HANDOUTS/ NOTES/ STUDY MATERIAL

Jagannath Institute of Management Sciences
Lajpat Nagar

OPERATING SYSTEM

UNIT 1

Introduction/Definition

An operating system act as an intermediary between the user of a computer and computer hardware.

The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.

An operating system is a software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

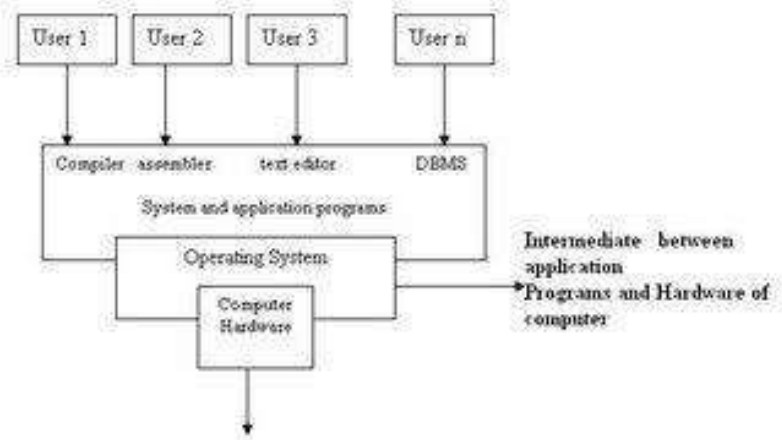
Definition of Operating System:

An Operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.

A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being applications programs.

An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The Operating System correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system. A computer system can be divided roughly into four components – **the hardware, the operating**

The operating system controls and co-ordinates the use of hardware among the various application programs for the various users.



Operating system from the user view-

The user's view of the computer varies according to the interface being used. While designing a PC for one user, the goal is to maximize the work that the user is performing. Here OS is designed mostly for **ease of use**.

In another case the user sits at a terminal connected to a **main frame or minicomputer.** Other users can access the same computer through other terminals. OS here is designed to maximize resource utilization to assure that all available CPU time, memory and I/O are used efficiently.

In other cases, users sit at **workstations** connected to networks of other workstations and servers. These users have dedicated resources but they also share resources such as networking and servers. Here OS is designed to compromise between individual usability and resource utilization.

Operating system from the system view-

From the computer's point of view, OS is the program which is widely involved with hardware. Hence OS can be viewed as **resource allocator** where in resources are –

CPU time, memory space, file storage space, I/O devices etc. OS must decide how to allocate these resources to specific programs and users so that it can operate the computer system efficiently.

OS is also a control program. A **control program** manages the execution of user programs to prevent errors and improper use of computer. It is concerned with the operation and control of I/O devices.

Defining operating systems-

OS exists because they offer a reasonable way to solve the problem of creating a usable computing system. Goal of computer systems is to execute user program and to make solving user problems easier. Hence hardware is constructed. Since hardware alone is not easy to use, application programs are developed.

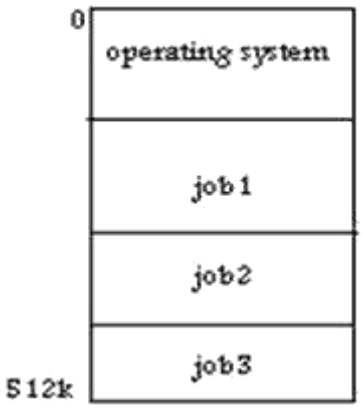
Operating System Structure-

An OS provides an environment within which programs are executed.

One of the most important aspects of OS is its ability to multi program. **Multi programming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

OS keeps several jobs in memory. This set of jobs can be a subset of jobs kept in the job pool which contains all jobs that enter the system. OS picks and begins to execute one of the jobs in

memory. The job may have to wait for some task, such as I/O operation to complete. In a non multi programmed system, OS simply switches to and executes another job. When that job needs to wait, CPU is switched to another job and so on. As long as at least on job needs to execute, CPU is never idle.



Multi programmed systems provide an environment in which the various system resources are utilized effectively but they do not provide for user interaction with the computer system. **Time sharing or multi tasking** is a logical extension of multi programming. In time sharing systems, CPU executes multiple jobs by switching among them but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an **interactive computer system** which provides direct communication between the user and the system. A time shared operating system allows many users to share the computer simultaneously. It uses CPU scheduling and multi programming to provide each user with a small portion of a time shared computer.

A program loaded into memory and executing is called a **process**.

Time sharing and multi programming require several jobs to be kept simultaneously in memory. Since main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool.**

This pool consists of all processes residing on disk awaiting allocation of main memory. If several jobs are ready to be brought into memory and there is not enough space, then the system must choose among them. Making this decision is **job scheduling**.

Having several programs in memory at the same time requires some form of memory management. If several jobs are ready to run at the same time, the system must choose among them. Making this decision is **CPU scheduling**.

An Operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware. A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being applications programs.

An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The Operating System correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system. Operating systems are there from the very first computer generation. Operating systems keep evolving over the period of time.

Following are few of the important types of operating system which are most commonly used.

1. Batch operating system

The users of batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. Thus, the programmers left their programs with the operator. The operator then sorts programs into batches with similar requirements. Some computer systems only did one thing at a time. They had a list of the computer system may be dedicated to a single program until its completion, or they may be dynamically reassigned among a collection of active programs in different stages of execution.

Batch operating system is one where programs and data are collected together in a batch before processing starts. A job is predefined sequence of commands, programs and data that are combined in to a single unit called job.

Memory management in batch system is very simple. Memory is usually divided into two areas : Operating system and user program area.

Scheduling is also simple in batch system. Jobs are processed in the order of submission i.e first come first served fashion. When job completed execution, its memory is releases and the output for the job gets copied into an output **spool** for later printing.

Batch system often provides simple forms of file management. Access to file is serial. Batch systems do not require any time critical device management.

Batch systems are inconvenient for users because users can not interact with their jobs to fix problems. There may also be long turn around times. Example of this system id generating monthly bank statement.

Advantages o Batch System

Move much of the work of the operator to the computer.

Increased performance since it was possible for job to start as soon as the previous job finished.

Disadvantages of Batch System

Turn around time can be large from user standpoint.

Difficult to debug program.

A job could enter an infinite loop.

A job could corrupt the monitor, thus affecting pending jobs.

2. Multiprogramming

When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system. Multiprogramming assumes a single processor that is being shared. It increases CPU utilization by organizing jobs so that the CPU always has one to execute. The operating system keeps several jobs in memory at a time. This set of jobs is a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the job in the memory. Multiprogrammed system provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.

Jobs entering into the system are kept into the memory. Operating system picks the job and begins to execute one of the job in the memory. Having several programs in memory at the same time requires some form of memory management. Multiprogramming operating system monitors the state of all active programs and system resources. This ensures that the CPU is never idle unless there are no jobs.

Advantages

1. High CPU utilization.
2. It appears that many programs are allotted CPU almost simultaneously.

Disadvantages

1. CPU scheduling is requires.

2. To accommodate many jobs in memory, memory management is required.

Time-sharing operating systems

Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing. The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, objective is to maximize processor use, whereas in Time-Sharing Systems objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receives an immediate response. For example, in a transaction processing, processor execute each user program in a short burst or quantum of computation. That is if n users are present, each user can get time quantum. When the user submits the command, the response time is in few seconds at most.

Operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are following

- Provide advantage of quick response.
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages of Timesharing operating systems are following.

- Problem of reliability.
- Question of security and integrity of user programs and data.
- Problem of data communication.

Distributed operating System

Distributed systems use multiple central processors to serve multiple real time application and multiple users. Data processing jobs are distributed among the processors accordingly to which one can perform each job most efficiently.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers and so on.

The advantages of distributed systems are following.

- With resource sharing facility user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

Network operating System

Network Operating System runs on a server and and provides server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. Examples of network operating systems are Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are following.

- Centralized servers are highly stable.
- Security is server managed.
- Upgrades to new technologies and hardwares can be easily integrated into the system.

- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are following.

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

Real Time operating System

Real time system is defines as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. Real time processing is always on line whereas on line system need not be real time. The time taken by the system to respond to an input and display of required updated information is termed as response time. So in this method response time is very less as compared to the online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. Real-time operating system has well-defined, fixed time constraints otherwise system will fail. For example Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, and home-applicance controllers, Air traffic control system etc.

There are two types of real-time operating systems.

HARD REAL-TIME SYSTEMS

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems secondary storage is limited or missing with data stored in ROM. In these systems virtual memory is almost never found.

