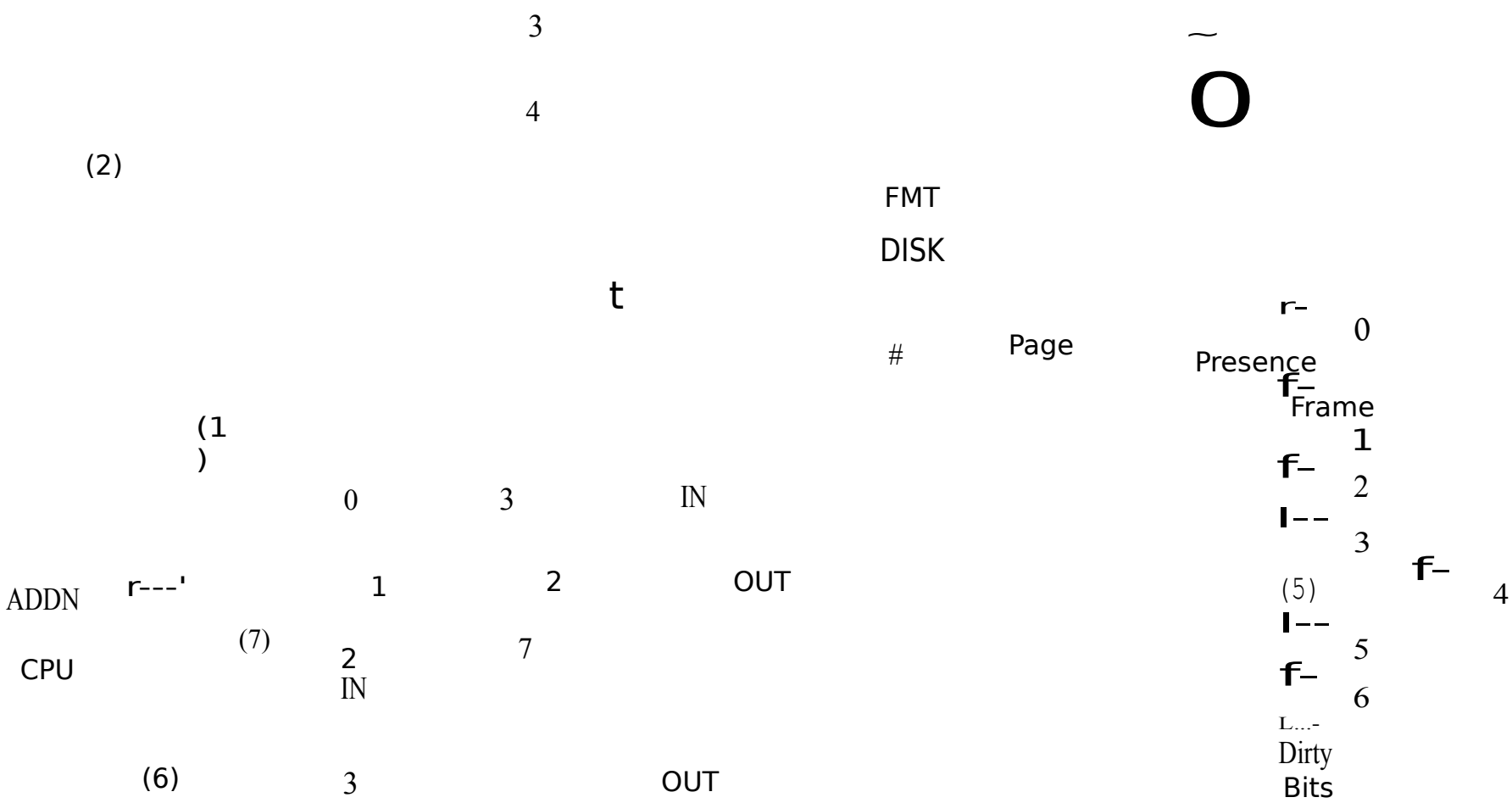
2

(4)

O

B

O



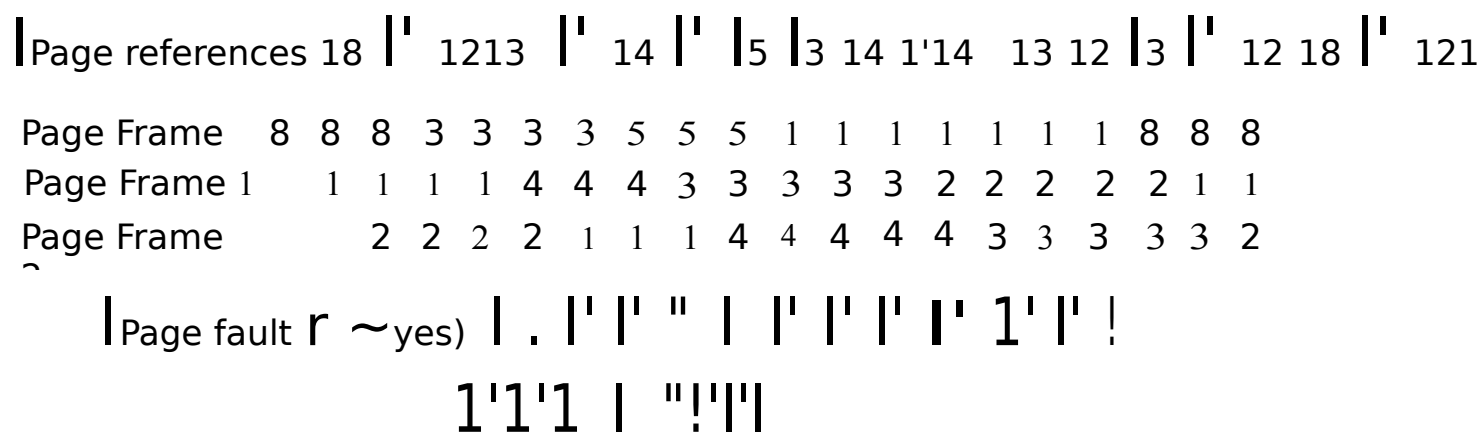
Physical Memory

....., Fig. 8.41 Sequence of operations

We present in Fig. 8.40, a process with 5 logical pages A to E numbered Q-4. Let us imagine that page numbers 0 and 2 (i.e. pages A and C) are already loaded in the physical memory page frames 3 and 7, respectively. This status is indicated by the PMT. We have shown that out of these, page frame 7 containing page number 2, i.e. page C is dirty. Pages I, 3 and 4 of that process are still on the disk. Therefore, the PMT entries for those pages show the presence bit for them as "OUT". We also show the FMT which gi ves us the disk addresses of all the logical pages.

Let us imagine that an instruction of the type "ADD N" is encountered in the Instruction Register (IR) of the CPU. We assume a direct addressing mode. Thus, N is the address of the memory locations where the data-item to be added to the CPU register will be located. We assume that the instruction is already in the IR and the task is to fetch the data item at address N to the ALU for addition. The sequence of events that now happen is depicted in Fig. 8.41. The steps are numbered in the figure to facilitate our explanation. We describe these steps below in detail so that the overheads of the page fault processing become clear.

First In First Out (FIFO) As the name suggests, in this scheme, the page that is removed from the memory is the one that entered it first. Assuming the same page reference string as before, we present in Fig. 8.44, the states of various page frames and page faults after each page reference. As is clear from the figure, 15 page faults result.



'--' Fig. 8.44 FIFO algorithm

This algorithm is easy to. understand and program. The first three columns are self-explanatory. In the fourth reference of page 3, a page fault results, and the FIFO algorithm throws out page 8 because it was the first one to be brought in. (It is a coincidence that the OPT also would decide on the same.) The fifth page reference does not cause a page :£'Iu11in both the OPT and FIFO algorithms as page I is already in the memory. The sixth page reference is for page 4. Here, FIFO will throw out page I, because it came in earlier than the remaining two pages at that time, viz. pages 3 and 2. Page I has been there the longest. Notice that OPT had chosen page 2 tor throwing out. The reason is that it "knew' in advance that page 1 is going to be required sooner. The FIFO policy does not "know" this. Hence, in just the next, i.e, the seventh page reference, page 1 is required causing yet another page fault. Following this logic. the table can be completed.

This algorithm can be implemented using a FIFO Queue. The queue is nothing but a pointer chain where the header of the chain is the page that came in first, and the end of the chain is the page that came in last. Thus. before the eighth reference of page 5 (i.e. after the seventh one for page 1), the queue would be as shown in Fig. 8.45. The figure shows the 3, 4 and I ill the page frames 0, 1, and 2 with a specific logic. Out of the pages 3,4, and 1 in the memory at that time, page 3 had come in the memory the earliest (in the fourth reference). Page 4 had come in the memory in the sixth page reference; and page I had come in the seventh page reference. Notice that page I had also come in the memory in the second page reference, but it had been evicted in the sixth page reference. Hence. that was of no consequence.

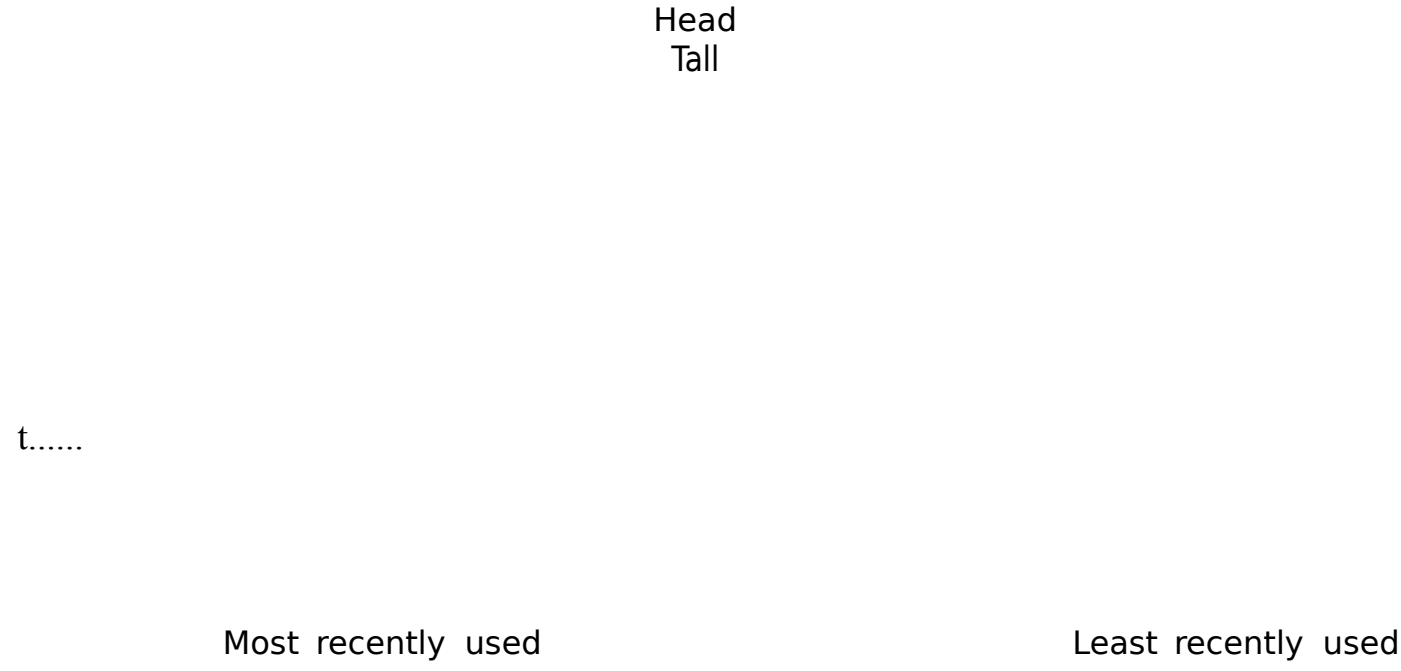
;.....F, ig. 8.45 FIFO Queue before 8th reference

belong to this class. That is the reason why we encounter the FIFO anomaly. What does the case depicted in Fig. 8.53 demonstrate? The case demonstrates that there can be a case when a policy appearing to be logically the best can produce the worst results. LRU can be bad in another situation too, though not as bad as the worst case. This can happen when the working set undergoes a drastic change, e.g. from housekeeping to the main routine.

Despite these problems, LRU is considered to be a very good algorithm coming very close to the OPT algorithm excepting in some situations as discussed above. The only problem with LRU is that it is very costly to implement. We will now study some of the ways in which it can be implemented.

Implementation of LRU Algorithm LRU can be normally implemented using one of the three methods: Stack, Counters and Matrix. We will discuss them in brief.

Stack (or Linked List) One approach to implementing LRU is to keep a stack of page numbers. Whenever a page is referenced, it is removed from its current position of the stack and is pushed on the top of the stack. Thus, the top of the stack always contains the most recently used page; and on the other hand, the bottom of the stack contains the least recently used page. This is normally implemented using a doubly linked list, because'eiiiries have to-be removed-from the middle 'Ofthe stack and repositioned, if a page is referenced. Figure 8.54 demonstrates this.



~ Fig. 8.54 LRU implementation using stack

This doubly linked list has a head and tail pointers as usual. By changing six pointers, one can remove a page and bring it at the head of the chain ifreferenced. It is not inexpensive, but if one has to implement LRU in software or microcode, this is probably a better option.

Counters In this scheme, we need a hardware counter (c) of large length (say 32 or 64 bits) to accommodate a large binary number. After each instruction, it is automatically incremented by the hardware. Hence, at any time, this counter is a measure of time. In addition to this system-wide counter, this method demands that there should be another counter of the same length as (c), associated with each page frame, as shown in Fig. 8.55. This counter is maintained to indicate the last time the page frame was referenced.

be removed? If it does so, Process A will continuously cause page faults for a while, thus degrading the performance.

Another problem is to decide when to free the pages. For instance, in the earlier example, when Process A also terminates, all the shared pages should be returned to the free pool (unless there is yet another process running an editor). How does the Operating System know this? One idea as discussed before is to use a usage count associated with each page frame. When a PMT for a process is created, the usage count field for all the allocated frames is incremented by 1. When a process is killed, they are
decremented by I. Now, at any moment, the page frames with zero usage count can be added to the free pool after a process terminates because these are not shared pages.

- Access protection bits
- Address translation
- Allocation algorithms
- Associative registers
- Basic File Directory (BFD)
- Best fit
- Block Move
- Cache
- Combined system
- Access times
- Aging
- Associative Memory
- Base register
- Belady's anomaly
- Bit maps
- Buddy system
- Coalescing of holes
- Compaction

- Content addressable memory
- Contiguous, Real Memory Management System
- Demand paging
- Dirty bit
- Dynamic relocation
- Execute Cycle
- Fence register
- Fetch policies
- FIFO anomaly
- File Map Table (FMT)
- First In First Out (FIFO)
- Frame
- Global replacement policy
- Hole
- Internal Fragmentation
- Least Recently Used (LRU) Algorithm
- Linked lists
- Locality of reference
- Logical addresses
- Look-ahead Memory
- Look-aside Memory
- Matrix
- Memory bandwidth
- Memory protection
- Multilevel Page Tables
- Contiguous Memory Management
- Counters
- Demand feeding
- Demand segmentation
- Empty slot
- Effective Access Time
- External Fragmentation
- Fetch Cycle
- FIF
- FIF queue
- First fit
- Fixed Partitioned Memory Management
- Global replacement
- Hardware Method
- Hybrid Method
- Key
- Limit register
- Local replacement policy
- Logical Address Space
- Look-ahead Buffer
- Look -aside Buffer
- LRU approximation
- Memory Management
- Memory sharing
- Multiple Queues

8.3 When the memory wastage is within the partition itself it is called _
(a) Compaction (b) External Fragmentation
(c) Internal Fragmentation (d) Worst Fit

8.4 In paging scheme any virtual address consist of and parameters.
(a) virtual page number, displacement (b) physical address, virtual address
(c) protection bits, fence register (d) single queue, multiple queues

8.5 is used to loeate the PMT itself in the memory.
(a) PMTLR (b) PMTBR (c) MAR (d) PMTCR

8.6 Address in segmentation is of dimensional.
○○~ ○○~ ○○~ ○○fiw

8.7 stores the maximum number of segments in the process.
(a) PMTLR (b) PMTBR (c) SMTBR (d) SMTLR

8.8 The rate at which page fault occurs is called -:--=._:-:-_
(a) Prepaging (b) LRU algorithm (c) Thrashing (d) NFU algorithm

8.9 Protection bits and are the two methods used to achieve protection.
(a) IntemalFragmentation (b) Wasted Memory
(c) Fence Register (d) Segmentation

8.10 The process of moving the program from disk into main memory is called
_ (a) Swapping in (b) Swapping out (c) Thrashing (d) Prepaging

Test Questions

8.1 Name the different memory management schemes.
8.2 Contrast contiguous versus non-contiguous memory management scheme.
8.3 Contrast real versus virtual memory management systems.
8.4 Contrast fixed versus variable partitioned memory management schemes.
8.5 Contrast paging and segmentation.
8.6 Discuss all the memory management schemes with respect to the following considerations:
(a) Reloeation and Address Translation
(b) Protection
(c) Sharing
(d) Wasted memory
(e) Access times (speed)
(f) Time complexity
8.7 Contrast and explain coalescing and compaction, explaining different methods to achieve them along with their merits/demerits.
8.8 Explain the Address Translation mechanism in Paging. Why is the page size normally some power of two?
8.9 Discuss the impact of page size on the overall system performance.
8.10 Contrast demand paging versus working set model as a 'page fetch policy' .
8.1 I What is Dirty page and Dirty bit? How are they used?
8.12 Explain the hardware, software and the hybrid methods of maintaining the page map tables and Address Translation.
8.13 Bxplain what is meant ~y reentrant code and its significance in memory sharing.

Of these five types of security threats, the first two, viz. tapping and disclosure, are categorized as passive tbreats and the other three as active threats. It is clear from the figure that, in both cases (i) and (ii), the information goes to the third party. However, there is a slight difference between tbem. In tapping, the third party accesses it without the knowledge of the other two parties (A and B). In disclosure, the source party (A) willingly or knowingly discloses it to the third party.

Security threats can arise from unintentional or deliberate reasons. Again, there may be casual or malicious attempts at penetration. Regardless of its origin and motivation, the Operating System designers have to build a security system to counter all possible penetration attempts. Disclosure of product prices or other competitive information or military/commercial secrets could be extremely dangerous in varying ways and degrees to people, organizations or governments.

The security system can be attacked and penetrated in a number of ways. Following sub-sections outline some of the possible ways in which this can happen.

9.3.1 Authentication

Authentication means verification of access to the system resources. We will talk about it in more detail in a later section. Following could be some of the ways of penetration of the system in this regard. (i) An intruder may guess or steal somebody else' s password and then use it.

(ii) An intruder may use the vender-supplied password which is expected to be used for the purpose of system generation and maintenance by only the system administrators.

(iii) An intruder may find out the password by trial and error method. It is fairly well known that names, surnames, initials or some other common identifiers are generally used as passwords by many users. A program also could be written to assist this trial and error method.

(iv) If a user logs on to a terminal and then goes off for a cup of coffee, an intruder can use that terminal to access, or even modify, sensitive and confidential information.

(v) An intruder can write a dummy login program to fool the user. The intruder, in this case, can write a program to throw a screen, prompting for the usemame and the password in the same way that the Operating System would do. When a user keys in the usemame and password for logging in, this dummy program collects this information for the use by the intruder later on. It may, then terminate after throwing back some misleading message like "system down ...", etc. This collected information is used for future intrusion. This is a form of 'chameleons' as we willieam later.