SOFT REAL-TIME SYSTEMS

Soft real time systems are less restrictive. Critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, Multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers etc.

SYSTEM STRUCTURES

OS services-

OS provides an environment for execution of programs. It provides certain services to programs and to the users of those programs. OS services are provided for the convenience of the programmer, to make the programming task easier.

One set of SOS services provides functions that are helpful to the user –a. User interface: All OS have a user interface(UI).Interfaces are of three types- Command Line Interface: uses text commands and a method for entering them

Batch interface: commands and directives to control those commands are entered into files and those files are executed.

Graphical user interface: This is a window system with a pointing device to direct I/O, choose from menus and make selections and a keyboard to enter text.

b. Program execution: System must be able to load a program into memory and run that program. The program must be able to end its execution either normally or abnormally.

c. I/O operations: A running program may require I/O which may involve a file or an I/O device. For efficiency and protection, users cannot control I/O devices directly.

d. File system manipulation: Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information.

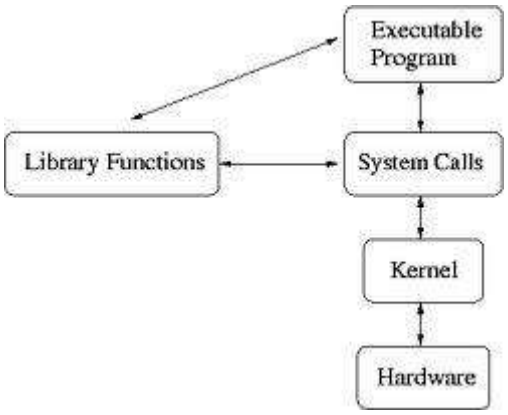
e. Communications: One process might need to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via shared memory or through message passing.

f. Error detection: OS needs to be constantly aware of possible errors. Errors may occur in the CPU and memory hardware, in I/O devices and in the user

program. For each type of error, OS takes appropriate action to ensure correct and consistent computing.

System Calls

System calls provide an interface to the services made available by an operating system.



An example to illustrate how system calls are used:

Writing a simple program to read data from one file and copy them to another file-

a) First input required is names of two files – input file and output file. Names can be specified in many ways-

One approach is for the program to ask the user for the names of two files.

In an interactive system, this approach will require a sequence of system calls, to write a prompting message on screen and then read from the keyboard the characters that define the two files.

On mouse based and icon based systems, a menu of file names is displayed in a window where the user can use the mouse to select the source names and a window can be opened for the destination name to be specified.

b) Once the two file names are obtained, program must open the input file and create the output file. Each of these operations requires another system call.

When the program tries to open input file, no file of that name may exist or file is protected against access. Program prints a message on console and terminates abnormally.

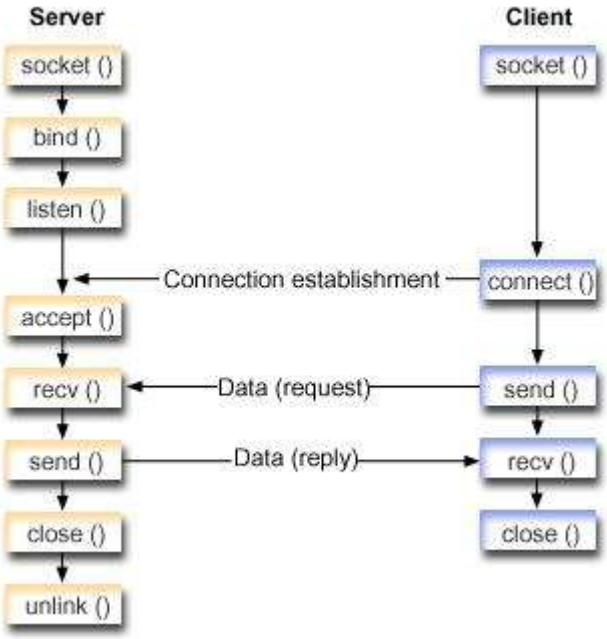
If input file exists, we must create a new output file. If the output file with the same name exists, the situation caused the program to abort or delete the existing file and create a new one. Another option is to ask the user(via a sequence of system calls) whether to replace the existing file or to abort the program.

When both files are set up, a loop reads from the input file and writes to the output file (system calls respectively). Each read and write must return status information regarding various possible error conditions. After entire file is copied, program closes both files, write a message to the console or window and finally terminate normally.

Application developers design programs according to application programming interface (API). API specifies set of functions that are available to an application programmer.

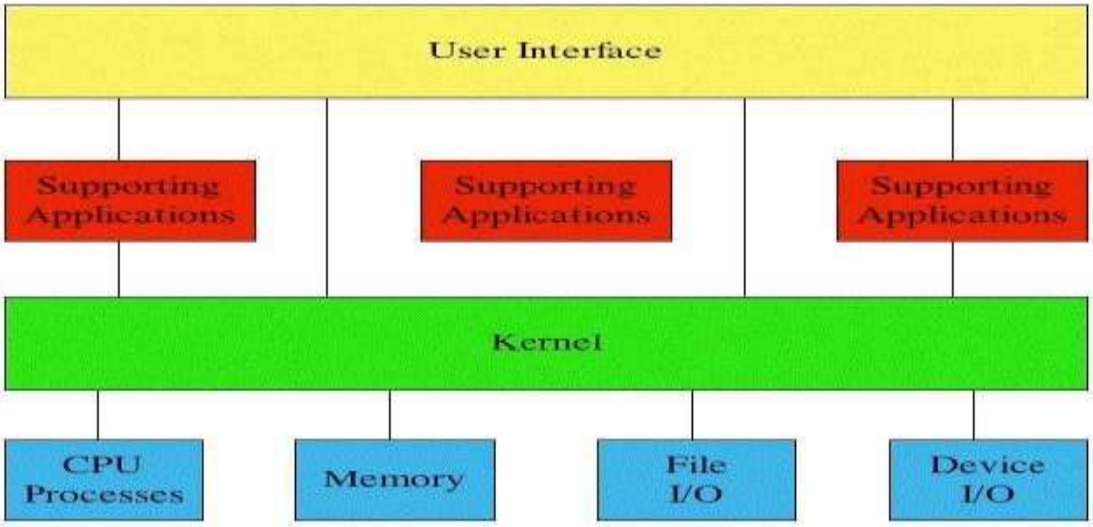
Three of the most common API's available to applicat ion programmers are the Win32API for Windows Systems; POSIX API for POSIX based systems (which include all versions of UNIX, Linux and Mac OS X) and Java API for designing programs that run on Java virtual machine.

Pictorial representation of system calls-



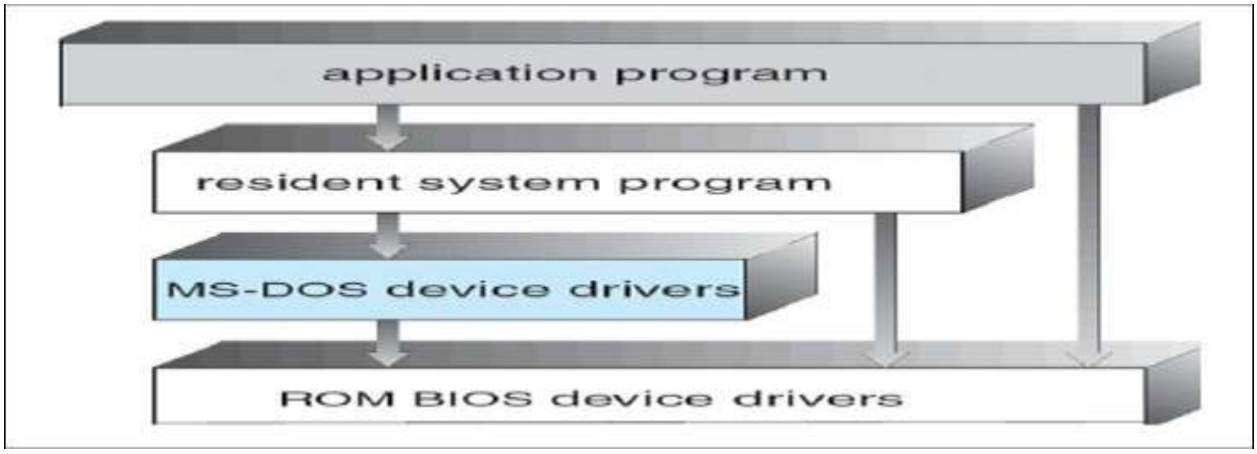
The functions that make up the API typically invoke the actual system calls on behalf of the application programmer.

Operating System Structure



Simple structure:

Operating systems of commercial systems started as a small, simple and limited systems. Example is MS-DOS. It was written to provide the most functionality in the least space, so it was not divided into modules.



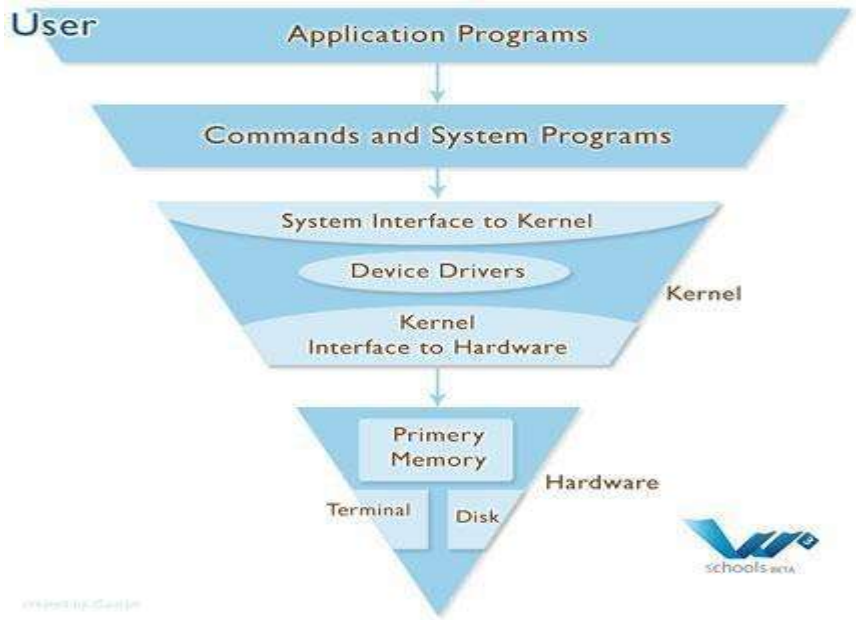
But the interfaces and levels of functionality are not separated. It was also limited by the hardware.

Layered approach:

With proper hardware support, OS can be broken into pieces that are smaller and more appropriate. OS can then retain much greater control over the computer and over the applications that make use of the computer. Under the top down approach, the overall functionality and features are determined and are separated into components.

A system can be made modular in many ways – one method is the layered approach in which the OS is broken up into number of layers (levels). The bottom layer is the hardware and the

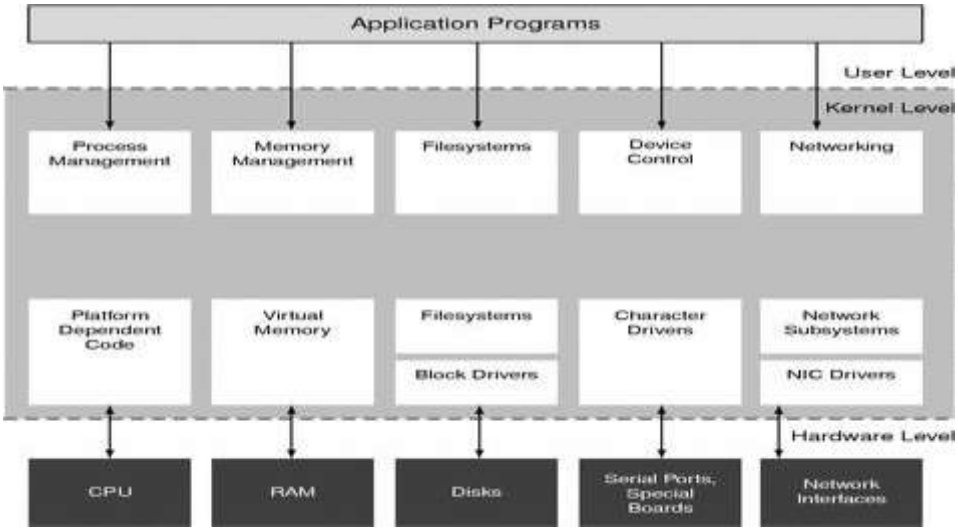
highest layer is the user
interface.



The main advantage of layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions and services of only lower level layers. This approach simplifies debugging and system verification.

The major difficulty with layered approach involves defining the various layers. They tend to be less efficient than other types.

The best current methodology for operating system design involves using object oriented programming techniques to create a modular kernel. The kernel has a set of core components and dynamically links in additional services either during boot time or run time.



UNIT 2

Concept of Process

A process is sequential program in execution. A process defines the fundamental unit of computation for the computer. Components of process are :

1. Object Program

2. Data

3. Resources

4. Status of the process execution.

Object program i.e. code to be executed. Data is used for executing the program. While executing the program, it may require some resources. Last component is used for verifying the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

Processes and Programs

Process is a dynamic entity, that is a program in execution. A process is a sequence of information executions. Process exists in a limited span of time. Two or more processes could be executing the same program, each using their own data and resources.

Program is a static entity made up of program statement. Program contains the instructions. A program exists at single place in space and continues to exist. A program does not perform the action by itself.

Process State

When process executes, it changes state. Process state is defined as the current activity of the process. Fig. 3.1 shows the general form of the process state transition diagram. Process state contains five states. Each process is in one of the states. The states are listed below.

1. New

2. Ready

3. Running

4. Waiting

5. Terminated(exist)

1. New : A process that just been created.

2. Ready : Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.

3. Running : The process that is currently being executed. A running process possesses all the resources needed for its execution, including the processor.

4. Waiting : A process that can not execute until some event occurs such as the completion of an I/O operation. The running process may become suspended by invoking an I/O module.

5. Terminated : A process that has been released from the pool of executable processes by the operating system.

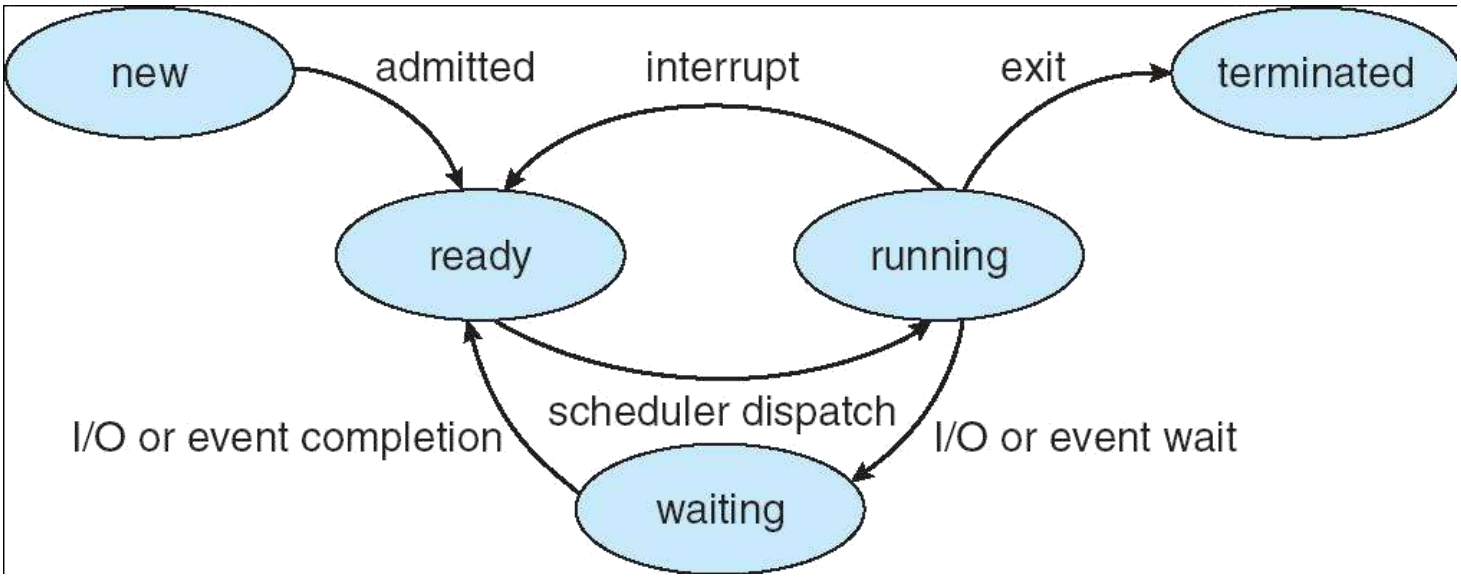


Diagram for Process State

Whenever processes changes state, the operating system reacts by placing the process PCB in the list that corresponds to its new state. Only one

process can be running on any processor at any instant and many processes may be ready and waiting state.