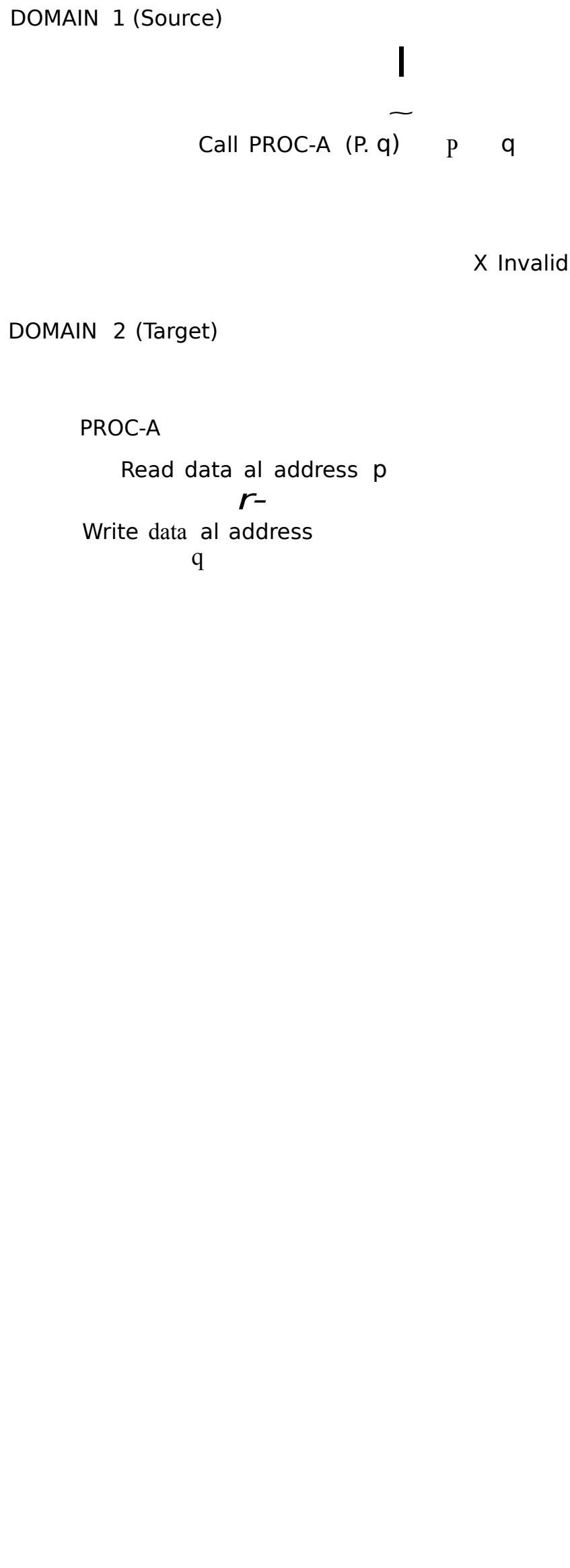
9.3.2 Browsing

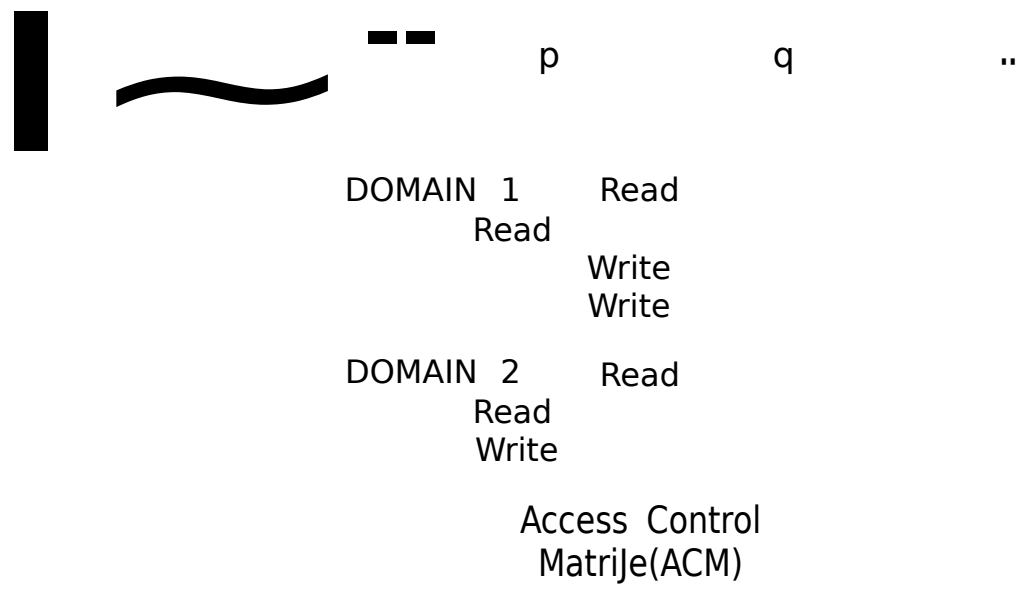
Quite often, in some systems, there exist files with access controls which are very permissive. An intruder can browse through the system files to get this information, after which unprotected files/databases could easily be accessed. Confidential information could be read or even modified which is more dangerous.

9.3.3 Trap Doors

Sometimes, software designers may want to be able to modify their programs after their installation and even after they have gone in production. To assist them in this task, the programmers leave some secret entry points which do not require authorization to access certain objects. Essentially, they bypass certain

A call will be made as usual, but at the time of execution, when the data read at address *p* has to be written at address *q,* the access will be denied because DOMAIN 2 does not have a write access for *q*. Thus, the denial of the service results in this case.





"--' Fig. 9.2 Denial of service

9.4.2 A More Serious Violation

A more serious violation than denial can occur if data gets accessed, or even modified, by unauthorized processes. This is not possible in case the parameters are passed by value. It can take place only if they are passed by reference. Figure 9.3 depicts this scenario.

As before, DO'MAIN 1 (source domain) contains a call to a procedure called PROC-A. This procedure has two parameters *p* and *q*. Let us assume that *p* is a direct parameter and *q* is an indirect one. The procedure (given in DOMAIN 2) is supposed to pick up the value *p* which is an integer and then write it at the address X. (At the time of execution, the value 'X' is substituted for *q.)* Let us assume that DOMAIN 2 is a more privileged one than DOMAIN 1 as is depicted by the Access Control Matrix (ACM) in the figure. For instance, DOMAIN 2 can perform both read and write operations on X, whereas DOMAIN I can only read it. In this case, the operation will be successfully performed in DOMAIN 2; but it should NOT have been allowed! The reason is that ifwe allow this to happen, in effect, DOMAIN

I will successfully write at address X which is contrary to the ACM for OOM,AIN I. And this definitely is a serious access violation!

viral code at this time is set to "Nascent" only. At still subsequent execution of the original viral code, the viral code itself contains instructions to check the status flag and to change it to "Harm" if it is already "Spread". Having changed its own status, it actually damages some other files or data in the system.

```
                                                    r ORDER ENTRY '/
main
()
{

LII=   gote VIRUS; r Instruction 1 is
replaced '/ BACK: Instruction  2;


▌
VIRUS:        H (strcmp(Status. = 'NascenF)  ==  0)
▌
PAOC·NASCENT ( );
elsa  If (atrcmp(Status. 'Spread")  ==0)
PAOC·SPAEAD ( );
elsa  PAOC·HAAM (
); PAOC·NASCENT
( )
Find out if waiting is to be continued.
H (waiting ls
over) Status =
'Spread";
gotoZZZ;
PAce-SPAEAD       ( )
Find out if spreading is to be continUed.
If (spreading is over)
▌
status = "Harm";
gotoZZZ
▌
else
{
Find another uninfected  program;  Copy this viral code at the end of it;
Change the first instruction  to jump to the viral code as shown at LLL in this figure;
Change the last instructions  at PPP. aaa and AAR to go back to correct address in the new program;
gotoZZZ;
)
PAOC·HAAM          ( )
{
Do damage (e.g. encrypt. blackmail,  corrupt  the data etc.)


ZZZ    Instruction 1  r Instruction 1 ls
executed '/ RRR        goto  BACK;
)
```

:.....0' Fig. *9.S* The program OE after the altack

This scheme works as follows:

(a) Assume that the virus is already attached to the Order Entry (OE) program, At this time, the status flag of the virus is set as "Nascent".

Access
Rights

For various objects, the Operating System allows different Access **Rights** for different subjects. For example, for a file, these access rights can be Own, Write, Append, Read, Execute (OW ARE), as in *AOSNS* of Data General machines. UNIX has only Read, Write and Execute (RWX) access rigbts. For example, for a printer as a device, the Access Rights can be 'Write' or 'None' only.

In general, we can list different access rights that can be granted as shown in Fig. 9.6. 10 Fig. 9.6 'Modify Protection (M)' means the ability to modify the Access Rights themselves. AU the rest are self- explanatory .

Serial No.	Access Rights	Code
1	ExectJ1e Only	E
2	Read Only	R
3	Append only	A
4	Update	U
5	Modify protection	M
6	Delete	O

=v Fig. 9.6 Different Access Rights

One way is tomention all the access rights for each object explicitly for each subject. This is a simple method, but requires more disk space.

An interesting alternative could be to organize the access rights in the form of a table as shown in Fig. 9.6 in such a way that the presence of any access right in the table implies the presence of all the ones preceding it in the table. For instance, if a process is allowed to delete a file (0), it should be certainly
allowed to execute it (E), read it (R), append to it (A), update it (U) or modify its protection (M). Similarly, if a process is allowed to update a file (0), it should be allowed to read it (R) but not allowed to delete it (0).

Thus, one could specify only one code against a file to specify all the access rights for it according to this scheme.In this case, we could associate only one code for a subject (user) and object (file) combina- tion. If a user creates a process, the process inherits this code from the user. When it tries to access a file F I, this access right code could be checked before granting any access to that process on F I. This scheme appears deceptively simple, but it is not very easy to create this hierarchy of access rights in a strict sense that only one code implies all the rest above it.

Why does a process inherit the access rights from the users who has created it? This is because mentioning all the access rights for each *process* for all the files explicitly will not only be very expen- sive but also an infeasible solution. The reason: At any time, a process hierarchy of very tall height can be created. The number of processes existing at 311~moment ill the system is hence, not easily predict- able. Implementing this scheme would require the users/administrators to assign all these access rights for each file separately for every process existing in the system. Any time a new file is created, it will entail a huge exercise. At the run time, the system will have to allocate a huge amount of memory and require long search times.

An easier solution is to assign these access rights to each *riser* for different files. When any user creates a process (e.g. runs the Shell program) to access a file, the assigned access rights for that file are

(vii) The Operating System now finds the BFD entry number using the full pathnamc of"EMP.DA T" and various SFDs as discussed earlier. The BFD has a pointer to the ACL records for tbat file (EMP.DA T) as depicted in Fig. 9. I 3. The Operating System now reads the ACL for that tile using the pointer(s).

(viii) The Operating System checks whether the user (with the username = u I stored in the PCB) has the access right "R" for that file. If it does not have it, it rejects the request with an error message;
otherwise it proceeds. For all the subsequent read operations, the Operating System need not again check the ACL (though according to the strictest security standards, this checking should be done at every operation).

(ix) Even if a file is being shared and actually being updated simultaneously by a number of processes initiatedby different users (e.g. Airlines booking), the access control verification proceeds as discussed above. This is made feasible, because, even if different user programs may call the file by the same or different symbolic names, it will have only one BFD entry in the system which points to a unique ACL for that file.

(~) Let us assume that "PAY. COB" calls a subprogram called "TAX.COB". When "TAX.COB" is called, a separate child process is created for this. It has its own Process Control B lock (PCB). Along with other detai Is, the username "u 1" is also inherited from the parent process (in this case "PAY.COB") and copied in the PCB of the new child process for "TAX.COB". If"TAX.COB" now wants to write to any file, ACL is verified in the same way as discussed above, i.e. the ACL for that new file is accessed using its pathname and the BFD, and then access rights are verified for that file for ul. It should have a (W)rite access right, ifTAX.COB is to be allowed to update that file.

We hope, this clarifies the exact steps that the Operating System goes through for the verification of the access rights. We will study how this is done for UNIX in the chapter devoted to a case study on UNiX.

9.9.3 CapabilityList

General Working

In the ACL method discussed earlier, we had sliced the Access Control Matrix (ACM) by the column. If we slice the ACM horizontally by a row, we get a 'capability list'. This means that for each user, the Operating System will have to maintain a list of the files or devices that he can access and the way in which it can be accessed. For example, Fig. 9.15 depicts the capability list tor user 3 in the ACM shown in Fig. 9.12.

Serial Number	Objects	Access Rights	Pointer (Address) the Object
0	file 2	-W-	Address file·2
.		R--	Address ..•..
:	file 10	--X·	Address file-10

.....Fig. 9.15 Capability list for user 3

1. Rail Fence Technique

The Rail Fence Technique is an example of transposition cipher. It uses a simple algorithm as shown in Fig. 9.22.

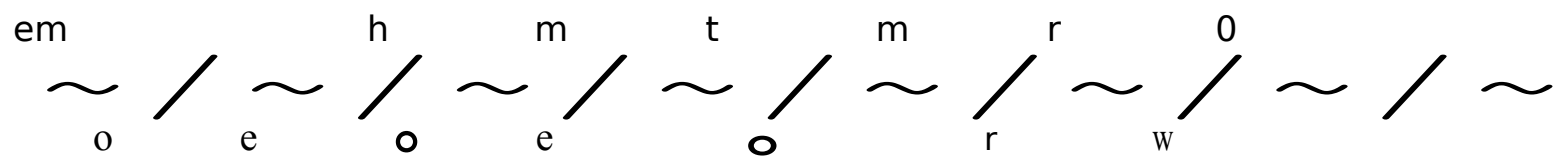
1. Write down the plain text message as II sequence of diagonals.
2. Read the plain text written in *step* 1 as a sequence of rows .

.....Fig. 9.22 Rail fence technique

Let us illustrate the Rail Fence Technique with a simple example. Suppose that we have a plain text message *Come home tomorrow.* How would we transform that into a cipher text message using the Rail Fence Technique? This is shown in Fig. 9.23.

Original plain text message: *Come* home *tomOlTOw*

1. After we arrange the plain text message as a sequence of diagonals, ~ would look as follows (write the fist character on the firstline, i.e.C, then next character on the second line, i.e.o, then the seoond character on the first line, i.e.m, then second character on the second line, i.e. e, and so on). This creates a zigzag sequence, as shown after the text.



2. Now read the text row-by-row, and write it sequentially. Thus, we have:
   Cmhmtmrooeoeoorw as the cipher text.

~ Fig. 9.23 Example of rail fence technique

As Fig. 9.23 shows, the plain text message *Come home tomorrow* transforms into *Cmhmtmrooeoeoorw* with the help of Rail Fence Technique.

2. Simple Columnar Transposition Technique

Variations of the basic transposition technique such as Rail Fence Technique exist. Such a scheme is shown in Fig. 9.24, which we shall call as Simple Columnar Transposition Technique .

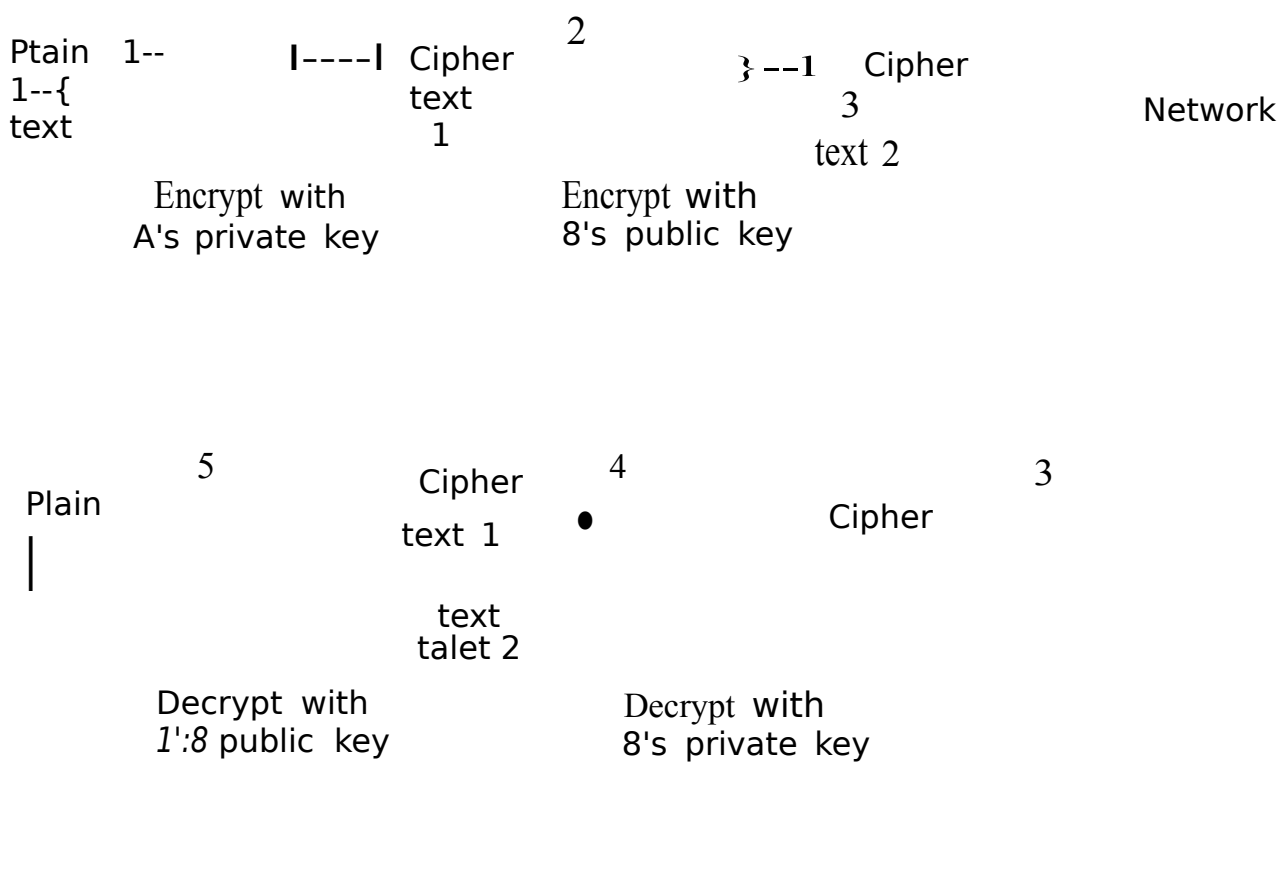
.1. Write the plain text message row-by·row in a rectangle of a pre-defined size.
2. Read the message column-by-oolumn. However, it need not be in the order of columns 1, 2. 3, etc. II can be any random order such as 2. 3. 1. etc.
3. The message thus obtained is the cipher text message.

c:;;. Fig. 9.24 Simple columnar transposition technique

Let us examine the Simple Columnar Transposition Technique with an example. Consider the same plain text message *Come home tomorrow.* Let us understand how it can be transformed into cipber text using this technique. This is illustrated in.Fig. 9.25.

principles of encryption and decryption. When a sender signs a message digitally, the digital signature establishes the concept of nOD-repudiation .. That is, the digital signature signifies that the person had signed a message digitally, and it can be proved using the digital signature. The sender cannot refuse or deny sending a signed message as the technology of digital signatures establish the necessary proof.

Let us understand how digital signatures work:, step-by-step, taking the same example of A (sender) and B (recipient) with the help of Fig.
9.31.



C> Fig. 9.30 Authentication

The above scheme works on the principle that the two keys (public and private) can be used in any order. Hence, the cipher text I at points 2 and 4 in the diagram is the same because it is encrypted by B's public key and decrypted by B's private key. Therefore, conceptually, we can remove the portions be- tween points 2 and 4. Now applying the same principle, the plain text at points I and 5 is also the same because it is encrypted using A's private key and decrypted using A's public key. This ensures that the original data is restored.

The important point is that this scheme provides authentication as well as protection. Authentication is provided because between points 4 and 5, the decryption is done by A •s public key, and only A could

at information is not accessed in an unauthorized manner. (Essentially controlling the Read operations.)
- Integrity: Ensuring that information is not amended or deleted in an unauthorized manner. (Essentially controlling the Write operations.)
- Availability: Ensuring that information is available to authorized users at the right time. (Essentially controlling the Read and Delete operations and ensuring fault recovery.)

» Security is concerned with tbe ability of the Operating System to enforce control over the storage and transportation of data in and between the objects that the Operating System supports.