Suspended Processes

Characteristics of suspend process

1. Suspended process is not immediately available for execution.

2. The process may or may not be waiting on an event.

3. For preventing the execution, process is suspend by OS, parent process, process itself and an agent.

4. Process may not be removed from the suspended state until the agent orders the removal.

Swapping is used to move all of a process from main memory to disk. When all the process by putting it in the suspended state and transferring it to disk.

Reasons for process suspension

1. Swapping

2. Timing

3. Interactive user request

4. Parent process request

Swapping :

OS needs to release required main memory to bring in a process that is ready to execute.

Timing : Process may be suspended while waiting for the next time interval.

Interactive user request : Process may be suspended for debugging purpose by user.

Parent process request : To modify the suspended process or to coordinate the activity of various descendants.

3.2.2 Process Control Block (PCB)

Each process contains the process control block (PCB). PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process.

Process Management / Process Scheduling

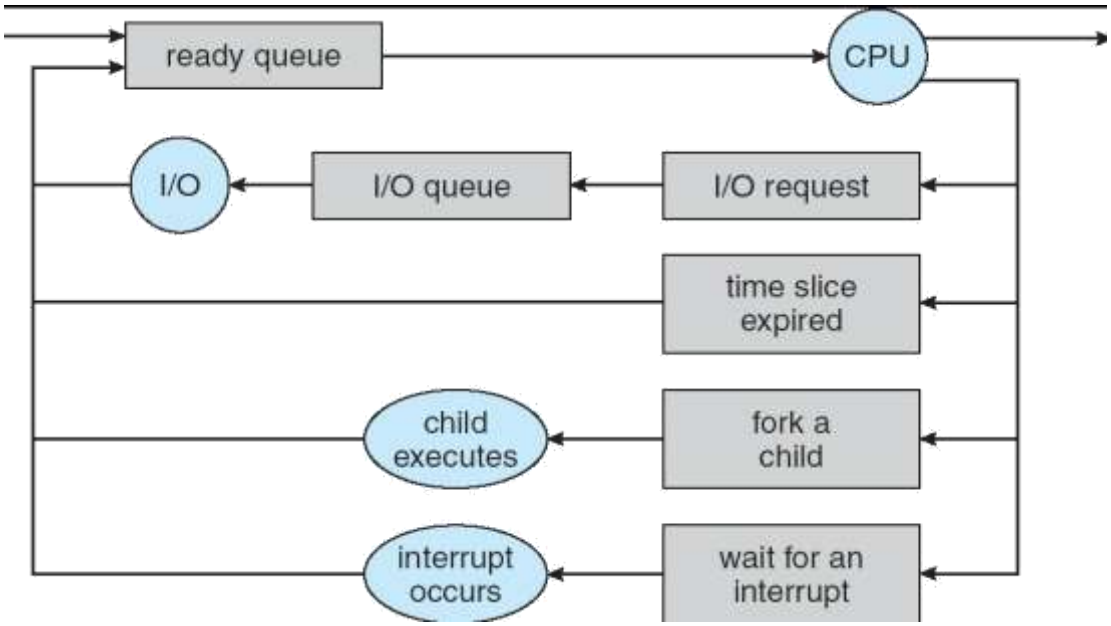
Multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded process to share the CPU using time multiplexing.

The scheduling mechanism is the part of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of particular strategy.

Scheduling Queues

When the process enters into the system, they are put into a job queue. This queue consists of all processes in the system. The operating system also has other queues.

Device queue is a queue for which a list of processes waiting for a particular I/O device. Each device has its own device queue. Fig. shows the queuing diagram of process scheduling. In the fig 3 queue is represented by rectangular box. The circles represent the resources that serve the queues. The arrows indicate the flow of processes in the system.



Schedulers

Schedulers are of three types.

1. Long Term Scheduler

2. Short Term Scheduler

3. Medium Term Scheduler

.1 Long Term Scheduler

It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduler. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

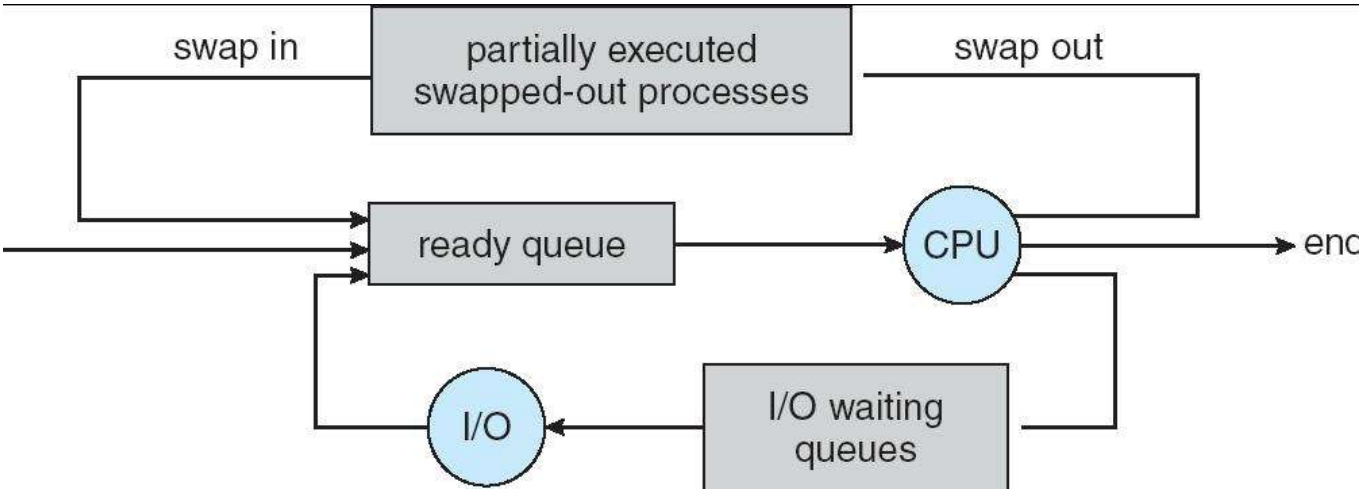
On same systems, the long term scheduler may be absent or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is a long term scheduler.

2 Short Term Scheduler

It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them. Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long tern scheduler.

.3 Medium Term Scheduler

Medium term scheduling is part of the swapping function. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in charge of handling the swapped out-processes. Medium term Scheduler is Shown as;



Queueing diagram with medium term scheduler

Running process may become suspended by making an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process. Suspended process is move to the secondary storage is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison between Scheduler:

Sr. No.	**Long Term**	**Short Term**	**Medium Term**
1	It is job scheduler	It is CPU Scheduler	It is swapping
2	Speed is less than short term scheduler	Speed is very fast	Speed is in between both
3	It controls degree of multiprogramming	Less control over degree of multiprogramming	Reduce the degree of multiprogramming.
4	Absent or minimal in time sharing system.	Minimal in time sharing system.	Time sharing system use medium term scheduler.
5	It select processes from pool and load them into memory for execution.	It select from among the processes that are ready to execute.	Process can be reintroduced into memory and its execution can be continued.
6	Process state is (New to Ready)	Process state is (Ready to Running)	-
7	Select a good process, mix of I/O bound and CPU bound.	Select a new process for a CPU quite frequently.	-

Operation on Processes

Several operations are possible on the process. Process must be created and deleted dynamically. Operating system must provide the environment for the process operation. We discuss the two main operations on processes.

1. Create a process

2. Terminate a process

Create Process

Operating system creates a new process with the specified or default attributes and identifier. A process may create several new subprocesses. Syntax for creating new process is :

CREATE (processed, attributes)

Two names are used in the process they are parent process and child process.

Parent process is a creating process. Child process is created by the parent process. Child process may create another subprocess. So it forms a tree of processes. When operating system issues a CREATE system call, it obtains a new process control block from the pool of free memory, fills the fields with provided and default parameters, and insert the PCB into the ready list. Thus it makes the specified process eligible to run the process.

When a process is created, it requires some parameters. These are priority, level of privilege, requirement of memory, access right, memory protection information etc. Process will need certain resources, such as CPU time, memory, files and I/O devices to complete the operation. When process creates a subprocess, that subprocess may obtain its resources directly from the operating system. Otherwise it uses the resources of parent process.

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

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

For address space, two possibilities occur:

1. The child process is a duplicate of the parent process.

2. The child process has a program loaded into it.

Terminate a Process

DELETE system call is used for terminating a process. A process may delete itself or by another process. A process can cause the termination of another process via an appropriate system call. The operating system reacts by reclaiming all resources allocated to the specified process, closing files opened by or for the process. PCB is also removed from its place of residence in the list and is returned to the free pool. The DELETE service is normally invoked as a part of orderly program termination.

Following are the resources for terminating the child process by parent process.

1. The task given to the child is no longer required.

2. Child has exceeded its usage of some of the resources that it has been allocated.

3. Operating system does not allow a child to continue if its parent terminates.

Co-operating Processes

Co-operating process is a process that can affect or be affected by the other processes while executing. If suppose any process is sharing data with other processes, then it is called co-operating process. Benefit of the co-operating processes are :

1. Sharing of information

2. Increases computation speed

3. Modularity

4. Convenience

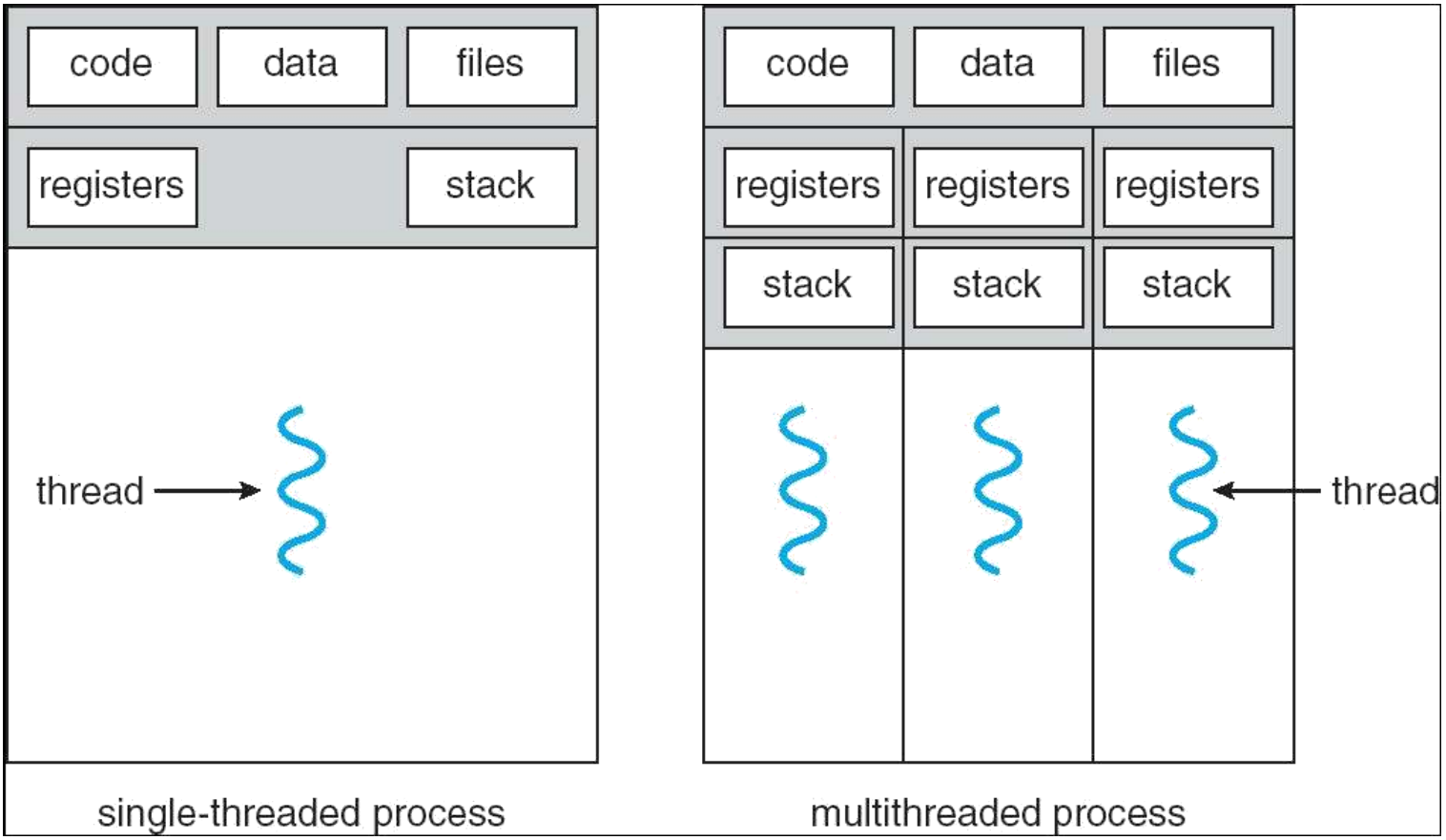
Co-operating processes share the information : Such as a file, memory etc. System must provide an environment to allow concurrent access to these types of resources. Computation speed will increase if the computer has multiple processing elements are connected together. System is constructed in a modular fashion. System function is divided into number of modules. Behavior of co-operating processes is nondeterministic i.e. it depends on relative execution sequence and cannot be predicted a priori. Co-operating processes are also Reproducible .

Introduction of Thread

A thread is a flow of execution through the process code, with its own program counter, system registers and stack. Threads are a popular way to improve application performance through parallelism. A thread is sometimes called **a light weight process.**

Threads represent a software approach to improving performance of operating system by reducing the over head thread is equivalent to a classical process. Each thread belongs to exactly one process and no thread
can exist outside a process. Each thread represents a separate flow of control.

Fig. 4.1shows the single and multithreaded process Threads



Advantages of Thread

1. Thread minimize context switching time.
2. Use of threads provides concurrency within a process.
3. Efficient communication.
4. Economy- It is more economical to create and context switch threads.
5. Utilization of multiprocessor architectures –

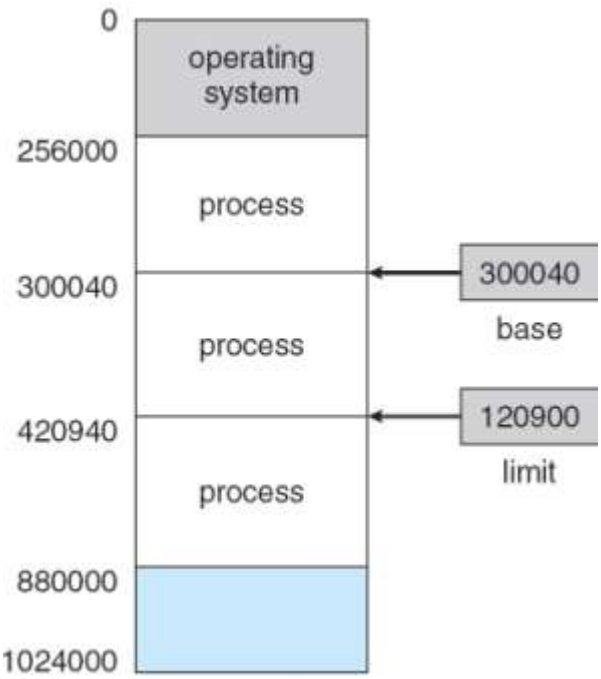
The benefits of multithreading can be greatly increased in a multiprocessor architecture.

UNIT 3

Memory Management

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address.

A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed Depending on the memory management in use the process may be moved between disk and memory during its execution. The collection of processes on the disk that are waiting to be brought into memory for execution forms the input queue. i.e. selected one of the process in the input queue and to load that process into memory. We can provide protection by using two registers, usually a base and a limit, as shown in fig. 7.1. the base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939(inclusive).



A base and limit register define a logical address space.

The binding of instructions and data to memory addresses can be done at any step along the way:

Compile time: If it is known at compile time where the process will reside in memory, then absolute code can be generated.

Load time: If it is not known at compile time where the process will reside in memory, then the compiler must generate re-locatable code.

Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Dynamic Loading

Better memory-space utilization can be done by dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed. The advantage of dynamic loading is that an unused routine is never loaded.

Dynamic Linking

Most operating systems support only static linking, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time,

linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. With dynamic linking, a stub is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routing.

The entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. So that a process can be larger than the amount of memory allocated to it, a technique called overlays is sometimes used. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.

Example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table 1, and common support routines used by both pass 1 and pass 2.

Let us consider

Pass1 70K Pass

2 80K

Symbol table 20K

Common routines 30K

To load everything at once, we would require 200K of memory. If only 150K is available, we cannot run our process. But pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2. We add an overlay driver (10K) and start with overlay A in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass 2. Overlay A

needs only 120K, whereas overlay B needs 130K As in dynamic loading, overlays do not require any special support from the operating system.