» The major threats to security in any computing environment can be categorized as follows:
- Unauthorized use of service (Tapping).
- Unauthorized disclosure of information (Disclosure).
- Unauthorized alteration or deletion of inforination (Amendment).
- Unauthorized fabrication of infonnation (Fabrication).
- Denial of service to authorized users (Denial).

» Tapping and Disclosure are categorized as Passive tbreats while Amendment, Fabrication and Denial are categorized as Active tbreats.

)- A computer worm is a full program which spreads to other computers over a network and consumes the network resources to a large extent.

)- A virus is a pan of a program which causes direct harm to the system Dr can corrupt useful data and files.

)- Types of viruses: This classification is done based on what is affected or where the virus resides: Boot Sector Infectors, Memory Resident Infectors. File Specific Infectors, Command Processor Infectors, General Purpose Infectors,

» There are five well known methods in which a virus cao infect other programs: Append, Replace, Insert, Delete, Redirect.

» A virus can be detected, removed, or prevented

» The OS stores an Access Controll\fatro (ACM) in order to decide to which user to grant which access rights for which file. There are three methods for storing the ACM: I. Access Control List (Stored by Column), 2. Capability Lists (Stored by Rows), and 3. Hybrid of ACL and Capability List.

,. Cryptography is a technique of encoding (i.e. encrypting) and decoding (i.e. decrypting) messages, so that they are not understood by anybody except the sender and the intended recipient.

» When the original message in the plain test is encrypted it is called as cipber text.

,. The intelligence applied for the algorithm for encryption by the sender and for encryption by the receiver is called as the key.

» In the substitution cipher technique, the characters of a plain text message are replaced by other characters. numbers or symbols. Caesar Clpber is an example of substitution cipher.

» Transposition c1pber tec.bniques do not simply replace one alphabet witb another. They perform some permutation over the plain text alphabets. Some basic transposition cipher techniques are:
- Rail F ence T echnique
- Simple Columnar Transposition Technique

» Types of cryptograpby: Based on the number of keys used for encryption and decryption, cryptography can be classified into two categories: Secret key encryption, also called as private key symmetric encryption and public key encryption, also called as asymmetric cryptograpby.

» Using digital signature, a computer-generated messages or text can be signed digitally in the same way as a hard copy of the document. When a sender signs a message digitally, the digital signature establishes nOD-repudiation.

Ope n Syst ems Inter conn ectio n (OS!) defi nes the clem ents of secu rity in the follo wing term s:

- Confidentiality: Ensuring th

improvement. Applications which handle arrays and matrices are the candidates for speed increase. However, these will need to be written using languages which support parallel processing.

Fault Tolerance

One of the real attractions of parallel processing is the resulting high fault tolerance. If one processor fails, the process can be rescheduled to run on another processor. This may reduce the throughput of the system. However, the system will continue to be available. Critical applications like aircraft flight con- trol cannot tolerate any failure during flying periods. In such applications, multiple processors are kept to achieve improved availability and good performance.

Incremental Growth

A parallel computing system can be configured according to needs. As the demands on computing in- crease, the system can be upgraded by putting more processors. The upgradation cost will be minimal, since the CPU costs are only a small part of an entire system.

Cost Performance

Using today's technology, parallel computers with nearly the same power as CRA Y machines can be configured at nearly I/IOth the cost of those huge machines. The maintenance costs are also lower during the life of the system, since most of the components are available off the shelf.

Human beings think in a sequential manner, one thing after the other. Hence, it is very difficult for us to write programs which can work strictly in parallel. In today's languages, many programming constructs can be handled in parallel. A good example will be a "DO Loop" given below:

```
DO 1001= I, 10
A(l) = B(l) • C(I)
100 CONTINUE
```

The above example is a sample FORTRAN code which is very common in most applications. This example can be rewritten as shown below:

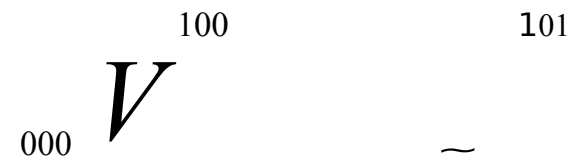
```
A(I) = B(I)·
C(I) A(2) = B(2)
• C(2) A(3) = B(3)
• C(3)
A(4) = B(4) *
C(4)
A(5) = B(5) •
C(5) A(6) = B(6)
* C(6) A(7) = 8(7)
* C(7)
A(8) = B(8) *
C(8) A(9) = B(9)
* C(9)

A(IO) = 8(10)*
C(IO)
```

Each line of computation can then be processed by a separate processor.

10.7.3 Hypercubes

Hypercube implies a parallel computer architecture in which, if there are *1/* processors in a system, each is connected to log *n* to the base of 2 neighbouring processors. An 8-node hypercube is shown in Fig.
10.3. In such an 8-processor system, l-processor is connected to three other processors. The number of interconnections each processor has with its neighbours is known as the dimension of the system. The
8-processor system in the above example is known as a 3-dimensional hypercube.

The advantage of this type of connections is that the delay in communication from one processor 10 another increases logarithmically to the total number of processors instead of increasing proportionately. Each processor is a full computer with CPU, memory and other *110* devices. Today, a lot of research work is done around such systems. These computers are also known as massh'ely parallel computer systems. Thirty-two and 64 computer hypercubes arc commonly available. Current research is on to create 16-dimensional hypercubes with 65,536 processors.



c.:; Fig. 10.3 An eight-node hypercube

An interesting property ofbypercubes is that an /I dimensional hypercube is a proper subset of an *n* + I dimensional hypercube. These are recursive structures and are found to be highly suitable for carrying out recursive computations.

The primary aim of an Operating System is to manage the available hardware resources. In a single processor Operating System, these resources were memory, file system and lIO devices. In a parallel processor system, there is a new resource to be managed-multiple processors. This new resource, instead of simplifying matters, complicates the issues further.

In previous chapters we have studied in depth the working of Operating Systems. In a single processor system, there is only one process running at any time. All other processes are ready to run, waiting for some resources to be freed, sleeping or waiting for some process to complete before this one can proceed. In a multiprocessor Operating System, to be efficient, the number of running processes should be equal to the number of processors available in the system. How can this be achieved? We can achieve it in three ways as follows:

- With separate Operating System running on each processor.
- With one master Operating System controlling other slave processors.
- With symmetric Operating System.

DGIUX is an advanced, redesigned and re-implemented version of the U1\TJX Operating System providing compliance to a wide variety of prominent industry standards for the A ViiON product line. The AViiON product line includes servers, large multi-processor systems and workstations, based on the Motorola 88000 Reduced Instruction Set Computer (RISC) processor.

DGIUX supports fully concurrent *VO* using its multiprocess technology. Symmetric Multiprocessing (Sft'IP) in DGIUX is transparent to applications. This ensures that the applications need not be modified when upgrading to a larger system. The kernel in DGIUX is pre-emptible and, therefore, the benefits from the multiprocessor implemented with SMP in DGIUX are also available to the single-processor operation. DGIUX provides a finegain locking mechanism. Many kernel services run as system processes.

In multi-processing A ViiON systems, resources such as memory and peripheral devices are shared by all Operating System and user processes. User and kernel processes can run on any physical CPU, and after a user or system process is suspended, it can be restarted and run on any available physical CPU. Concurrency has been extended in the kernel to various things like *TCP/IP,* File System, and *1/0* operations, providing true SMP throughput that takes advantage of added CPU and memory resources.

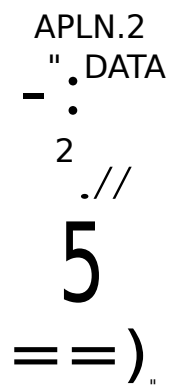
On most contemporary implementations ofSMP, the device drivers and the File System have not been made multi-processor (MP) safe. I/O becomes the bottleneck on these systems because control for I/O has to be driven from a master or global CPU rather than any CPU, or because *1I0* is allowed only to one CPU at a time. DG,UX has none of these restrictions. The software model ofSMP in the DGIUX kernel is fully reentrant, multithreaded, and pre-emptible. SMP on DGIUX ensures that:

(i) Any kernel process can execute on any processor.
(ii) Any user process can execute on any processor.
(iii) More than one user can execute in the kernel simultaneously.
(iv) All I/O can be initiated and completed on any processor, including starting an I/O operation on one processor and completing it on another.

Virtual Processor (VP) paradigm is implemented through S1\11'on DGfllX. A consistent view of memory and I/O from all processes is ensured by the virtual memory management and 88000 hardware. This paradigm hides the acrual number of processors above the kernel and system call level. All VP synchronization is optimized with a large number and variety of counters, sequencers and locks. From outside the entire VP subsystem, all active processes appear to be executing in parallel, but are in fact being multiplexed among the physical processors.

Distributed process tables are implemented in DGIUX to avoid global data sets and their associated problems. A process index is assigned to each active process and each smaller process table can be individually locked by code in the subsystem owning it. A short-term scheduler does not manage, interpret, or enforce policy. Its only job is to quickly multiplex VPs to Physical Processors.

A kernel VP pool is created at init time to execute kernel code only with a number of permanently bound daemon processes that are always ,ina run state. A process VP pool is heuristically assigned by the Medium Term Scheduler (MTS). The MTS takes a longer term view of scheduling priorities by evaluating process attributes such as interactivity, memory paging behaviour and percentage of CPU lime recently allocated, among other metrics. The MTS modifies the list of candidate processes and their priorities, which in rum are scheduled to run on the virtual processors. The MTS multiplexes processes


APLN.2
DATA

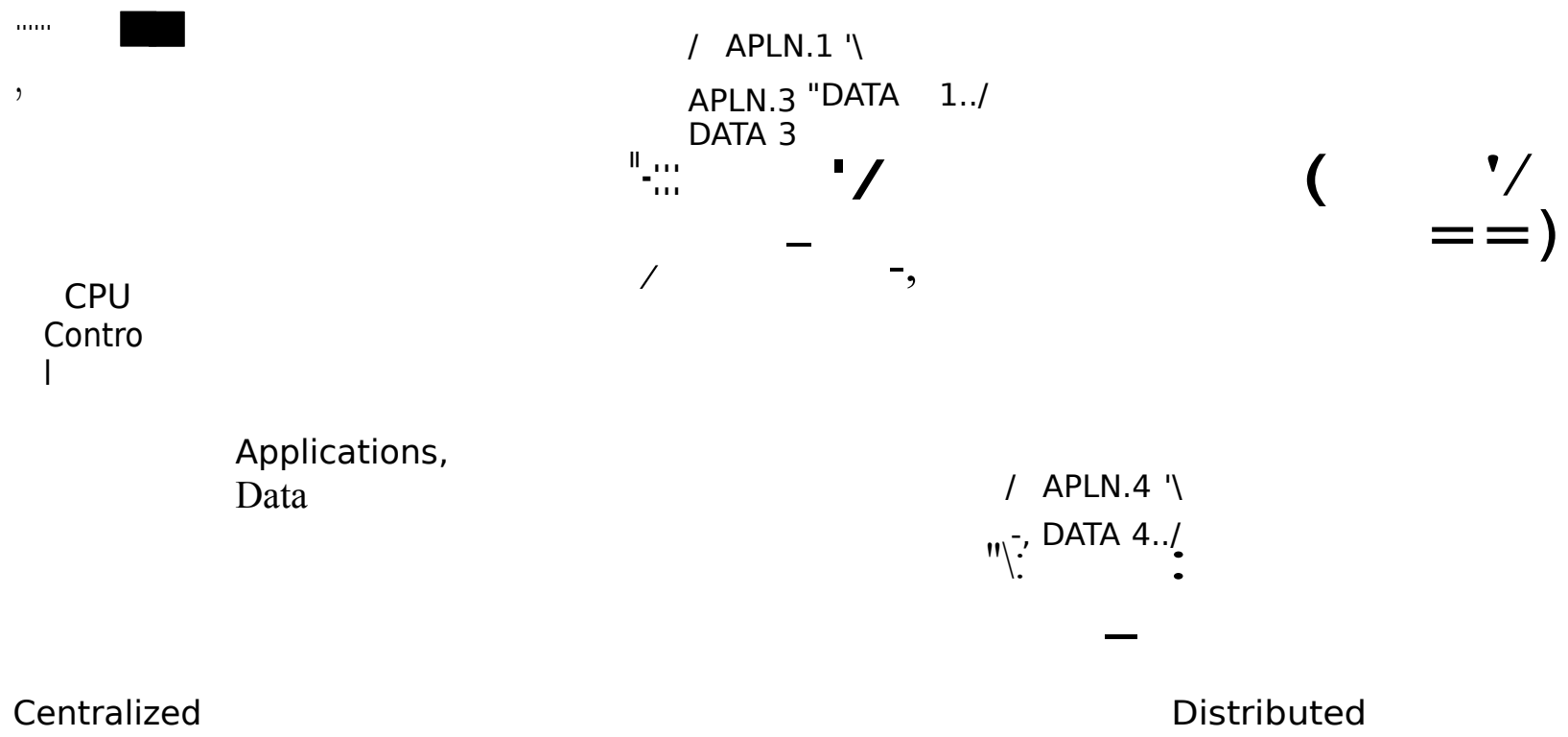


Fig. 11.1 Centralized vs distributed processing

In vertical or hierarchical distribution, there are various levels of functionality. These levels normally reflect the levels in an organization. For instance. a division of a company may have a number of zones and each zone in tum may have a number of branches. In such a case, a number of computers at the branch level will carry out the functions of various branches and send summaries to their higher level (zones). The computers at the branch level can be networked together, if data sharing is required. This is shown in Fig. 11.2. The computers at the zonal level also can be networked together, and they can share each other's information as depicted again by Fig. 11.2.

Fig. 11.2 Hierarchical distribution

(b) If an AP makes a system call which is not related to the Input/Output functions, ii is directly handled by the LOS running on the client workstation as shown in the figure.

(c) Normally, in a non- NOS environment, all *VO* calls also are made by the AP to the LOS (Operating System running on the machine). However, in the NOS environment. this is not true. In such a case, the call is made to the Redirection Software which is a part of NOS. This is shown in the figure. The AP making this *1/0* call has to be aware the data location (local or remote) and accordingly has to make the request to LOS or NOS. Hence, NOS cannot be said to implement location transparency completely.

(d) Different NOS implementations employ different methods to differentiate the calls for the local and remote data. For instance, in NetWare. if you use a drive number which is F: or above for any directory, it assumes that the data is remote. Using such a technique, the Redirection Software either sends the request to LOS if the request is local, or sends it to the remote server if the request is remote. In NetWare, pieces of software called NET3 or NET4 carry out this redirection func- tion.

(c) [f a request is to be made for remote data. it has to traverse as a Remote Procedure Call (RPC) from thectient workstation to the server. After this, the requested data on the server has to traverse back to the client workstation. In this case, a piece of software has to ensure that both these messages (the request for the data and the actual data) are transmitted correctly to the desired nodes. This piece of software is called Communication Management Software. This also is a part of NOS and has to reside on both the client workstation as well as the server. This is also shown in the figure.

This software has to implement packetizing, routing, errorlflow control functions. etc.: and therefore, has to carry out the tasks at various levels of OSI reference model. In NetWare, SPX, lPX, NIC.LAN and some other modules carry out these functions. We will call all these functions together by the name Communication Management Software. The function of this software is essentially to make sure that a message is communicated between the client and the server without any error.

(f) If the requested data is remote, the Redirection Software on the client workstation passes the request to the Communication Management Software on the client workstation as shown in the figure.

(g) The Communication Management Software on the client workstation sends a request across the network to the server in the form of RPC. Ultimately, bits traverse over the communication medium.

(h) The Communication Management Software on the server receives the request and ill turn, makes a request to the Network Services Software residing on the server, as shown in the figure. This, again, is a part of NOS. This module is responsible for services on the network for sharable resources such as disks, files. databases and printers. This module will receive many requests from different clients. It will then generate a task for each of these requests. It will schedule these tasks and service them.

This module will bave to have multiple *1/0* buffers to store data for these dif erent tasks. The NOS will have to implement some kind of multitasking to achieve this. LOS such as DOS did not have to do this. Network Services Software has to implement access control and protection mechanisms for the

Case (ii) discussed above. In this case, instead of NetWare carrying out the functions of multitasking, a multitasking LOS already running on the server can be used to handle multiple simultaneous user requests. There is no sense in using the multitasking capabilities of NetWare in addition. In Case (ii), NetWare was also responsible for the Information Management functions. It used its own file system and device management routines. It used a set of its well-defined system calls to handle various functions required. All these functions in the present case are provided by the LOS such as UNIX, VMS Or *AOSI* VS running on the server. Hence, on the server, all modules of NetWare need not be running.

However. you will need an interface module (a part of NetWare) which will receive the remote request along with the parameters, It will then find out whicb system call(s) of the LOS running on the server have to be executed and in which order, to fulfil the request. This module will then need to pick up the parameters coming in from the client workstation and use them to load various CPU registers of the server machine according to the expectations of the LOS running on the server.

After the request is fulfilled, the LOS running on the server will set certain CPU registers of the server to denote the success/failure of the operation. The interface module of NetWare will then need to transport the actual data (in case of success) as well as the returned parameter to the interface module of NetWare running on the client workstation which, in tum. will load the particular CPU register on the client workstation to denote the success/failure. The AP on the client workstation can then resume as if all these operations were local and proceed as per instruction 4 in Fig.II.S.

The interface module of NetWare therefore, carries out the function of translating a remote request into a form understood by the LOS running on the server. Therefore, the interface module of NetWare will be different for each different Operating System running on the server. If Novell tomorrow wants to allow a specitic new machine M running its own Operating System to act as a server, it will have to write this interface software anew to carry out this translation before it cao declare something like: "Now Novell can support VMS on the VAX as the server," This interface software for this translation is not quite trivial, as the number of system calls to be translated, their formats, the way they expect the input parameters and the way they return the output parameters might differ quite substantially from one Operating System (o the other.

(b) Communications Management Software

A part of any NOS will have to run on both the client workstation as well as the server. This part is needed in all the cases discussed above. It is responsible for Communications Management. Basically this software implements the necessary protocols at various levels of OSI reference. It is concerned with delivering any message (requests for data, the actual data or acknowledgments) to the destination without any error. We will study various protocols in later sections.

The ordinary Operating System does not perform these functions. and therefore, it has to depend on a separate piece of software (such as SNA). But in the NOS environment, Communications Management Software is a part of the NOS. This software has to reside on all the nodes including the server. Without this, no communication will be possible. The formats and the protocols of all the messages and packets have to be exactly the same. The Communications Management Software can consist of a number of modules corresponding to different OSI layers which could reside on the same or different pieces of hardware, as we shall see later.

sible for all the communications as well, it has to have a means of handling these deadlocks too! This was net necessary in the nun-distributed environments.

Deadlock prevention in a distributed environment uses similar methods as used in a centralized system. These methods normally try to prevent the circular wait condition, or the hold and wait condition, Beth of these methods require a process to determine the resource requirements in advance, and there- fore, are nut very satisfactory in all practical situations.

Some of the partial solutions offered to guarantee the proper and consistent al location of distributed resources are listed below:

- Random graphs.
- Decision theory.
- Garnes theory/Team theory.
- Divide and conquer.
- Best effort.
- Bay scan theory.
- Stochastic learning automate.

A detailed discussion on these solutions is beyond the scope of the current text.

Process migration is central to any DOS-based process management. In simple terms, this is the trans- fer of adequate amount of ioformation and state of a process from one host to another, so that the process can execute on the target host.

Process migration is a very important aspect of distributed operating systems. This aspect deals with the decisions of transferring the state of a process from one computer to another computer (i.e. distribut- ing the process). Clearly, at least two computers are involved in process migration: the source and the destination. The source transfers the process to the destination, where it now executes. There couLd be an additional computer, the arbitrator, which may help in the process migration activity.

11.3.1 Need for Process Migration

What are the motivations for process migration? The chief ones are as follows:

Load sharing/balancing

There could be situations where a particular server computer in a distributed environment is overloaded with the client requests. AI the same time, the other servers in this distributed environment may be relatively idle, or underutilized, In sucb situations, we can enhance the overall performance by transfer- ring some of the load from the overloaded server to another server, which takes over from the overloaded server.

Availability

By making sure that more than one computer are involved in the processing of a client's request. we can guarantee continuous availability of resources. More specifically, we can set up the distributed environ- ment in such a way thatsome servers are available as backup. So, these redundant servers can take over from a server, which is either overloaded, or is on the verge of failure because of any errors.

similar to the message handling module discussed earlier runs to ensure packing/unpacking of parameters in the message in accordance with the packet/frame formats of the protocol used. and also to ensure error-free communication.

Notice that when the client process issues an RPC. it gets blocked. The interface process then takes over. completes the call and returns the results after which the client process becomes 'ready' again. This is exactly how a remote read operation in any NOS takes place. The interface modules at both the nodes take care of error-free communication between the client and the server nodes. This takes care of passing of parameters and transmitting the actual data between the nodes. The actual functioning of various modules will be clear from the figure.

There are some issues regarding the way the parameters are represented while passing. and the way their binding takes place. We will now discuss these in brief.

11.4.6 Parameter Representation

If the RPC is operating between processes running on two identical machines with the same Operating Systems. tbe way tbe parameters are passed will be identical within at least a given programming lan- guage. But we know that if the machine architecture (i.e. CPU registers, etc.) or the Operating System or the programming language differs. problems can arise. POJ instance. in the two systems. numbers or even the text could be represented differently. We also know that the word length can affect bow the param- eters are passed. In tbe earlier example of "file open" RPC. the three parameters will have to be sentfrom
a DOS machine with Intcl Stlxxx architecture to the server.

The server may be running DGIUX on tbe Aviion (M88000 based) machine which is a variation of UNIX. with its own system calls. Therefore. in order to execute the "file open" system call (now on the server machine), if ought expect the same three parameters in different formats and loaded 011 some specific CPU registers of Aviiou. In this case, we Deed some software layer to do the conversion re- quired. If a proper full-fledged communications software adhering to the OSI standards is used. the presentation layer is supposed to take care of all tbis. But then using such a sofrware has tremendous overheads. This is the reason, most of the RPC facilities themselves (interface module shown in Fig.
11.12) provide for this conversion routines. Hence, this interface module bas to be rewritten every time a new workstation with a new Operating System has to be supported on a network.

A recommended approach would be to devise a common standard format. Every interface module then is designed to have routines both to convert from/to its own formats to/from the standard format. Let us call these routines R I (to convert from its own internal forma! to the standard one) and R2 (to convert from tbe standard format to its own internal format). For each machine and its Operating System, both these routines will bave to be written. Both R I and R2 must also be present on each node as per this scheme. A sending interface then uses its own Rl routine to convert to the standard format, The receiving interface uses its own R2 routine to convert from the standard format to its own internal format which is then used to actually execute the remote procedure.

11.4.7 Ports

If a number of services are provided by a server node. normally a port number Is associated with a specific service. For instance. a port number 1154 could be associated with the function of listing the current users. Another port Dumber 2193 could be associated with opening a file. RPc uses these port numbers heavily. A port can be viewed as a specific service provided by an OS similar to a system call. This is devised to simplify the communications. The message sent by an RPC to the remote node contains

An OR.B infrastructure has two main pieces: the client and the server. The client ORB communicates with the'serverORB using a protocol, such as nop. Usually, the client ORB is called as the stub, and the server ORB is called as skeleton. The objects on the client and the server can be created in different programming languages, and yet they can communicate with each other. In order to facilitate this, a special language, called as Interface Deftnidon Language (IDL) is used. The IDL is a uniform wrapper over the underlying programming language, which hides the syntaxes and implementations (and therefore, the differences) of the usual programming languages.

11.6.1 Introduction

The network consists of a number of nodes. At each node, there may be some local fileslatabases. In the NOS environment, if a user/programmer wants to access a file from another node, he has to know the location of that file and explicitly specify it before he can make a request to transfer it to his node. In the GOS environment, this is not necessary. This gives it location transparency.

11.6.2 File Replication

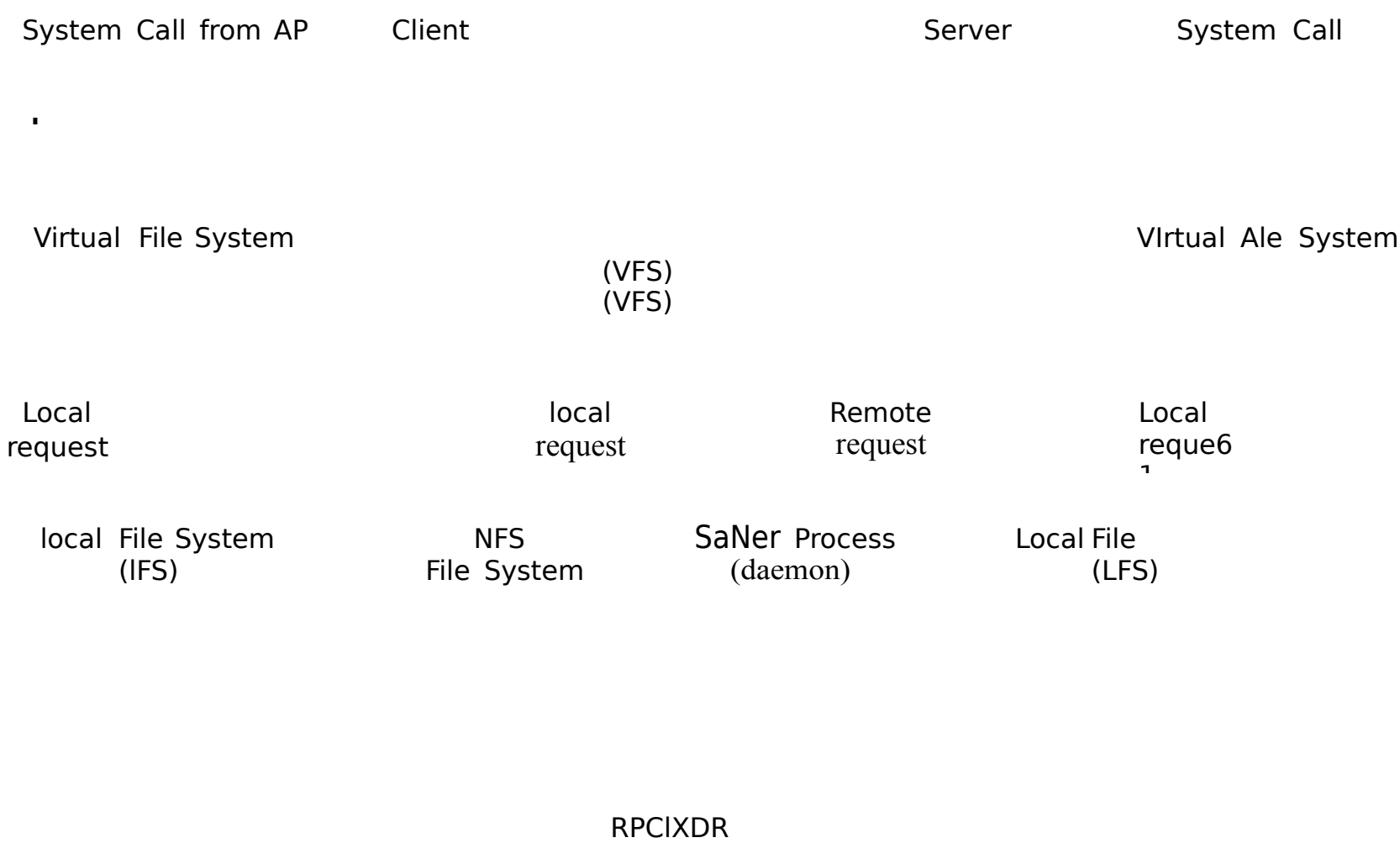
In the NOS environment, it is often advantageous to keep multiple copies of the same file on different nodes, so that the time to transfer the file is reduced substantially. The request for a file in this case is satisfied from the nearest node having the file. To implement this strategy, a study has to be undertaken to identify which node has been requesting for which file from which remote node and at what frequency. After this study, multiple copies of the same file can be maintained at the appropriate nodes.

The advantage of this approach is the reduction in the network load and the improvement in the performance. but the disadvantage is the increased storage requirement, and the need to maintain the consistency in all the copies of a file. If a node updates that file, the changes have to be propagated to all the copies of a file on all the nodes. Another disadvantage is the need to monitor the file replication strategy, as the pattern of requests for remote files may change with the passage of time.

11.6.3 Distributed File S.vstem

In a network, each node runs an LOS with its own file system. Let us call it the Local File System (LFS). LFS is responsible for allocating disk blocks to a file and for maintaining the allocation tables such as FAT. TheLFS provides various services such as "create a file", "delete a file", "read from a file" and "write to a file". It maintains the directory structure for all the local directories and files. It allows the user to change his working directory. list all the files in a local directory and implement the access controls on the local files and directories.

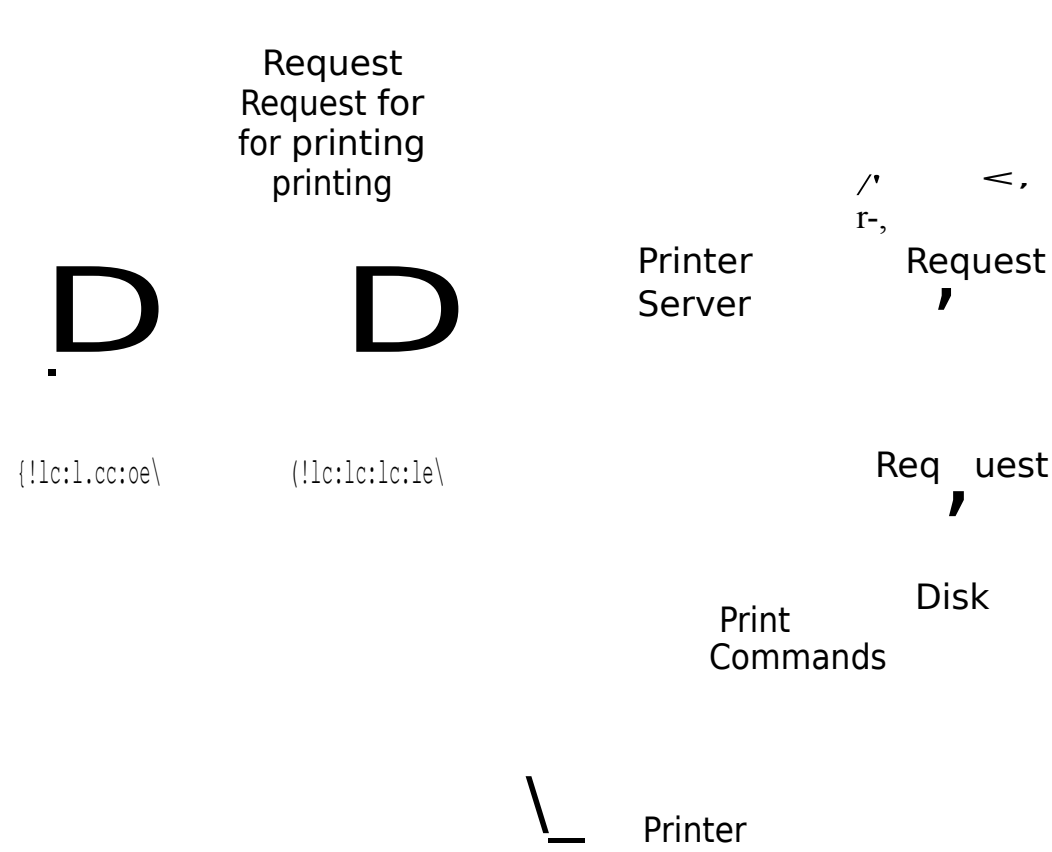
The problem arises when a user wants to carry out all these functions on a remote file. The Distrib· uted File System (DFS) aims at allowing precisely that. The ultimate aim is to allow any user to see the entire structures of files and directories on all the nodes put together as one composite hierarchy from his perspective. And an interesting point is that the perspectives of different users may, or almost certainly will. differ. The DFS has to cater to all these personalized views from within a global physical reality. Let us imagine a network of three nodes. Figure 11.18 depicts different directories at various nodes in this network. In this figure, a three-dimensional box represents a root directory at any node, a rectangle represents a directory or a subdirectory, and a circle represents a file.



~ Fig. 11.22 NFS components

happen. The first is called 'exporting'. This is a mechanism by whi~h all the server nodes make all the sharable file systems known to all other nodes. Usually, this process occurs when the machine is booted. The second thing is called 'mounting'. This is a process by which a client selects the remote file systems out of the exported and available file systems. The selected file systems are then attached to his local file system and a VFS is actually constructed. Very often, the remote file systems required by a client are constant and therefore, mounting also can occur at the boot time. A file on the client workstation keeps the information as to which file systems are to be mounted during booting. Another file on the server workstation keeps the information about which file systems can be mounted at other nodes (exponed). This file also keeps information about which nodes can and cannot access these file systems, and in what capacity.

Like mount, NFS also has an 'unmounf command which can be used by a client to disconnect a remote file system from the VFS.



~ Fig. 11.24 Printer server

A client workstation desiring to print something (say, a report) issues commands as usual. However, the data to be printed is not directly printed on the printer. Instead, it is redirected to the disk attached to the printer server. The data cannot be directly printed because the printer may be busy printing a report for another client workstation! Therefore. while the printer is busy. all other requests for the printer are queued. The spooler software keeps track of all these requests and services them one by one. The "report" stored on the disk is normally not stored exactly in the same way in which it is printed. Some coding' scheme is used to denote specific actions.

A specific control character may mean skipping a certain number of blank lines. Another could mean throwing a page. Yet another one could mean printing bold characters or with underlining. The control characters themselves are unprintable and hence, are not used in the actual report to be printed. This is the reason why these control characters are used in this coding scheme, This philosophy is the same as the one used in a spooler system. While printing a report. the server program interprets the control characters and generates corresponding print commands such as "skip a page",. "skip a line", etc. to actually carry out the printing.

In the client-based. or file server. environment, the LAN is principally used for sharing the peripherals. All the sharable data is stored on the file server and if a client workstation requires any data, the server sends down the entire tile to that client workstation, in most cases. The file server stores the files as only raw files or as a stream of bits and has no knowledge of chains, pointers or indexes interconnecting various data records to reflect their relationships. Hence, if a client wants a re.cord for a specific customer. the entire file is transferred from the server to the client workstation first, after Which the client

The compiler is responsible for generating the actual executable statements to search the indexes and data records. and retrieve die appropriate records which match the given conditions. This is obviously done according to the strategy suggested by the optimizer. These statements form the program which the task has to execute. The program contains call statements for reading/writing the actual data records, locking a record. etc. These are the actual system calls only. A database server can use the services (system calls) of the NOS to implement these. However. a database server can have its own 110 routines. bypassing the *110* calls of the NOS. This is done to improve upon the speed.

As discussed earlier. at the rime of executing a task. it can get 'blocked' after requesting an *110* routine, whereas it can be made 'ready' after the *110* is over. whereupon other instructions such as the ones for the index search in the memory can be executed. AU this is managed by the kernel of the database server.

Today. in the client-server environment, we have a number of products to handle the client side. Powerbuilder and Visual Basic are some of the examples. They provide a user-friendly Graphical User Intertace (GUL) for user interaction. They make database calls in a standard fashion as expected by the database server products such as Oracle, Sybase or Gupta SQ.L, Tbe latter can be called as examples of database engines which receive the database requests in a pre-specified standard format and service

them one by one as discussed earl

ier.

We can imagine that aU the issues that are found in a simple, lion-distributed environment, such as mutual exclusion, starvation and deadlock are quite possible in the case of a distributed environment In fact, me possibility of these happening is more here, as a number of entities are involved, which can lead to chaos. To add to the trouble. there is no global state. That is. there is no way for an operating system or a participating process to know about the overall state of all the processes. It can only know about its own state (i.e. local processes). For obtaining information about remote processes, it has to rely on the messages from communication with other processes, Worse yet, these states reflect the position that was in the past. Although tbis *past* can be as Jess as 1 second, it can have very serious consequences in business and other applications.

For instance. suppose that Process A gets me information from ıı remote Process, B that the balance in an account is USD 2000. at 11.59 a.m, Therefore, Process A considers that as a sufficient balance for performing a withdrawal transaction of USD l500, and goes ahead. Unfortunately, at 12.00 pm, Process B also performs a withdrawal transaction ofUSD 1800 on the same account. Quite clearly, this is a very serious problem of concurrency. In a local system, we can somehow control it. with the help of database transactions. How do we take care of it in a remote transaction?

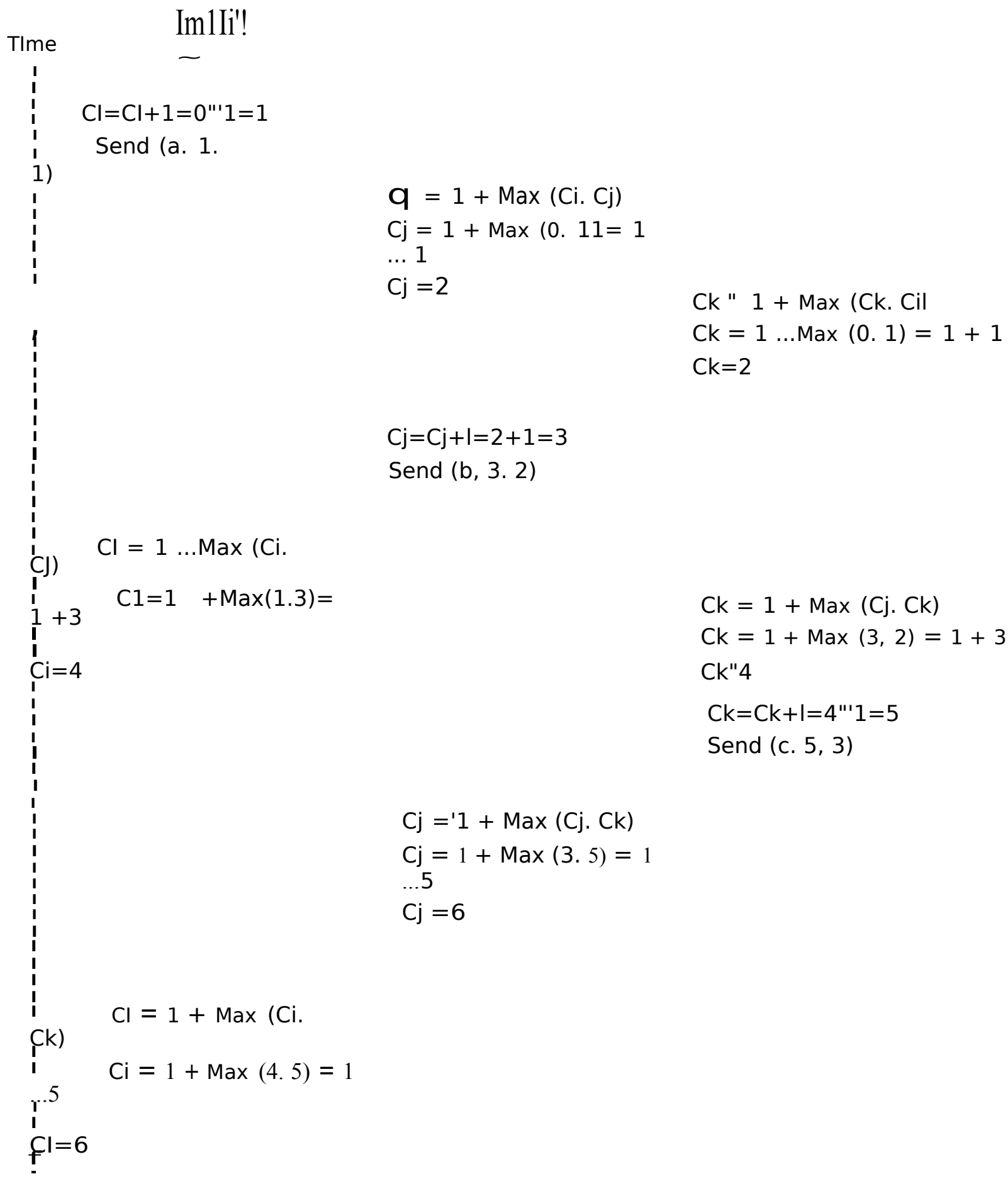
To solve this, there are many approaches, of which we shall discuss two,

Distributed SllapshOI Algorithn,

The distributed snapshot algorithm is used 10 record a consistent global state. Here, the basic assump- tion is that the messages are correctly received at the destination, without any failures (similar to the way TCP guarantees message delivery- without loss, in the same order as the message was sent. and without

earlier), it defers its reply. The reason for this is simple. This is because the process (rightlyj believes that it should get a higher priority than the process, which bas sent it a message.