Logical versus Physical Address Space

An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit is commonly referred to as a

physical address.

The compile-time and load-time address-binding schemes result in an environment where the logical and physical addresses are the same. The execution-time address-binding scheme results in an environment where the logical and physical addresses differ. In this case, we usually refer to the logical address as a virtual address. The set of all logical addresses generated by a program is referred to as a logical address space; the set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The run-time mapping from virtual to physical addresses is done by the memory management unit (MMU), which is a hardware device.

The base register is called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 13000, then an attempt by the user to address location 0 dynamically relocated to location 14,000; an access to location 347 is mapped to location 13347. The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes.

The user program never sees the real physical addresses. The program can create a pointer to location 347 store it memory, manipulate it, compare it to other addresses all as the number 347.

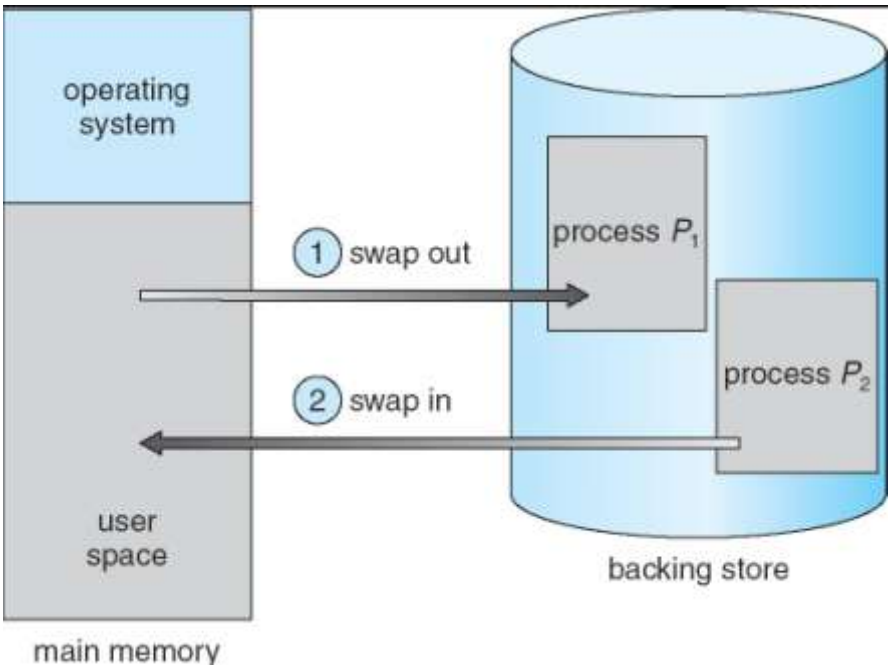
The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addressed Logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R). The user generates only logical addresses.

The concept of a logical address space that is bound to a separate
physical address space is central to proper memory management.

Swapping

A process, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. Assume a multiprogramming environment with a round robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed . Fig . When each process finishes its quantum, it will be swapped with another process.

Swapping of two processes using a disk as a blocking store



A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher priority process. When the higher priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called rollout, roll in. A process is swapped out will be swapped back into the same memory space that it occupies previously. If binding is done at assembly or load time, then the process cannot be moved to different location. If execution-time binding is being used, then it is possible to swap a process into a different memory space.

Swapping requires a backing store. The backing store is commonly a fast disk. It is large enough to accommodate copies of all memory images for all users. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

The context-switch time in such a swapping system is fairly high. Let us assume that the user process is of size 100K and the backing store is a standard hard disk with transfer rate of 1 megabyte per second. The actual transfer of the 100K process to or from memory takes

100K / 1000K per second = 1/10 second
= 100 milliseconds

Contiguous Allocation

The main memory must accommodate both the operating system and the various user processes. The memory is usually divided into two partitions, one for the resident operating system, and one for the user processes.

To place the operating system in low memory. Thus, we shall discuss only me situation where the operating system resides in low memory (Figure 8.5). The development of the other situation is similar. Common Operating System is placed in low memory.

7.3.1 Single-Partition Allocation

If the operating system is residing in low memory, and the user processes are executing in high memory. And operating-system code and data are protected from changes by the user processes. We also need protect the user processes from one another. We can provide this 2 protection by using a relocation registers.

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100,040 and limit = 74,600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

The relocation-register scheme provides an effective way to allow the operating system size to change dynamically.

Multiple-Partition Allocation

One of the simplest schemes for memory allocation is to divide memory into a number of fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block, of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process.

For example, assume that we have 2560K of memory available and a resident operating system of 400K. This situation leaves 2160K for user processes. FCFS job scheduling, we can immediately allocate memory to processes P1, P2, P3. Holes size 260K that cannot be used by any of the remaining processes in the input queue. Using a round-robin CPU-scheduling with a quantum of 1 time unit, process will terminate at time 14, releasing its memory.

Memory allocation is done using Round-Robin Sequence as shown in fig. When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, we merge these adjacent holes to form one larger hole.

This procedure is a particular instance of the general dynamic storage-allocation problem, which is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate, first-fit, best-fit, and worst-fit are the most common strategies used to select a free hole from the set of available holes.

First-fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best-fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy-produces the smallest leftover hole.

Worst-fit: Allocate the largest hole. Again, we must search the entire list unless it is sorted by size. This strategy produces the largest leftover hole which may be more useful than the smaller leftover hole from a best-t approach.

External and Internal Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough to the memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes.

Depending on the total amount of memory storage and the average process size, external fragmentation may be either a minor or a major problem.

Given N allocated blocks, another 0.5N blocks will be lost due to fragmentation. That is, one-third of memory may be unusable. This property is known as the 50- percent rule. Internal fragmentation - memory that is internal to partition, but is not being used.

Paging

External fragmentation is avoided by using paging. In this physical memory is broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames. Every address generated by the CPU is divided into any two parts: a page number(p) and a page offset(d) (Fig 7.3). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the gage offset to define the physical memory address that is sent to the memory unit.

Paging Hardware

The page size like is defined by the hardware. The size of a page is typically a power of 2 varying between 512 bytes and 8192 bytes per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset. lf the size of logical address space is 2m, and

a page size is 2n addressing units (bytes or words), then the high-order m - n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

page number

page offset p

d

m – n n

where p is an index into the page table and d is the displacement within the page.

Paging is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address.

When we use a paging scheme, we have no external fragmentation: Any free
frame can be allocated to a process that needs it.

If process size is independent of page size, we can have internal fragmentation to average one-half page per process. When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, there must be at least n frames available in memory. If there are n frames available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on.

The user program views that memory as one single contiguous space, containing only this one program. But the user program is scattered throughout physical memory and logical addresses are translated into physical addresses.

The operating system is managing physical memory, it must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free allocated and, if it is allocated, to which page of which process or processes.

The operating system maintains a copy of the page table for each process. Paging therefore increases the context-switch time.

Segmentation

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of **segments.** It is not required that all segments of all programs be of the same length, although there is a maximum segment length. As with paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning. In segmentation, a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates

internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less. Whereas paging is invisible to the programmer, segmentation usually visible and is provided as a convenience for organizing programs and data.

Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses. Segmentation scheme would make use of a segment table for each process and a list of free blocks of main memory. Each segment table entry would have to as in paging give the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment, to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware.

Consider an address of n + m bits, where the leftmost n bits are the segment number and the rightmost m bits are the offset. The following steps are needed for address translation:
Extract the segment number as the leftmost n bits of the logical address.

Use the segment number as an index into the process segment table to find the starting physical address of the segment. Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid. The desired physical address is the sum of the starting physical address of the segment plus the offset. Segmentation and paging can be combined to have a good result.

Virtual Memory

Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory.

Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Fig 8.1). Following are the situations, when entire program is not required to load fully.

1. User written error handling routines are used only when an error occurs in the data or computation.

2. Certain options and features of a program may be used rarely.

3. Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

The ability to execute a program that is only partially in memory would counter many benefits.