Does this not lead to deadlock? What if all the processes believe that it is their turn to enter into the critical section at the same time? Interestingly, this situation will never arise, because we know that a process will defer its reply only if its timestamp value is smaller than tbat contained inside the received message. Since timestamps are arranged in a sequentially increasing manner very strictly, at any given point, only one process would have the smallest timestamp value. All other processes would have replied back to this process (as their timestamp values would be higher than the timestamp value of this process). As such, that process can enter its critical section, and there is no fear of deadlocks.



O. Fig. 11.34 Example of Lamport algorithm with three processes

Frame type. frame number. frame size. source address, destination address, CRC and flag are the "overhead" fields which have to be added to the actual data bits. This is a necessary evil as we shall see! It must be remembered that the frame can be a data frame or an acknowledgment frame denoted by the "flag". This is valid only if the system follows a convention of acknowledging the messages.

The address has three components. A network could be constructed by having multipl.e LANs as depicted in Fig. 11.39.

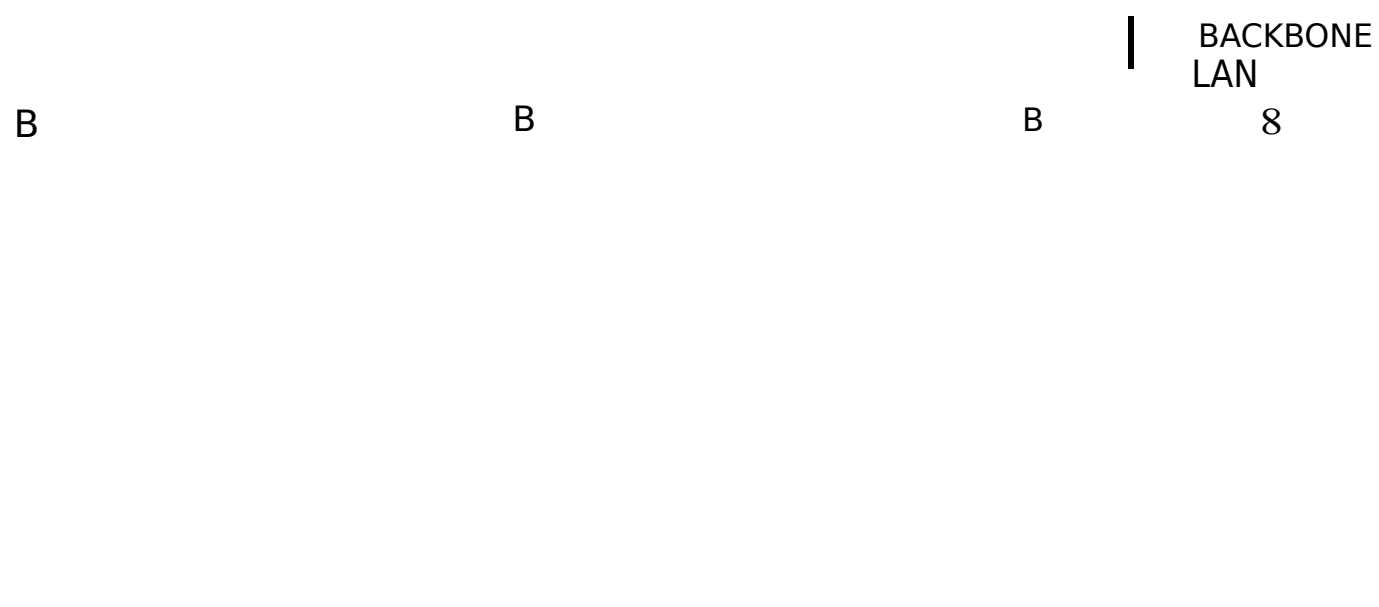


Fig. 11.39 Multiple LANs connected by bridges

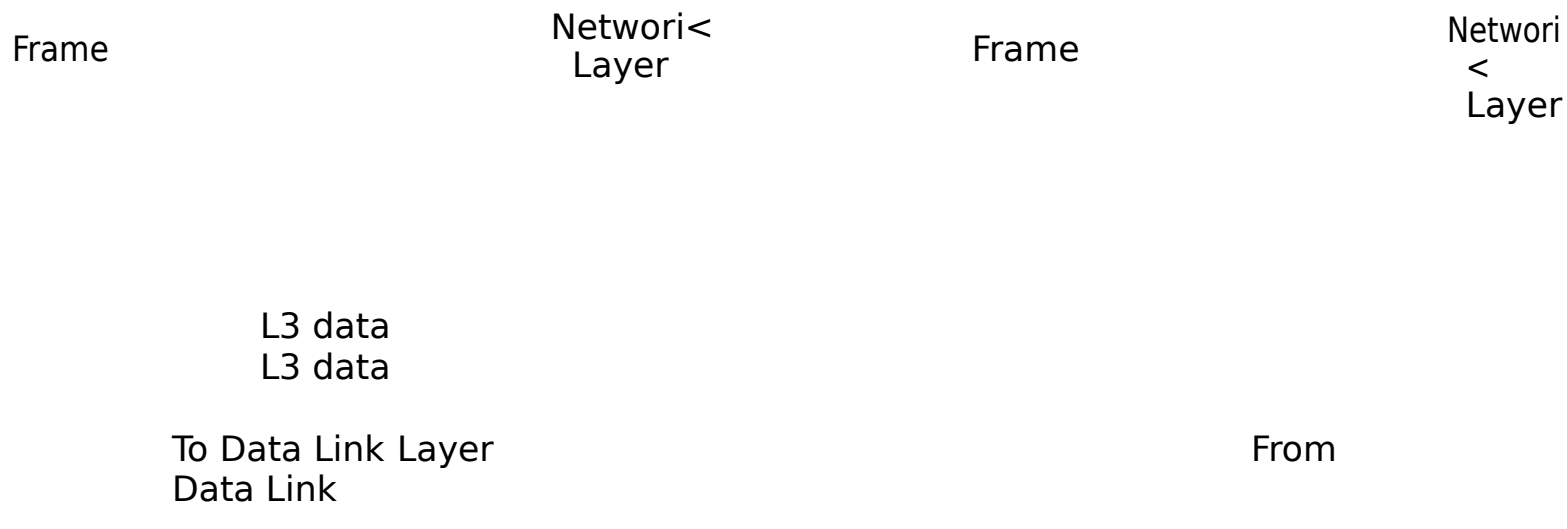
Figure 11.39 shows three networks, each with a number of nodes. Each node could be running a multiuser Operating System. It could be executing various processes which are associated with different sockets-one for each socket-shown in the diagram. Therefore, the address consists of three components, viz. network number, node number and the socket number. Using the source and destination addresses, any process running on one node can send a message to any process running on any other node on the same or a different network.

A node sends messages to another for any of the following reasons:

(a) Open a communication session with another node.
(b) Send an actual data frame to another node.
(c) Acknowledge of a frame received correctly.
(d) Request for the retransmission of a frame.
(e) Broadcast a message to all nodes.
(f) Close a communication session.

From Transport Layer

To Transport Layer



~ Fig. 11.53 Network layer between adjacent nodes

To summarize, the network layer performs the following functions:

- Routing: As discussed earlier.
- Congestion control: As discussed before.
- Logical addressing: Source and destination logical addresses (e.g. lP addresses).
- Address transformations: Interpreting logical addresses to get their physical equivalent (e.g. ARP protocol). We shall discuss this in detail in the next section of the book.
- Accounting and billing: As discussed before.
- .Source to destination error-free delivery of a packet.

11.16.7 Transport Layer

Transport layer is the first end-to-end layer as shown in Fig. 11.54. All the lower layers were the protocols between the adjacent nodes. Therefore, a header at the transport layer contains information that helps to send the message to the corresponding layer at the destination node, although the message broken into packets may travel through a number of intermediate nodes. As we know, each end node may be running several processes (may be for several users through several terminals). The transport layer ensures that the complete message arrives at the destination, and in the proper order and is passed on to the proper application. The transport layer takes care of error control and flow control, both at the source and at the destination for the entire message, rather than only for a packet.

As we know, these days, a computer can run many applications at the same time. All these applications could need communication with the same or different remote computers at the same time. For example, suppose we have two computers A and B. Let us say A hosts a file server, in which B is interested. Similarly, suppose another messaging application on A wants to send a message to B. Since the two different applications want to communicate with their counterparts on remote computers at the same time, it is very essential that a communication channel between not only the two computers must be established, but also between the respective applications on the two computers. This is the job of the transport layer. It enables communication between two applications residing on different computers.

- Ageot
- Basic Input-Output System
- Backbone LAN
- Blocking Message Passing

Bridges

Buffering

Cache Consistency

Cache Management

Carrier Sensing

Centralized Data

Call by reference

Call by value

Centralized Processing

Client-Server Computing

Collision Sensing

Communication Management Software

Compiler

Computation Migration

Cost-based Optimizer

Cyclic Redundancy Check (CRC)

Daemon

Data Migration

Database Engines

Database Server

Datagram

Diskless Workstation

Distributed Application

Distributed Control

Distributed Data

Distributed File System

Distributed Processing

Exporting

File Handle

File Server

Flow Control

Frames

Global Operating Systems

Global Priorities

Hierarchical Distribution

Horizontal Distribution

Interfaces

Interprocess Communications

Layer

Load Balancing

Local Area

Network Local File System

Local

Independence Local Operating System

Local Priorities Location Transparency

Logical Link Control

Longitudinal Redundancy Check (LRC)

Media Access and Control

Message

Message Passing

Metropolitan Area Network (MAN)

Mounting

Multitasking

Multitasking Server with Scheduling Capability

Multithreading

Network File System

Network Operating System

N1Cdriver

Non-blocking Message Passing

Optimizer

Packets

Parser

Partitioned Data

PhantomlFalse Deadlock

Physical Chunclc:

Ports

Process Migration

Protocols

Redirection Software

Reliable Message Passing

Reliable Service

Remote File System

Remote Procedure Call

Replicated Data

Selective Retransmission

Server

Session

Session Recovery

Token Bus/Ring

Token Passing

Transceiver

- Unmount
- Vertical Distribution
- Virtual Circuit
- Wide Area Network (WAN)
- Unreliable Service
- Vertical Redundancy Check (VRC)
- Virtual File System

Windows NT/2000 -A CaseStudy

12

This chapter begins with a brief history of Windows Opera ling Systems. It then depicts the differences between Windows NTf2oo0 and their cousins Windows 9x1Me.

IL then discusses the unique features of Windows 2000 Operating Systems including the Win 32 API aud Windows Registry. The chapter then goes on to describe organization of Windows 2000 Operating System.

The chapter explores process management. memory management and file management in detail. It concludes by outlining importaru features of NTFS and explaining security features of Windows 2000 in detail.

The history of **Windows 2000** can be traced back to the 1980s. **ffiM** had developed the world's first true microcomputer, the **ffiM** PC. Also called as the desktop computer, this microcomputer was based on the 8088 microprocessor. Microsoft wrote the Operating System for this computer. This Operating System was called as Disk Operating System (DOS). The microcomputer quickly became very successful. This also meant that DOS was also becoming popular.

DOS (version 1.0) was then a very basic Operating System. It consisted of just 8K of code in the memory. This was enhanced to 24K mOOS 2.0. When **ffiM** developed the PC-AT (AT stands for *Advanced Technology)* in 1986 to work with Intel's 80286 microprocessor, Microsoft increased the. DOS (now version 3.0) memory size to 36K.

.All this while, DOS was a traditional command-I me Operating System. This meant that users had to type commands to issue instructions to the Operating System. For instance, to copy a file a.dat with another name b.dat, the user would need to issue an instruction such as *copy a.dat b.dat* at the DOS command prompt. AboUI the same time, people were becoming more interested in Graphical User Interface (GUT). A company called as Apple Computers had developed a GUI Operating System, called as Lisa (which was a precursor to the Apple Macintosh), This meant that instead of having to type

information via the memory and resources that they share. As an aside, if it is a multiprocessor system, more than one thread can execute at the same time.

We had remarked that one of the goals of Windows NT was to provide support for various Operating Systems. Let us understand what this means, and how it works. Consider an older Operating System, such as the POSIX standard or 16-bit Windows (i.e. Windows 3.1). These Operating Systems do not have the concept of threads. Does it not contradict our earlier argument that a process in Windows NT begins by spawning a thread"?It may seem so. However, it is not true. What happens is this: When such an Operating System client uses Windows NT, and wants it to create a process, Windows NT creates a process (which in tum, spawns a thread, as usual) and returns the process handle back to the client. The fact that a thread was also created remains hidden from the client. The needs of the client (i.e. a process id) thus get satisfied, without compromising on the internal working of Windows NT (i.e. a process must spawn a thread).

The earlier versions of Windows (e.g, Windows 3.11, Windows 95) used to support the non-preemptive multitasking. Many times, this used to lead to serious problems, as one misbehaving application couId bring the entire computer down on its knees. In contrast, Windows NT introduced the concept of preemptive multitasking. Erring applications could no longer bring the whole system to a halt. Moreover, each process now got its own address space. Therefore, problems in the area of one process could not cause damage to other processes.

12.3.1 Process Synchronization

Windows NT provides rich features for synchronization of processes and threads. As we have mentioned earlier, Windows NT treats processes and threads as objects. In addition to these, Windows NT defines several other objects, which are specific to synchronization. They are: event, event pair, semaphore, timer, and mutant. Table L2.3 summarizes these objects and their description.

Table 12_3 *WindOIVS NT'synchronization objects*

Object	*Description*

Process	Running instance of a program, which has us own address space and resources. It terminates when the last running thread within that process is terminated.
Thread	This is a serially executable entity within a process. Tbe synchronizationis over as soon as it terminates.
File	This is tbe instance of an open file or device. After the *VO* operation is complete, the synchronizationeffect is lost.
Event	This is an alert that a system event bas occurred.
Event pair	This is reserved only for client-server communication. For instance. a client thread could communicate that it bas sent a message 10 a server thread. This causes tbe synchronization (i.e. a transaction)to start 10 ensure thai the server receives the message correctly.
Semaphore	This is tbe counter, which specifies the number of threads thai can simultaneously use a resource. When the semaphorecounter becomes zero, the synchronizationpoint is reacbed. The system counter, which records the passage of time, This is useful for CPU scheduling synchronizationactivities to determine time slices, expiration times. etc.
Timer	
Mutant	This object is similar to the MUTEX object, which provides mutual exclusion capabilities.

12.4.2 \Viodows Registry

Every Operating System must keep a lot of information about the files on the disk. registered programs, authorized users. and so on. There are different mechanisms for storing this information. The early versions of Windows (e.g. Windows 3.1/3.11) used to maintain a lot of initialization files (called as .ini files) to store this information. This was quite clumsy and chaotic.

Beginning with Windows 95. this information is now stored in a centralized database, called as the Windows regist.ry. Since it contains all the critical information about a given computer. the registry is perhaps the singlemost important aspect of any Windows installation. If it is corrupted. the computer may become completely or partially useless. Unfortunately, to add to the problems, working with registry is not easy. The nomenclature is quite tough to understand and work with.

The basic idea of the registry is actually quite similar to that of a directory of disk files found on any computer. Just as we have the root directory. which contains sub-directories, sub-sub-directories, and so on. the Windows registry is also organized in a similar hierarchical fashion. Each high-level directory is called as a key. All top-level directories begin with the word *HKEY* (handle to a key). For instance, the current information about the currently logged on user is stored in a top-level directory called as *HKEY_CURRENT_USER*. Sub-directories have names that signify better meanings. The lowest-level entries contain the actual information, and are called as values. Each value consists of three parts: name, type and data. For example, a value could indicate that (a) this is the information regarding the default window position *(llalne),* (b) it is an integer *(type)* and (c) it is equal to 2eOO3fc *(data).*

Unless the user is well conversant with the concepts of the registry and the implications of changing any values therein, it is strongly recommended not to play around with it. One should be safe viewing its contents, though. Generally, the command *regedit* or *regedit32* causes the contents of the registry to be displayed on the screen. A sample screenshot of the registry is shown in Fig. 12.4.

;;';'~ Fig. 12.4 Windows registry

Power manager The main task of this component is to save power. When a computer is not in use for some time, this component detects this fact, and turns the monitor and disk dri ves off. Additionally, on laptops, the power manager monitors the battery, and when it realizes that the battery is going to become dry, it alerts the user, so that the user can save open files and other pieces of data and gracefully shut down the computer, rather than causing an abrupt shut down.

Local Procedure ('aU (LP{') manager This component provides for inter-process communication between processes and the sub-systems. Notably, the LPC mechanism is also used for some system calls. Therefore, it is imperati ve that this mechanism is efficient.

System Interface

The system interface portion facilitates communication between user application programs and the Windows 2000 kernel. Its main job is to accept Win32 API calls, and map them on to the corresponding Windows 2000 system calls. This marks the interface between the kernel area and the user area. This also frees the application programs from worrying about the system call details, and only concentrate on the higher Win32 API calls.

Windows 2()(}() and objects

Objects are perhaps the most important concept in Windows 2000. They provide a standardized and reliable interface to all the Operating System resources and data structures, such as processes, threads, semaphores, etc. Some of the most significant aspects of this are as follows:

- The naming convention and access mechanism for all the objects is the same. This is in the form of handles to objects.
- Since all the accesses to any object have to go via the object manager, it becomes easy [O centralize the security checks and constraints. This minimizes the chances of bypassing these checks.
- Since the object manager is the single interface for the creation or release of objects, it is far easier to keep track of used and unused objects.
- This uniform mechanism for object management makes it simple to handle resource requests and its fulfilment.
- Object sharing among the processes becomes quite simple.

Windows 2000 views objects as consecutive words in the memory. It is considered as a data structure in the main memory. Each object has an object name, the directory in which it resides, security information, and a list of processes with open handles to the object. The overall structure of an object is shown in Fig. 12.6.

As we can see, each object has two portions: a header (which contains information COIDIDonto all object types); and object-specific data (which contains data specific to that object). The header of the object mainly contains information such as its name, directory, access rights, processes that access it, etc. Additionally, it also specifies the cost involved in using this object. For instance, if a queue object costs one point and a process has an upper limit of 1() points (which it can consume), the process can lise at the most I() different queues. This also helps in putting checks on the usage of the various Operating System resources.

12.4.4 Process Management in Windows 2000

Basic Concepts

Windows 2000 supports the concept of processes, like any other Operating System. However, it also introduces a few more concepts. Let us discuss all of them.

- A job in Windows 2000 is a collection of processes. Windows 2000 manages a job with a view of imposing restrictions, such as the maximum number of processes per job. the total CPU time available for each process within a job, and the security mechanisms for the job as a whole (e.g, whether any process within ajob can acquire administrative privileges).
- In Windows 2000, a process is a running instance of a program, which demands and uses system resources. such as main memory. registers. CPU time. etc. This is quite similar to processes in any other Operating System. Every process is provided with a 4GB address space, half of which is reserved for the Operating System. A Win32 API call can be used to create a process. This call accepts the executable file name associated with the process, which specifies the initial contents of the process's address space.
- A process can create one or more threads. In fact, every process must start by creating a thread. A thread is the basis for Windows 2000 scheduling, because it always selects a thread for execution. never a process. A thread has one of the possible states (ready. running. blocked, etc.), which a process does not have. Every thread has a unique thread id, which is different from the parent process id. A thread runs in the address space of its parent process. When a thread finishes its job, it can exit. When all the threads in a process are dead. the process itself gets terminated.
- Windows 2000 goes one step beyond what most Operating Systems do. It allows a thread to create one or more fibres, just as one process can create many threads. The reason for this is the overheads associated with thread switching. As we know, to switch from one thread to another, the Operating System must save the context of the running thread, mark it as *blocked,* load another thread and its parameters, mark it as *running,* and hand over the CPU control to this new thread. This is quite resource consuming. For this reason, a Windows *2900* user process can spawn one or more fibres, and allow them to execute in parallel w/rhor« making any explicit system calls. Windows 2000 provides Win32 API calls to create and manage fibres. A detailed discussion of how this works is beyond the scope of the current text.

A summary of these ideas is shown in Fig. 12.9.

Windows does not have a concept of process hierarchy. That is, all processes are treated as

equal. There is no concept of a parent process or a child process.

Implementation of process management

Windows 2000 provides a Win32 API call. *Createi'rocess*. This caU can be used to create a new process. This function call expects 10 parameters, each of which has many options. It should be quite evident that this sort of design is quite complicated when we compare

Process

• .,..... Fig. 12.9 Concepts of job, process, thread and fiber in Windows 2000

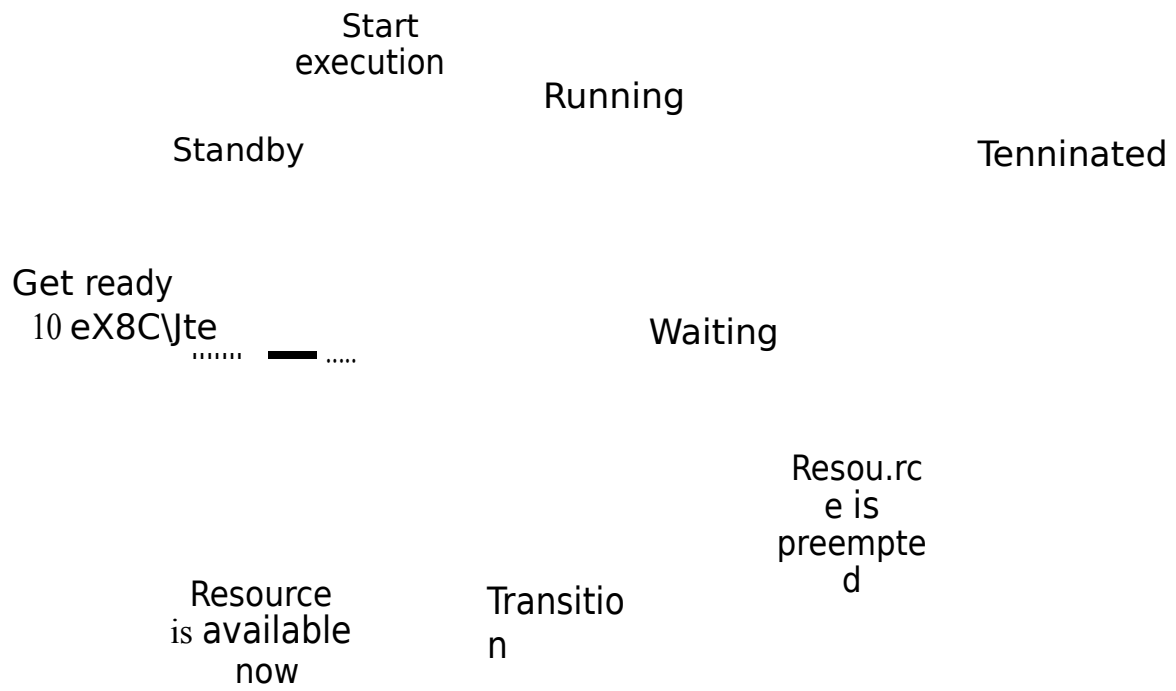


Fig. 12.10 Windows 2000 thread state transition

12.4.5 Memory Management in Windows 2000

As we have mentioned previously, each process in Windows 2000 is allocated a 4GB virtual address space. Windows reserves about half of this address space for its own storage. Virtual addresses are

bits long.

Each virtual page can take one of the three possible states: free, reserved, or committed. *Afree page* is the one iliat is not used currently. A *free page* becomes a *committed page* when code or data are mapped on to ir, Of course, a *committed page* can be in the main memory, or on the disk. If a reference is made to a *committed page,* it is directly accessed from the main memory, if it is available. Otherwise, a page fault occurs, and the page is brought into the main memory from the disk. A page can acquire the status of being a *reserved page,* in which case, it cannot be used by any other process than the one, 'which reserved it.

The Win32 API provides many functions for memory management. These functions allow a program to allocate, free, query and protect areas of the overall virtual address space. Some of these calls are tabulated in Table 12.9.

Table 12.9 *Some memory management calls* ill *Windows 2000*

Function	*Description*
Vi.rtualAlloc	Reserve an area of memory
VinualFree	Release an area of memory
VirtualProtect	Change the protection parameters (read/wrire/execure) on a region
VinualQuery	Obtain information about a region
VirtuaiLock	Disable paging for a region
VirtuaiUnJock	Enable paging for a region

Windows 2000 supports the concept of a sequential 4GB address space per process. Segmentation is not allowed. Interestingly. the memory manager docs not work with threads, but instead, it works with

Record 11: *$EXlend* This record is actually a directory, which contains miscellaneous files containing information regarding disk quotas. object identifiers, etc.

Records 12-15: *Unused* These records are currently unused, and are reserved for future use.

After this, the entries for the user files begin.

Important Features 0/ NTFS

Let us now discuss some important features of NTFS.

Recoverability The requirementof most modern Operating S),stenlS is to be able to gracefully recover from disk crashes and system failures. NTFS is capable of restoring the file system in the case of such events. It uses the concept of transactions, as is done in the case of database systems, to perform tbe recovering function. Whenever NTPS makes any changes to the file system, it actually begins a transaction internally. and only when operation is successful, it marks the transaction as complete. Otherwise, it aborts the transaction. What this means is that it records the pre-image of the disk in a log. and in the case offailures, it rolls back the state of the disk to the pre-image. This makes the data on the disk recoverable. Moreover. as we have mentioned earlier, NTFS uses redundant copies of the important system files, so that it can fall back on the copies, in case the main file gets damaged because of any reasons.

Security NTFS uses the Windows 2000 object concept in order to make the file system highly secure. We have discussed the object model earlier. When a file is opened, for example, it is treated as an object. and an object security descriptor associated with that file describes the security characteristics associated with that file.

lArge file handling NTFS is far more efficient at handling large files and disks, than its predecessors, such as FAT.

Indexing NTFS provides a collection of attributes per tile. The file descriptors are organized as a relational database, making the indexing of files by any attribute possible.

NTFS File Structure

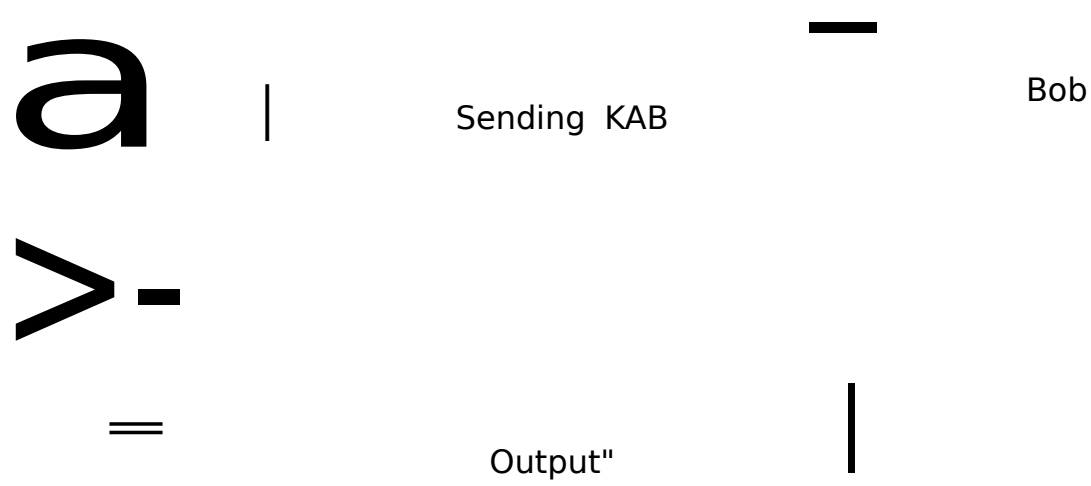
NTFS uses the following concepts when allocating disk space to files:

Sector This is the smallest area of physical space on the disk, from the viewpoint of NTFS. A sector typically consists of 512 bytes (or another power of 2).

Cluster One or more consecutive sectors in the same track make up a cluster.

Volume This is a logical partition on the disk. It consists of one or more clusters, and is used by NTFS while allocating space to a file. Typically, a volume not only contains files, but also the information about these files, free space information, etc. There can be one or more volumes per physical disk.

This concept is shown in Fig. 12.11.


Bob
Sending KAB
Output"

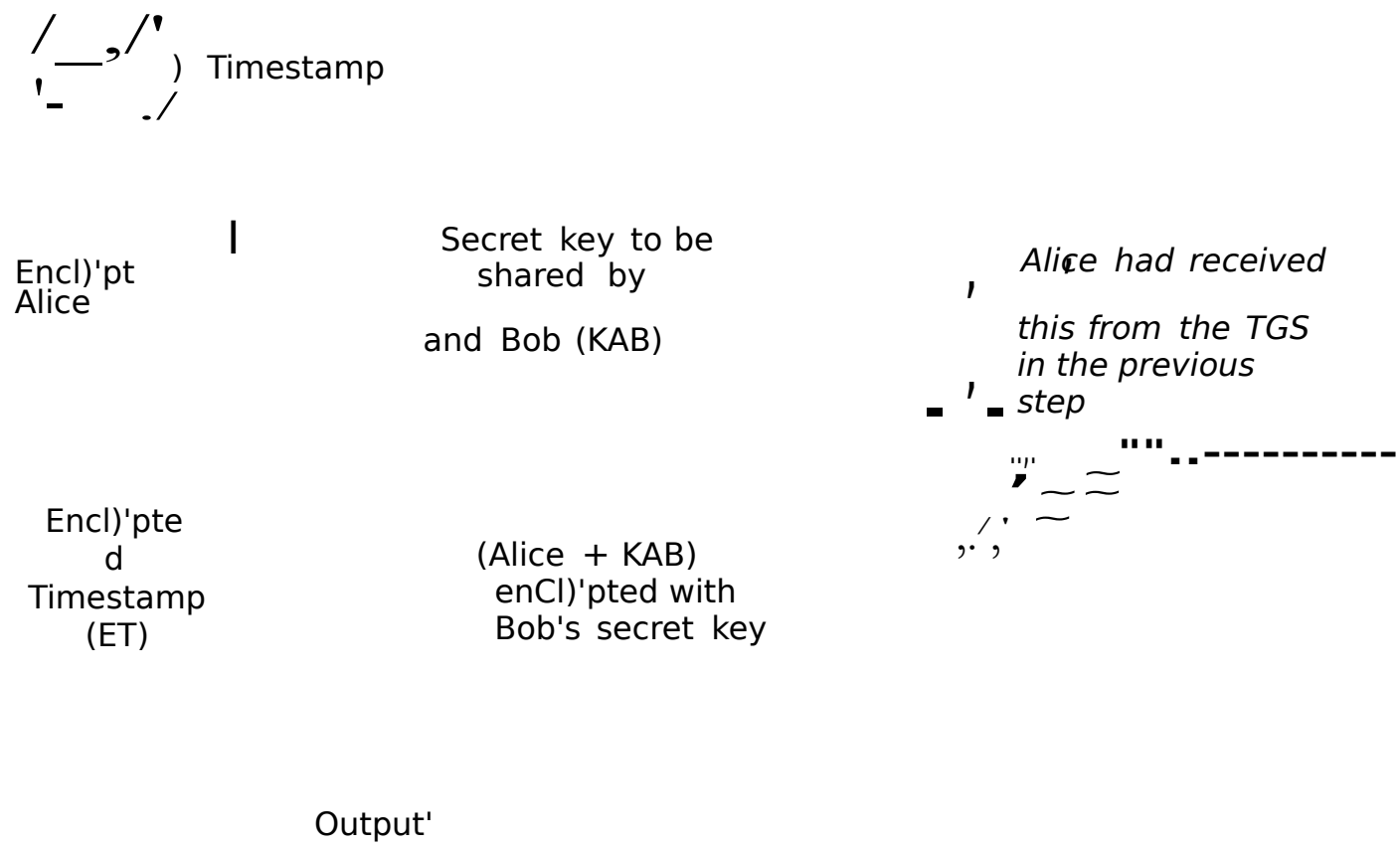
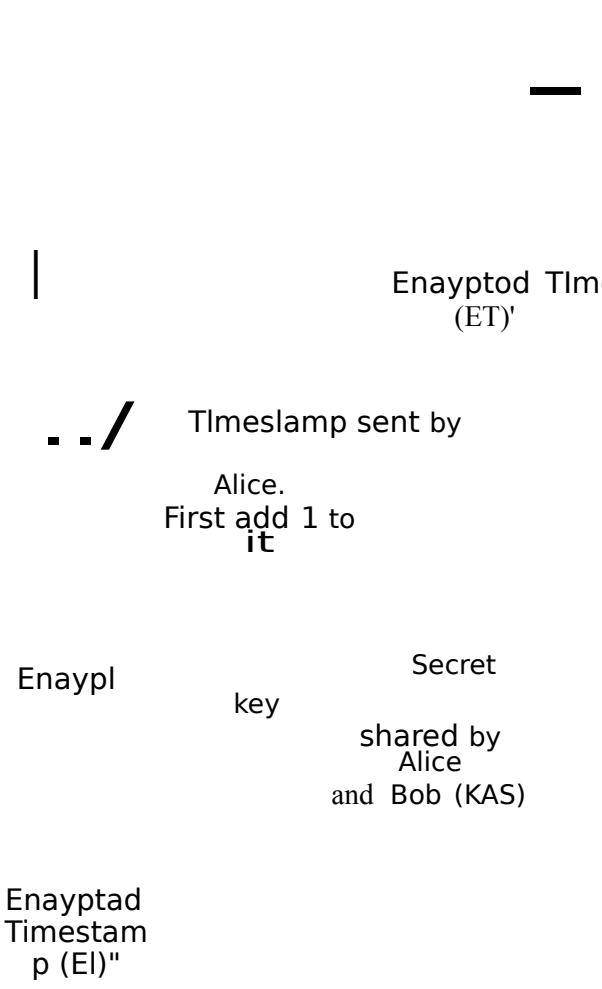


Fig. 12.19 Alice sends KAB securely to Bob



'--" Fig. 12.20 Bob acknowledges the receipt of KAB

Now. Alice and Bob can communicate securely with each other. They would use the shared secret key KAB to encrypt messages before sending. and also to decrypt the encrypted messages recei ved from each other.

Multiple Choice Questions

12.1 Another word for desktop computer is _
(a) microcomputer (b) minicomputer
(c) mainframecomputer (d) Microprocessor

12.2 Windows NT is a Operating System.
(a) 16 bit (b) 8 bit
(c) 32 bit Cd) 24 bit

12.3 In paging technique used in Windows NT, the 32-bit address field divided into page directory pointer, and _
(a) page pointer, page table pointer (b) page table pointer, page offset
(c) segment pointer, segment offset (d) page table pointer, segment offset

12.4 is used to store the critical information about the computer.
(a) CPU (b) Windows Registry
(c) RAM (d) ROM

12.5 The main feature of portability is achieved with the help of _
(a) kernel (b) hardware
(c) system interface (d) HAL

12.6 Thread handling is achieved in _
(a) hardware layer (b) hardware abstraction
(c) kernel (d) system inrerface

12.7 In UNIX _ is used for inter-process communication,

(a) pipe
(b) rnailslots
(c) sockets
Cd) RPC

12.8 FAT-16, FAT-32and are the three main/significant file system supported by Wi.ndows 2000.
(a) PAL
(b) J'I.'TSC
(c) SBCA.M
(d) FAT-S

12.9 Windows 2000 uses for authentication. (a) Kerberos
(b) EPS
(c) Semaphore
(d) Mailslot

12.10 [redacted] list the filename, other attributes of file and is used to describe a file or a directory.
(a) MFr
(b) NTFS
(c) BPS
(d) *SSO*

Test Questions

12.1 Discuss the history of Windows NT and Windows 2000.
12.2 How does Windows NT handle processes?
12.3 Discuss tile Significance ofWin32 API.
12.4 Why is Windows 2000 registry important?
12.5 Windows 2000 is made up of a few layers. Describe them.
12.6 What are the various entities to be managed with reference to process management in Windows 2000?
12.7 What is NTFS? What is ~1FT?
12.8 Discuss authentication and file security in Windows 2000.

typed in bad to be displayed on the terminal with aU the features such as backspace, tab, scroll. etc. The same text had to be stored in files. The files had to be stored on a disk in some form of directory structure. When a user wanted to edit an existing file, the file had to be retrieved in the buffers and the desired chunk had to be displayed on the terminal. Hence, the editor involved a support of all these *1/0* functions. An Operating System, therefore, was a must for this editor to be successful. This reaUy provided the motivation for the development of UNIX (or at least Thomson grabbed this opportunity and used it to his advantage!)

Initially, UNIX was written in assembly language like all other operating systems of the time. Portability was not thought of then. The actual creation of the UNIX system was Thomson's idea. Ritchie made valuable contributions to it though. For instance. in the UNIX system, devices are treated as files in the file system. This was Ritchie's idea. Both XDS-940 and MULTICS bad influenced Thomson and Ritchie a lot in terms of UNIX design. Another researcher at Bell Labs, Brian Kernighan used to call this new operating system as UNICS (i.e. Uniplexed Information and Computing Service). Because UNIX was regarded as a 'Castrated' version of tvltfLTICS, some people at the Bell Labs, used to call it, disparagingly, as 'Eunuchs' as well. Despite this, the name UNICS continued for a while. In the end, the spelling was changed to UNIX.

The end of 1971 saw the first version of UNIX, and finally three typists from the patent department were called in to "system test" the product. Ritchie and Thomson worked hard during those days to avoid any errors. Those were the days of anxiety, tension and uncertainty. As THE DAY approached. the tension mounted. Luckily, the experiment was successful. The patent department immediately adopted UNIX as th.estandard Operating System. As the need for computing as well as the popularity of UNIX both increased through the word of mouth, it spread gradually within the Bell Labs first and then slowly outside. Even then, until 1.973.very few people outside BeD Labs had heard about U"NlX.

It was only in October 1973 that UNIX was first publicly described at a symposium on Operating Systems Principles held ironically at mM in Yorktown Heights!

Despite these developments, Bell Labs had no particular interest in UNIX as a commercial product. They only set up a development group to support projects inside Bell. Several research versions were licensed for use outside.

The main problem in the growth of popularity was its lack of portability. UNiX was written in assembly language. In those days, very fast and efficient computers had not come into existence and writing an Operating System in a higher level language was considered to be very impractical by the computing community. Despite this. simplicity rather than speed or sophistication. was always the most important objective of UNIX design. Wanting to rewrite UNIX in a High Level Language (HLL), Ritchie himself designed a language for this purpose. He called it "B". B was a simplified version of another language called BPCL, which was very similar to PLII but which never was successful. "B" did not have good control structures. Ritchie designed another language for this purpose which he called C. He also wrote a very good compiler for it.

After this, UNIX was rewritten in C by both Thomson and Ritchie together. This happened in 1977. C combined features of both high and low level languages. It could, therefore, provide a good compromise between the execution speed and development time and ease. UNIX now consisted of I0.()00 lines of code, of which only 1000 lines were retained io assembly language for the purpose of speed because they controlled the hardware directly. If you wanted to port UNIX, you needed onl y a C compiler on the new machine and you would have to rewrite *100/0* of assembly code for the new machine!