1. Less number of I/O would be needed to load or swap each user program into memory.

2. A program would no longer be constrained by the amount of physical memory that is available.

3. Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

Virtual memory is commonly implemented by demand paging. It can also be

implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

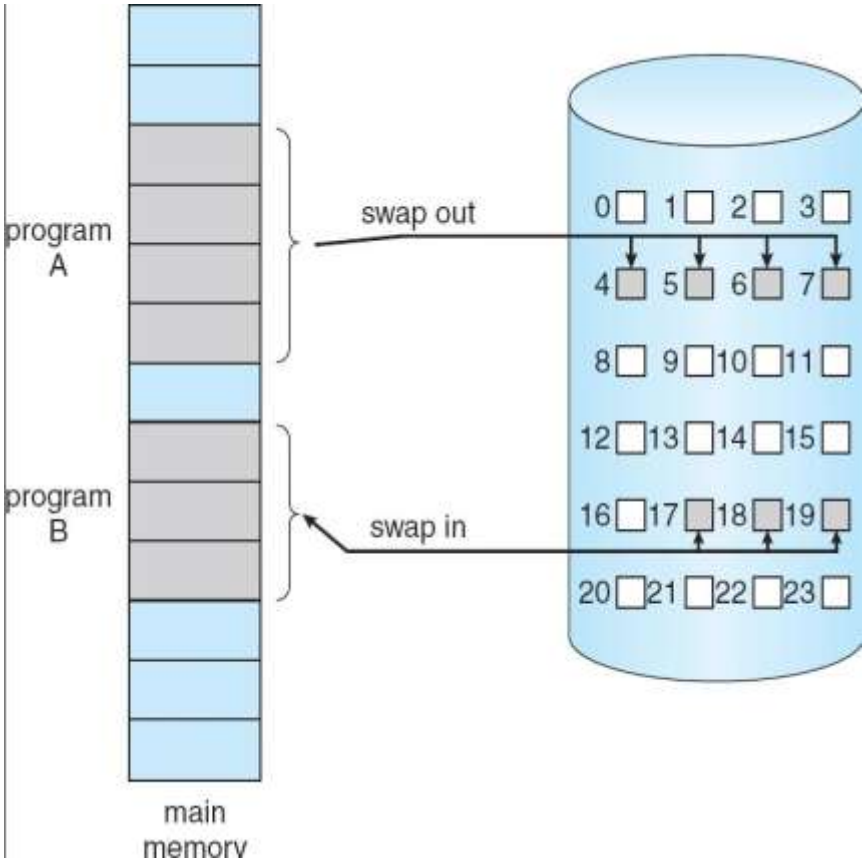
Demand Paging

A demand paging is similar to a paging system with swapping(Fig 8.2). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory.

When a process is to be swapped in, the pager guesses which pages will be

used before the process is swapped out again Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme. Where valid and invalid pages can be checked checking the bit and marking a page will have no effect if the process never attempts to access the pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.



Transfer of a paged memory to continuous disk space

Advantages of Demand Paging:

1. Large virtual memory.

2. More efficient use of memory.

3. Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

Disadvantages of Demand Paging:

1. Number of tables and amount of processor over head for handling page interrupts are greater than in the case of the simple paged management techniques.

2. due to the lack of an explicit constraints on a jobs address space size.

Page Replacement Algorithm

There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults. The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data.

1. For a given page size we need to consider only the page number, not the
entire address.

2. if we have a reference to a page p, then any immediately following references to
page p will never cause a page fault. Page p will be in memory after the

first reference; the immediately following references will not fault.
Eg:- consider the address sequence
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0104, 0101, 0609, 0102, 0105 and reduce to 1, 4, 1, 6,1, 6, 1, 6, 1, 6, 1

To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available. As the number of frames available increase, the number of page faults
will decrease.

FIFO Algorithm

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory.

The first three references (7, 0, 1) cause page faults, and are brought into these empty eg. 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1 and consider 3 frames. This replacement means that the next reference to 0 will fault. Page 1 is then replaced by page 0.

Optimal Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply

Replace the page that will not be
used for the longest period of time.

Now consider the same string with 3 empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. Optimal replacement is much better than a FIFO.

The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Algorithm

The FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. In LRU replace the

page that has not been used for the longest period of time.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

Let SR be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on SR.

LRU Approximation Algorithms

Some systems provide no hardware support, and other page-replacement algorithm. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced. Reference bits are associated with each entry in the page table Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

Additional-Reference-Bits Algorithm

The operating system shifts the reference bit for each page into the high-order or of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit.

These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been

used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

UNIT 4

DEADLOCKS

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called deadlock. If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.

2. **Use:** The process can operate on the resource.

3. **Release**: The process releases the resource

Deadlock Characterization

In deadlock, processes never finish executing and system resources are tied up, preventing other jobs from ever starting.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait :** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption :** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process, has completed its task.

4. **Circular wait:** There must exist a set {P0, P1, ..., Pn } of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …., Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. The set of vertices V is partitioned into two different types of nodes P = {P1, P2, … Pn} the set consisting of all the active

processes in the system; and R = {R1, R2, …, R1}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi → Rj, it signifies that process Pi requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj toprocess Pi is denoted by Rj_ Pi it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi_ Rj is called a request edge; a directed edge Rj _ Pi is called an assignment edge.

When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the, resource it releases the resource, and as a result the assignment edge is deleted. Definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains the cycle, then a deadlock must exist.

If each resource type has several instances, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resources types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

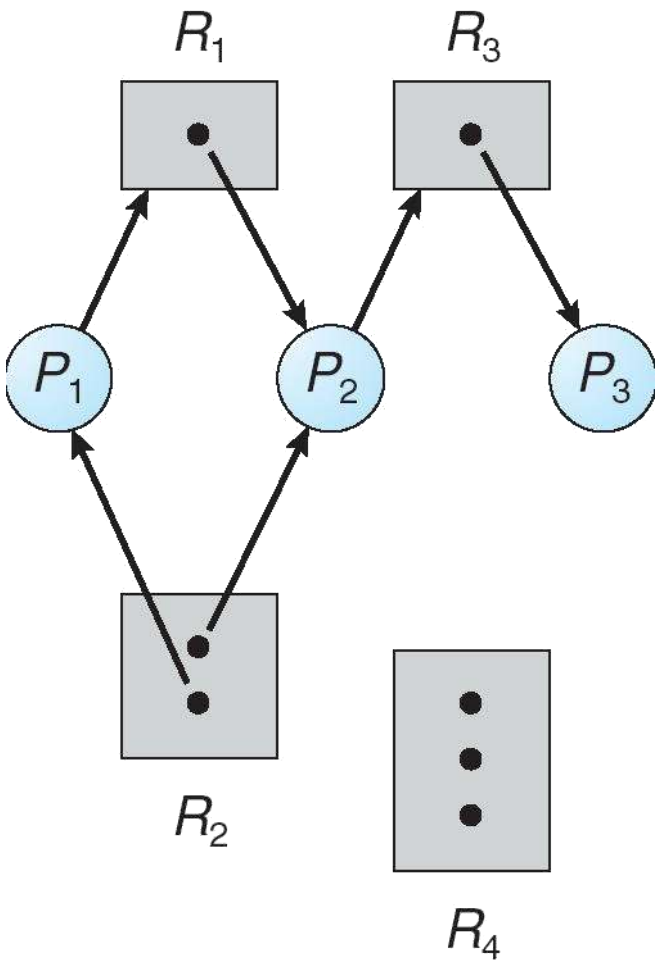
A set of vertices *V* and a set of edges *E*.

V is partitioned into two types:

o *P* = {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system.

o *R* = {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system. request edge – directed edge *P*1 *Rj*

assignment edge – directed edge *Rj Pi*

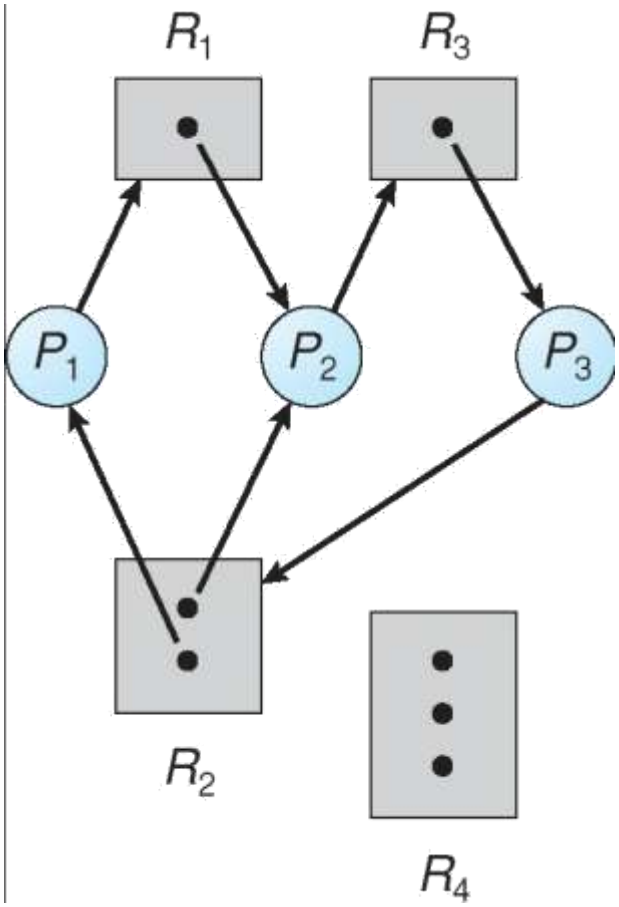


Resource Allocation Graph

If each resource type has several instance, then a cycle does not necessarily imply that a deadlock incurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Suppose that process P3requests an instance of resource type R2 Since no resource instance is currently available, a request edge P3 → R2 is added to the graph. At this point, two minimal cycles exist in the system:

P1 → R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2



Resource Allocation Graph with Deadlock

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process P1 or process P2 to release resource R2. In addition, process PI is

waiting for process P2 to release resource R1.