A number of different types of devices can be anached to a computer running UNIX. The Information Management portion of UNIX kernel has drivers for different devices. As new devices are manufactured, the device manufacturers or the vendors of UNIX Operating System add drivers for these devices to support them. This process is greatly facilitated by a concept called "device independent *IIO*". A device in UNIX is treated like a file. What does this statement actually IDCan? There is a directory in UNIX under the root directory called "/dev" which consists of rl number of files, one for each device. Each file has access rights siuularl y defined as for normal fj les,

Each device file is a very small file which contains details of the.specific characteristics of that device. You could DOW think of a generalized device driver in UNIX to which these characteristics can be supplied as parameters to obtain a 'tailor made' device driver for a specific device at a given moment.

Things of course are not that simple to be so easily generalized. But you could write generalized drivers for devices which are at least similar. If such a driver is available, adding a new device is simple. You only have to create a file for that device in/dev directory. If the device characteristics differ a lot, you may have to modify the generalized device driver or write a new one for it corresponding to the file created in *Idev* directory.

For instance, the devices can be character devices like terminals, or block devices like disks. Each will require a certain specific driver program to be executed, and a specific memory buffer from/to which data can be read/written. Hence. for any device *IIO.* you will have to specify the device type, and its characteristics, and the addresses of the driver program as well as the memory buffer reserved for the device. These and other details about the device are stored in the device file for thai device in the Idev
directory in a predefined format or layout.

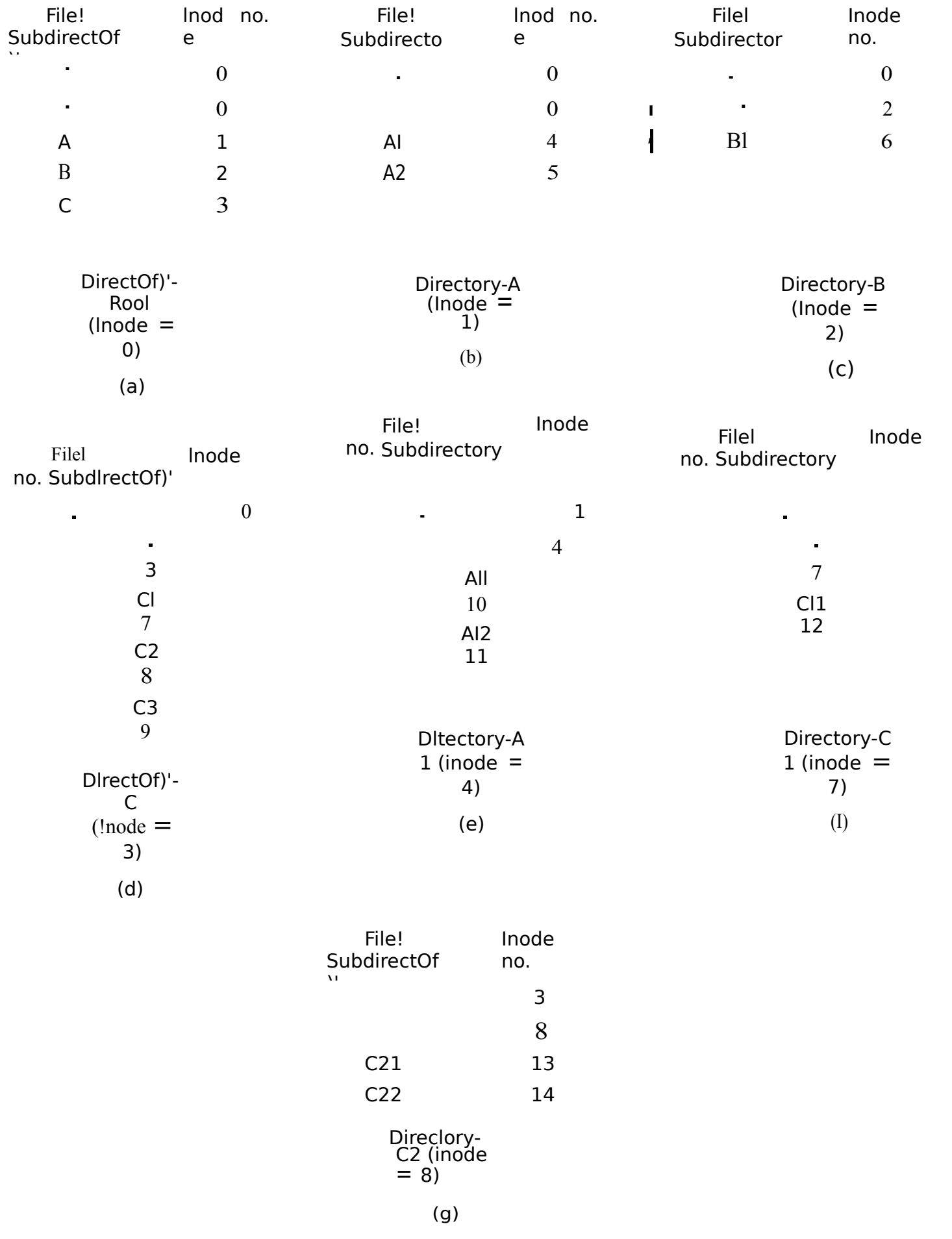
In UNIX, if you want to write to a specific device, you issue a system call to write onto the device file for that device in the /dev directory. The user issues a usual instruction as per the programming language syntax. For instance, he may want to write to a printer. Corresponding to this instruction, the compiler generates a system call to write to a device file for that printer defined in Idev directory. This system call to write to a device file is intelligent. At the time of execution, it extracts the contents of the device file
which contains the device characteristics and the address of the device driver. It uses this information to invoke the required device driver and so on. When a new device is added to a system, the device driver for that device is written and a corresponding "device file" is created under Idev. This is what we mean when we say that devices are treated as files in UNIX.

You do not need all the hardware and devices available 011 your machine all the time. Remember, the more your need, the m.ore will be the size of the operating system because, those many drivers will need to be included and loaded and those many buffers will have to be created. Thus, you uormally specify to the system only what you need. This process is called SYSTEM GENERATION.

The 'UNIX, file system is hierarchical. Its logical implementation will be discussed later. UNIX processes execute in either a user mode or kernel mode. When an interrupt occurs (e.g. due to I/O
completion or a timer clock interrupt), or a process executes a system call, the execution mode changes from user to kernel. At this rime, the kernel starts processing some routines, but tbe process running at that time will still be the same process except that it will be executing in the kernel mode.

Processes have different priorities which are "intelligently" or heuristically changed. so that *1/0* bound processes have rnore priority than the CPU bound processes with the passage of time. Processes with the same priority gel their CPU time slices in a strictly round robin fashion.

if a process with higher priority wants to execute when there is no memory available, two techniques are commonly used across UNIX implementations. One is swapping, where the entire process image is swapped out (except its resident part). Another is demand paging where only some pages are thrown out and the new process is now created. In this case, both the processes continue. These days. demand paging is more popular.



D;;. Fig. 13.6 The directories (SFD) for Fig. 13.4

After this, we can repeat the procedure to read that inode, pick up the pointers or addresses in the inode to get to the actual blocks. If it is a directory within AI, we could repeat this procedure iteratively to traverse anywhere within the hierarchy. (This is why we say that directories are treated as tiles in UNIX.) If it is a file within the directory Al, the address field takes us directly to the data blocks allocated to that file. The kernel now can use this to carry out the address translation to arrive at the sectors that are needed to be read. It then can instruct the controller to read those sectors to read the data.

Every directory has two fixed entries-one for the entry of'.' and the other for' ..'. The '.' entry is for

the. same directory. The inode number against it is for the same directory. For instance, in directory for Cl in Fig 13.6(f), the' .' entry maintains 7. This is the inode number for directory C 1 as Figs 13.5 and 13.6(d) show.

The' ..' entry is for the parent directory and its corresponding inode number. For instance, the·_.' entry for directory Al shows the inode number I. This is the inode number of directory A as Figs 13_5

We will now study some of these in more detail.

(i)
/bin/ls

Ibinlls is a utility program to list the contents of a directory. For example. if Is command is given at a root directory. the command as well as the displayed results are shown in Fig. 13.12. which correspond to the subdirectories under the root directory shown in Fig. 13.11.

This utility is used a number of times by almost every end-user. When this command is given to the shell running

on the system as the command interpreter for UNIX, the shell process uses the system call to locate the binary executable file for this command. It finds it in Ibin directory. The Ibin directory will contain various symbolic names and their corresponding inodes. The kernel can search for the name "Is" as the symbolic file name. Having found it, it can get the in ode number, and access the corresponding inode . The inode for the file "Is" contains the address of the disk blocks, where the actual executable binary code for "Is" is stored. It then creates another process (forks it), loads the binary compiled code of "Is" utility into the process address space of this newly created or forked process. This new process now takes charge and executes the command "Is" by .again issuing various system calls for searching for different directories under the root directory. and then displaying their names.

```
Sis
  bin
  uni
  x
  usr
  lib
  mn
  !
  etc
  Imp
  dev
S
```

O. Fig. 13.12 The "ls" command

"Is" has many options, the discussion of which is beyond the scope of the current text.

(ii)
/bin/who

Ibin/who utility outputs information about the currently logged on users. It reports user names. the terminals being used. the dates and the times at which they have logged on to the system. A sample command and its output are shown in Fig. 13.13.

```
Swho
Empl     console     Dec.1S      10:10
Emp2     ttyl        Dec.IS      11:15
Emp3     tty2        Dec.1S      11:30
$
```

Fig. 13.13 The "who" command

UNIX maintains these details for every user in its internal data structures which are used to execute this command. Again. in this case too, the binary code for "who" is located first. then loaded and finally executed.

U' UNIX had followed the same strategy as *AOSNS* of maintaining the access rights explicitly for each user, the ACL would have been very long and would have consumed a lot of disk space. It also would have taken a long time for verification. UNIX, therefore. arrives at a compromise, It divides all the users for each file into three broad categories. The tirst is called the Owner-the one who creates the file. The owner's uid, therefore. defines this category. The second category is that of the Group- defined by the gid of the owner. The third category is that ofOtbers-which do not belong 10 the first two categories. Apart from the owner. all others either belong to the category of "group" or the category of "others".

UNIX, therefore, has to define access rights for each tile for only these three categories of users. UNIX provides for only three types of access rights-r (read), w (write) and x (execute). It reserves 3 bits for these 3 access rights. If 0 means access right denied, 1 means that it is granted. Hence, if the access rights for a given file for the owner are 101, it means that the user can read from the file, and cannot write onto the tile, but can execute it. On the assumption that it is a file consisting of a compiled, binary. executable program, This is the reason it is sometimes written as r-x, though internally it is
maintained as 101 in the ACL. Similarly. if the group has access rights of 001 or - x for a file, it means that anybody in the group call only execute the file, but nobody within that group can perform any read! write operations. Therefore, there are 9 bits in inode (3 bits each for 3 categories), which define the access rights for that file completely.

The Access Control Veritication now proceeds as follows:

(i) A user is assigned a uid and gid by the System Administrator as discussed above. This is stored in the letc/passwd file. Let us call them uid-u and gid-u, for future reference.

(ii) At any time. when a user logs on to the system, he keys in his user name and password. As seen earlier, the system retrieves his uid after consulting the letc/passwd file. UNIX (a Login process in this case) validates these with the ones stored in the letc/passwd file. Only after a match is found, access to the system is granted: and the user can give a command to the shell. When this user logs on and starts any process, the uid-u and gid-u for that user are copied from the letcl passwd fi le to the u-area of that process. If that process creates a child process, that child process also will have a u-area which will inherit the same uid-u and gid-u from the u-area of the parent process.

(iii) Let us now assume that a user gives a command to execute a COBOL program called PA Y.COB.

PA Y.COB will be a file in the directory system of UNIX containing the executable program, i.e. its binary code. Also let us assume that after compiling the program and creating this file, the system analyst decides as to who can execute(x) it and the system administrator accordingly sets its access control bits in the inode appropriately using certain previleged commands which only be can execute. After receiving the command from a user to execute this program, the kernel accesses the illode for PA Y .COB by accessing the chain of Symbolic File Directories and their inodes as seen before for resolving the pathname.

After finally accessing the inode for PAY.COB •.it stores the user id and group id stored in the inode for PAY.COB. Let us call these uid-i and gid-i for future reference. These are the user id and group id of the owner of that file. Now, the matching of the user id and group id in the u-area of the executing process (uid-u, gid-u) and those in the inode of the file that the executing process wants to access (uid-i, gid-i) has to take place. This is done in the following steps.

(iv) The kernel now executes the following algorithm to decide the category of the user (assuming that the user is not a superuser),

(1) It now grabs the next free entry in the UFDTwhich is a part of the u-area for the process opening that file. Hence, this table will already be existing after the process was created. This "next free entry" is called the file descriptor (fd). The kernel stores the "fd" and generates a pointer from that UFDT entry to the entry in the FT.

(g) The kernel returns the "fd" to the process.

If the same process opens the same file but in two modes-one for reading and the other for writing, the algorithm is pretty much the same. This is because, in this case, an in-core inode entry will already exist for that inode, and, therefore, it will not be copied a fresh from the disk. The algorithm takes care of this. In this case, there will be two UFDT entries pointing to two respective entries in the FT but both FT entries will point to only one entry in the IT. Figure *13.21* depicts this position.

13.4.8 "Read" system call

The system call for "Read" is as follows:

number = Read (fd, buffer. count), where:
fd stands for file descriptor.
buffer stands for the starting address in the memory where the data is to be read.
count means the number of bytes to be read.
number indicates the actual number of bytes read after the execution of this system *call.* If the system call fails, or end of file is encountered before all the bytes are read, this number will be different from the "count" field supplied as a parameter. (Otherwise it should be the S8I11e.)

It should be noted that the reading struts in a file at the offset maintained in the FT entry for that file and that mode (i.e. Read). The kernel carries out the following steps to execute the system call:

(a) The kernel uses the fd as the index to access the correct entry in the UFDT.

(b) It follows the pointer from the UFDT to the FT to the IT to access the correct IT entry. As we have seen, all these entries are set up at the time a file is opened in a specific mode.

(c) The kernel verifies that the user for which the process has issued this system call has the 'read' access to this file. The inode contains rwx access rights for all the three categories. From the user's category, the kernel establishes this access right for the user. If it is denied, it outputs an error message and exits. We have studied this earlier.

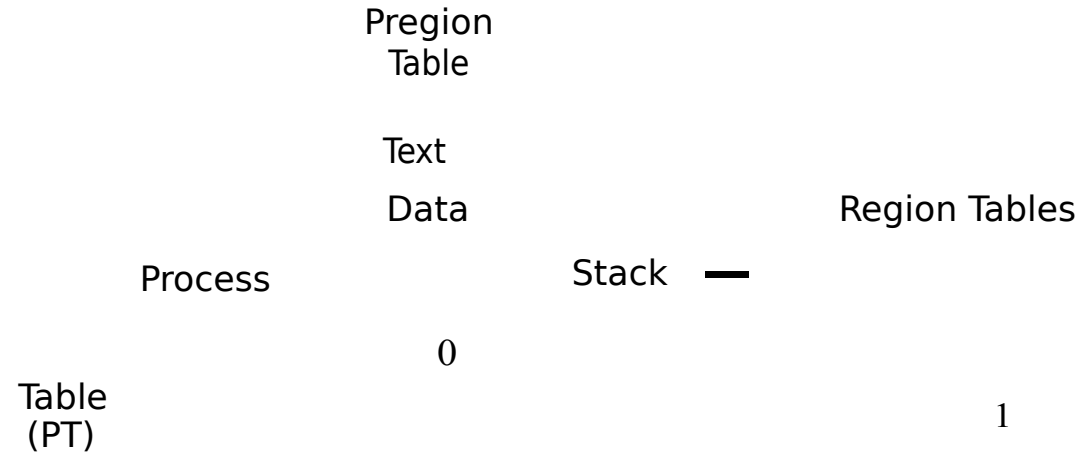
(d) The kernel now sets up various fields in the u-area to eliminate the need to pass them as function parameters. These parameters are as follows:

mode	.	read or write (read in this case).
offset	.	byte offset In the file from the FT entry.
address		target starting memory address-supplied with the system call parameters.
▪.		to indicate whether the target memory is in user or kernel memory space.
Indicator		Number of bytes to be read-supplied with the system call parameters.
•,		
count	:	

Out of the above parameters, almost aU the fields are derived from the parameters of the system call. Mode has to be read for the 'read' system call, and the offset is picked up from the entry in the FT.

(e) The kernel now locks the inode entry in the IT. It is known that for any operation on that file (write, close, etc.) this inode entry in the IT has to be consulted. Hence, by changing the status of

. The **Other** Infonnatio.n includes symbol tables which are used for debugging a program at a source level. When a program siam executing, this symbol table is also loaded. It gives various data and paragraph names (labels) and their machine addresses. Thus, when a source level debugger wants to display the value of a counter, the debugger knows where to pick up the value from.



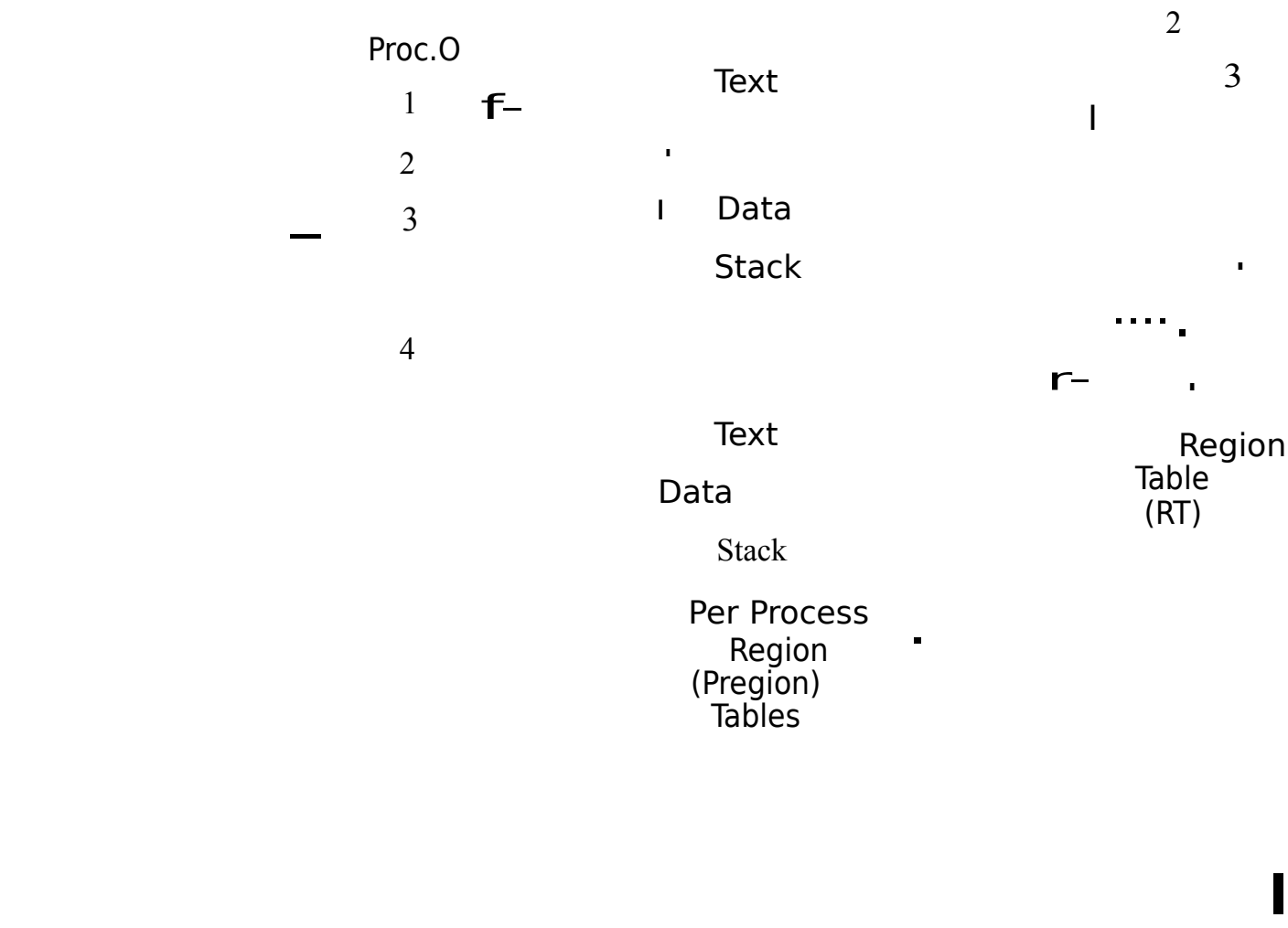
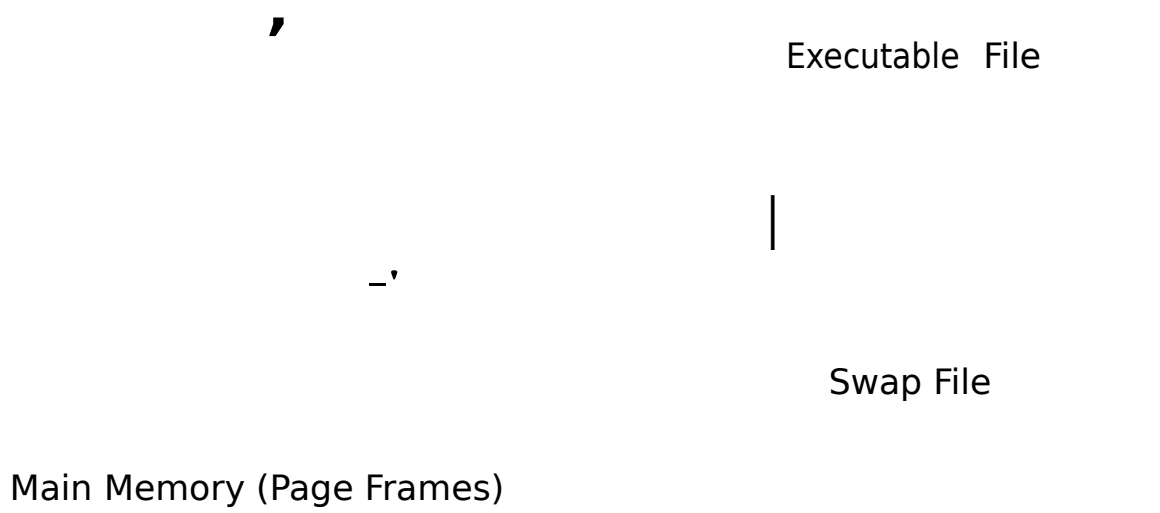


Fig. 13.27 Various UNIX data structures

(iii) If the page belongs to the data or stack region. and is not modified (i.e, dirty bit = 0). the same logic applies. It does not need overwriting. Only if the dirty bit = I. it is overwritten onto the swap
file.



G:> Fig. 13.31 Main memory. executable and swap files

The point is that a page from a process could be at any of the following three locations as shown in Fig. 13.31.

- Executable File
- Physical Memory
- Swap File

This complicates the Page Map Tables because for each virtual page. more information to locate and access it needed. The Page Map Table then looks as shown in Fig. 13.32.

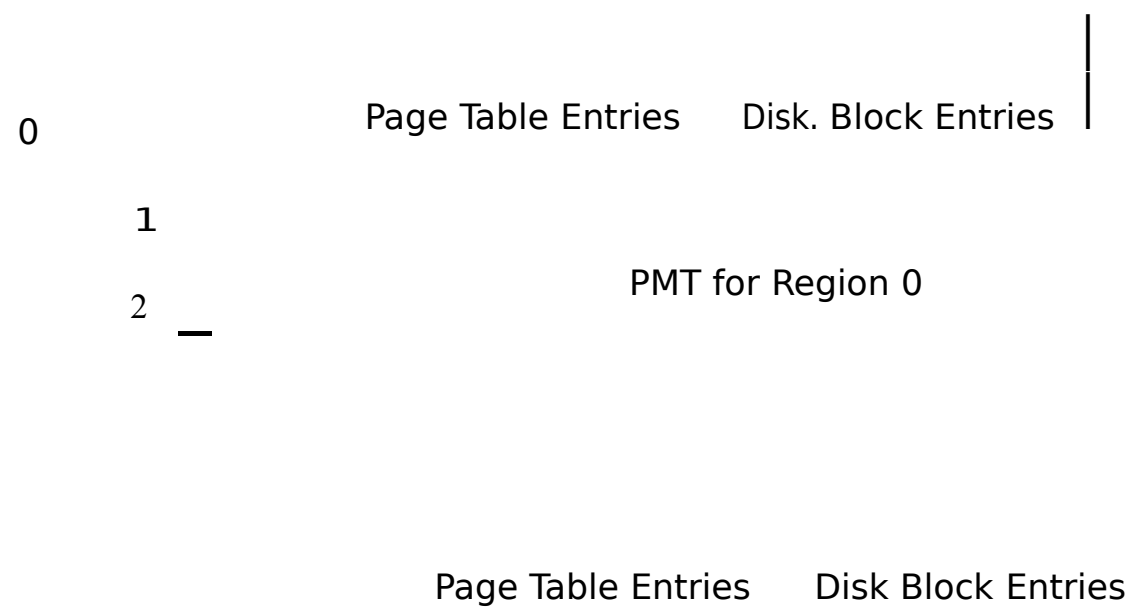


Fig. 13.32 PMTs in demand paging

the parent was shelL. In this case, the shell prompts for a new command. As soon as a new com- mand is given, it executes the algorithm in Fig. 13.35 again. This time if a different program is to be executed, the "Exec" will have to locate and load the code for that program and execute. However, the general scheme remains the same,

A detailed disJussion on the "Fork" and "Exec" system calls is given in tbe following sections:

13.7.2 "Fork." System Call

When tbe fork system call is executed, the following actions are carried out (refer to Fig. 13.36).

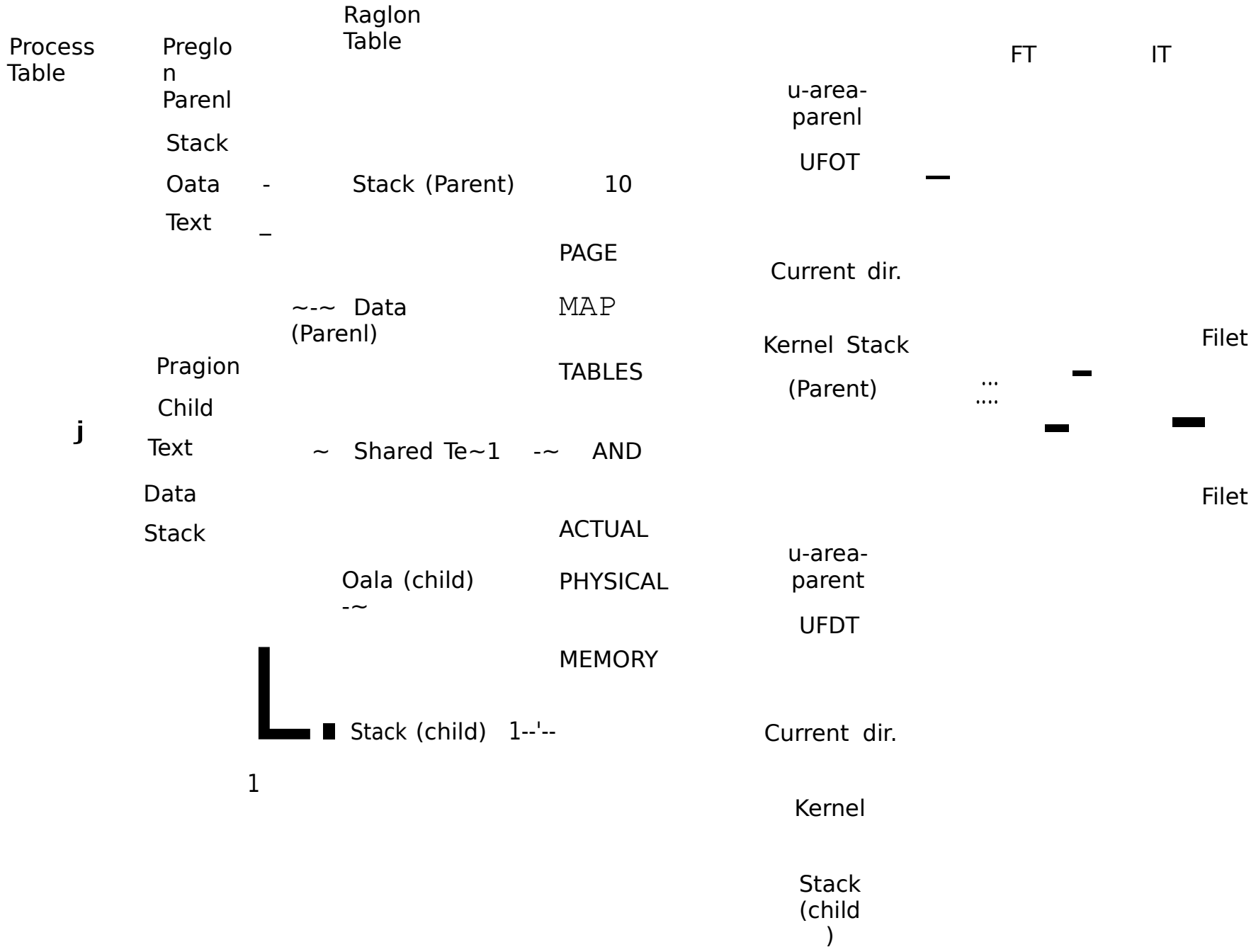


Fig. 13.36 Data structure after "Fork"

(a) The kernel ensures that there are enough system resources to create a new process. This is done as follows.

(i) It ensures that the system can bandle one more process for scheduling and that the load on the scheduler is manageable.

(ii) It ensures that this specific user is not running too many processes by monopolizing on the existing resources.

(iii) It ensures that there is enough memory to accommodate the new process. It is known that the new process is identical to the parent process in every aspect at this juncture. This also in- cludes the memory requirements. In the swapping system, tbe entire memory has to be avail- able. In pure paging systems, the memory for all the pages to hold the entire address space as well as the page map tables is necessary. In demand paging, only the page map tables are necessary at the least to initiate a process. Further pages from the address space can be accumulated with page faults in demand paging.

of memory management were considered in earlier sections. They are valid for UNIX with only a few variations. Therefore, only these two schemes will be considered in brief.

When the full process image needs to be in the memory before the process can be executed. swapping is used. If a high priority process arrives when the physical memory is insufficient to hold its image, a process with lower priority has to be swapped out. At this time, the entire process image is written onto the portion of the disk known as a swap device. When that process needs to be run again, it has to be "swapped in" from the swap device to the physical memory.

A point needs to be clarified here. Swapping can co-exist with simple paging. The executable code of a program can still be divided into a number of pages and the physical memory can be correspondingly divided into several page frames. The process image in the main memory can spread over a number of physical page frames, Which are not necessarily contiguous. and, therefore, requires the page map tables to map the logical or virtual addresses into physical page frames. However, even in this case of simple paging. the entire process image is swapped out or swapped in. This is what really differentiates swapping or simple paging from demand paging. In the former, the entire process image has to be in the physical memory before execution. whereas in demand paging, execution can start with no page in the memory to begin with. The pages are brought in only when "demanded" through page faults as we have studied earlier.

Initially, UNIX implemented swapping as a memory management system. Swapping limits the maximum size of process to that of the physical memory. Later implementations such as BSD 4.0 implemented demand paging, thus, removing this limitation. However, it must be remembered that in terms of implementation, swapping is an easier method to implement, whereas demand paging is more difficult. Both swapping and demand paging will now be considered.

13.10.2 Swapping

A swap device is a part of the disk. Only the kernel can read data from the swap device or write it back. Normally, the kernel allocates one block at a time to ordinary files, but in the case of a swap device. this allocation is done contiguously to achieve higher speeds for the 110 while performing the functions of swapping in or out. This scheme obviously gives rise to fragmentation and may not use the disk space optimally, but it is still followed to achieve higher speeds.

~ Fig. 13.43 Allocated and free blocks

For instance, Fig. 13.43 shows the allocated (shaded) and free blocks on the swap device. If a process with size = 300 blocks is to be swapped out, these blocks will be allocated out of the second free chunk of 500 blocks. The kernel does not decide to allocate 200 blocks from the first free chunk and 100 from the next chunk because then the allocation will no longer remain contiguous. If the allocation remains non-contiguous. the kernel will have to maintain an index (like inode) to keep track of all the blocks allocated to the swapped process file. Such a process wiU degrade the 110 performance. When a process has to be swapped in, the kernel will have to go througb the index. find the blocks that need to be read,

(d) The kernel then finds out whether the pages belonging to the text regions are already available in the main memory. This is done by using the hashing techniques. From the device number and the block number for each block in the executable file, an appropriate hash queue is searched to check if any page is available. If available, it can be directly assigned to this process. In this case, the page frame numbers in the PTEs of these pages are then updated in the PMT. The valid bits of these pages also are set to 'ON'. It can skip the steps (e), (0, (g) and (h), if the required page is already existing in a page frame,

(e) If any page does not exist in the main memory, a page has to be loaded from the disk into a free page frame. To do this, the kernel goes through the PFDT chain for free page frames, starting with its header.

(0 This is the way in which the kernel allocates the page frames to this process. (They could amount to less than the total size of the process address space.) After the allocation, the kernel removes tbe pages from the free list and it chains them in the list of PFDT entries as per the hashing algorithm. It now sets the valid bit ON and other (Reference, Modify, etc.) bits for these pages OFF in the PTEs.

(g) It updates the page frame numbers in these PTEs appropriately.

(h) The kernel then loads the pages from the executable file into those allocated page frames and also updates the DBD entries. It marks the executable/swap file indicator to "executable" for those DBD entries.

(i) For all the other pages, which are still not in the memory, the kernel sets valid bits = OFF, and page frame number = blank to denote that the page is not in the physical memory so that a page fault would occur. It, however, fills up the details of the DBD entries for those pages with the respective device number, block number addresses of the "executable" file. so that those pages could be located on a page fault.

(j) The process now starts executing. For every instruction. the virtual address consists of two parts: page number (P) and displacement (d). The hardware separates these two parts. The page number (P) is used as an index into the PMT directly. At this juncture, its valid bit is checked. Assume that it is valid (i.e. it is in the memory). The page frame number of the PTE is then extracted and the displacement (d) is appended to it to get the physical address. The instruction can be com- pleted. The reference bit is then set to ON. If the instruction is a 'write' instruction, the 'modify' bit in the PTE is also set to ON.

(k) Assume that the CPU encounters an instruction with an address containing an "invalid" page. This is called "page fault". At this time, the instruction which caused the page fault is abandoned. It is again taken up only when the required page *is* read into the physical memory. The hardware must have this capability to abandon and restart an instruction (after the page fault processing) to support demand paging. At this juncture, the kernel again finds out whether the page is already in the physical memory by using the hashing algorithm. As discussed earlier, the page may have been used by a process terminated some time ago but the page may not have been removed.If itis not in the physical memory, it goes through the free list of PFDT entries to check if a free page frame is available, so that it can be loaded from the disk.

(l) Assume that a free page frame is available. The kernel grabs it and removes it from tbe free list. The kernel extracts the device and block number from the DBD of the PMT entry for that page with valid bit OFF. It then loads the contents of thai page from that address into the allocated page

- Page Frame Data Table (PFDT)
- Page Stealer Process
- Pathname Resolution
- Pipes
- Reentrant Code
- Region Table
- Relative Path Name
- Page Map Table (PMT)
- Page Table Entry
- Per Process Region Table
- Process Table
- Reference Bit
- Regions
- Shell Scripts

- Superblock
- Swap Device
- Unmount
- Superblock Cache
- n-area
- User File Descriptor Table (UFDT)

- User Id
- Valid Bit
- Zombie State
- User Slack
- Working Directory

- UNIX was written by Ken Thomson.
- The success of UNIX can be largely attributed to its simplicity. spread into universities and the use of a high-level language (C).
- UNIX consists of a kernel and a number of utility programs.
- Shell scripts are like macros in the assembly language.
- UNIX kernel is supposed to be divided only into two parts: Information Management and Process Management.
- UNIX implements a hierarchical file system. In this scheme, a directory can have a number of files and subdirectories underneath it.
- UNIX recognizes four types of HIes: Ordinary files. Directory files. Special files, and FIFO files.
- In fork. a process creates a child process and duplicates all the essential data structures such as u-area.
- Pipe is a mechanism provided by UNIX to communicate between two processes.
- The process table is maintained at a global level, wherein there is one entry per process known to UNIX on the system at any given juncture, regardless of its state. The u-area contains information about a process.
- The page map tables maintain information about logical pages versus physical page frames for each region.
- A process executes either in a user mode or a kernel mode .
- In UNIX. Fork is the only way that a process can create another process. UNIX bas an Exit system call to terminate a process.
- The Wait system call is used by a parent process to sleep until the death of the child after which itcan continue.
- Two memory management schemes are used roost extensively in UNIX. One is swapping and the other is demand paging.
- Solons supports multiple levels of threads. Overall. it provides support for four separate thread-related concepts. These are: process, user-level thread, lightweight process and kernel thread.

The "clone" function begins with a call to the function (which is the first parameter to the *clone* function - refer to the syntax). The last parameter to the clone function *is* argument, which is passed as it is to function. This concept is illustrated in Fig. 14.1.

```
pid = clone (function.  stackJlointer. Sharing_flags./ent);
```

```
function(argu;;;nt)
{
        ...
}
```

'"'i' Fig. 14.1 How the *clone* function begins execution

The new thread created by the "clone" function call receives its own stack. Its stack pointer is initialized to the value as specified in the stack_pointer parameter,

The value of the parameter sharing_flags provides bener sharing properties with regard to this thread 3I1d its parent. Table 14.1 depicts the possible values of this flag, and the meaning that it conveys.

Table 14.1 *Sharing flags in the "clone" call*

Flag value	*Meaning if this flag* is *set*	*Meaning if this jkJg is not set*

CLONE_VM	Create a new thread within the process when this call is executed	Create a completely new process
CLONEYS	Share the properties of the parent process, such as the root directory, working directory, etc.	Do not share these properties of the parent process
CLONE_FILES	Share the files of the parent process	Create a copy of the files used by the parent process
CLONE_SIGHAND	Share the signal handler table	Create a copy of the signal handler table
CLONE_PID	Use the same PIO, as of the parent process	Use a new PID for this process

Linux maintains the information about every task/process with the help of a table, called as task_struct. The main entries in the task_struct table are shown in Table .14.2.

Table 14.2 *Details maimained by Linux for each process*

Parameter	*Meaning*
State	Represents the state of the process, such as *Executing, Ready. Suspended, Stopped,* etc.
Scheduling information	Specifies the information required for scheduling the process. such as its priority. allocated time slice, etc.
Identifiers	Lists the process identifiers, such as the process' PIO, UID, G[D. etc.
IPC	Identifies which Inter Process Communication (IPC) mechanism out of the ones possible, to usc.
Links	Contains links to the parent (the process which created this process). siblings (other processes which have the same parent as this process) and children (processes thai are created by this process).
Times/timers	Specifies the process creation time, time used so far, etc.
File system	Contains pointers to the Illes used/accessed by this process.

Contd ...

Second Edition

This book, designed for the firs1 course on Operating Systems, aims at demystifylng the subject by using the step-by-step approoch of going from the very basics to advanced concepts. This book would be useful not only to the undergraduate students of computer science but also to opplicotion programmers who ore interested in knowing about the internols of the operating systems.

Ta ta McGra}v-Hill

Salient Features

- Good coverage of Porallel Processing.
- Coverage of Multimedia Operoting Systems.

New to this Edition

- Increased coverage of Operoting SystemsSecurity and Protection.
- Enhonced coveroge of Distributed Processing.
- Detailed cose studies on LINUX, WIN NT/2000.