Method For Handling Deadlock //Detection

There are are three different methods for dealing with the deadlock problem:

• We can use a protocol to ensure that the system will never enter a deadlock state.

• We can allow the system to enter a deadlock state and then recover. • We can ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. Each request requires that the system consider the

resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened.

Deadlock Prevention

For a deadlock to occur, each of the four necessary-conditions must hold. By ensuring that at least on one these conditions cannot hold, we can prevent the occurrence of a deadlock.

Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock.

Hold and Wait

1. When whenever a process requests a resource, it does not hold any other resources. One protocol that be used requires each process to request and be allocated all its resources before it begins execution.

2. An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated here are two main disadvantages to these protocols. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.

Second, starvation is possible.

No Preemption

If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted. That is this resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait

Circular-wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration. Let R = {R1, R2, ..., Rn} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to

determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R _ N, where N is the set of natural numbers.

Deadlock Avoidance

Prevent deadlocks requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this, melted, however, are Tow device utilization and reduced system throughput.

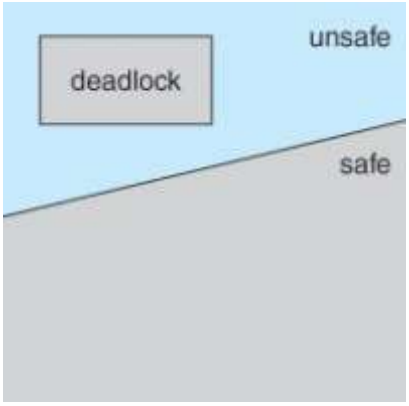
An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process we can decide for each request whether or not the process should wait.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular wait condition. The resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in

a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, .. Pn> is a safe sequence for the current allocation state if, for each Pi the resources that Pj can still request can be satisfied by the currently available resources plus the resources held by all the Pj, with j < i. In this situation, if the resources that process Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task return its allocated resources, and terminate. When Pi terminates, Pi + 1 can obtain its needed resources, and so on.



Safe, Unsafe & Deadlock State

Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm.

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

• An algorithm that examines the state of the system to determine whether a deadlock has occurred.

• An algorithm to recover from the deadlock.

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

The algorithm used are :

• **Available**: A vector of length m indicates the number of available resources of each type.

• **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.

• **Request**: An n x m matrix indicates the current request of each process. If Request [i, j] = k, then process P, is requesting k more instances of resource type Rj.

Detection-Algorithm Usage

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has spurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

• **Abort all deadlocked processes**: This method clearly will break the dead – lock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed.

• **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted a deadlock-detection algorithm must be invoked to determine whether a processes are still deadlocked.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until he deadlock cycle is broken.

The three issues are considered to recover from deadlock

1. **Selecting a victim**
2. **Rollback**
3. **Starvation**

UNIT 5

Disk Structure

Disk provide bulk of secondary storage of computer system. The disk can be considered the one I/O device that is common to each and every computer. Disks come in many size and speeds, and information may be stored optically or magnetically. Magnetic tape was used as an early secondary storage medium, but the access time is much slower than for disks. For backup, tapes are currently used.

Modern disk drives are addressed as large one dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The actual details of disk I/O operation depends on the computer system, the operating system and the nature of the I/O channel and disk controller hardware.

The basic unit of information storage is a sector. The sectors are stored on a flat, circular, media disk. This media spins close to one or more read/write heads. The heads can move from the inner portion of the disk to the outer portion.

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. Track selection involves moving the head in a movable head system or electronically selecting one head on a fixed head system. These characteristics are common to floppy disks, hard disks, CD-ROM and DVD.

Disk Performance Parameters

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track.

Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system. On a movable-head system, the time it takes to position the head at the track is known as **seek time.**

When once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes for the beginning of the sector to reach the head is known as **rotational delay,** or rotational latency. The sum of the seek time, if any, and the rotational delay equals the **access time,** which is the time it takes to get into position to read or write.

Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation; the time required for the transfer is the **transfer time.**

Seek Time Seek time is the time required to move the disk arm to the required track. It turns out that this is a difficult quantity to pin down. The seek time consists of two key components: the initial startup time and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed.

Ts = m x n + s

Rotational Delay Disks, other than floppy disks, rotate at speeds ranging from 3600 rpm up to, as of this writing, 15,000 rpm; at this latter speed, there is one revolution per 4 ms. Thus, on the average, the rotational delay will be 2 ms. Floppy disks typically rotate at between 300 and 600 rpm. Thus the average delay will be between 100 and 50 ms.

Transfer Time The transfer time to or from the disk depends on the rotation speed of the disk in the following fashion:

T= b/rN

where

T = transfer time

b = number of bytes to be transferred

N = number of bytes on a track

r = rotation speed, in revolutions per second

Thus the total average access time can be expressed as

Ta = Ts +

where Ts is the average seek time.

Disk Scheduling

The amount of head needed to satisfy a series of I/O request can affect the performance. If desired disk drive and controller are available, the request can be serviced immediately. If a device or controller is busy, any new requests for service will be placed on the queue of pending requests for that drive. When one request is completed, the operating system chooses which pending request to service next.

Different types of scheduling algorithms are as follows.

1. First Come, First Served scheduling algorithm(FCFS).

2. Shortest Seek Time First (SSTF) algorithm

3. SCAN algorithm

4. Circular SCAN (C-SCAN) algorithm

5. Look Scheduling Algorithm

First Come, First Served scheduling algorithm(FCFS).

The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order. This strategy has the advantage of being fair, because every request is honored and the requests are honored in the order received. With FIFO, if there are only a few processes that require access and if many of the requests are to clustered file sectors, then we can hope for good performance.

Priority With a system based on priority (PRI), the control of the scheduling is outside the control of disk management software.

Last In First Out ln transaction processing systems, giving the device to the most recent user should result. In little or no arm movement for moving through a sequential file. Taking advantage of this locality improves throughput and reduces queue length.

Shortest Seek Time First (SSTF) algorithm

The SSTF policy is to select the disk I/O request the requires the least movement of the disk arm from its current position. **Scan** With the exception of FIFO, all of the policies described so far can leave some request unfulfilled until the entire queue is emptied. That is, there may always be new requests arriving that will be chosen before an existing request.

The choice should provide better performance than FCFS algorithm.

Under heavy load, SSTF can prevent distant request from ever being serviced. This phenomenon is known as starvation. SSTF scheduling is

essentially a from of shortest job first scheduling. SSTF scheduling algorithm are not very popular because of two reasons.

1. Starvation possibly exists.

2. it increases higher overheads.

SCAN scheduling algorithm

The scan algorithm has the head start at track 0 and move towards the highest numbered track, servicing all requests for a track as it passes the track. The service direction is then reserved and the scan proceeds in the opposite direction, again picking up all requests in order.

SCAN algorithm is guaranteed to service every request in one complete pass through the disk. SCAN algorithm behaves almost identically with the SSTF algorithm. The SCAN algorithm is sometimes called elevator algorithm.

C SCAN Scheduling Algorithm

The C-SCAN policy restricts scanning to one direction only. Thus, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again.

This reduces the maximum delay experienced by new requests.

LOOK Scheduling Algorithm

Start the head moving in one direction. Satisfy the request for the closest track in that direction when there is no more request in the direction, the head is traveling, reverse direction and repeat. This algorithm is similar to innermost and outermost track on each circuit.

Disk Management

Operating system is responsible for disk management. Following are some activities discussed.

Disk Formatting

Disk formatting is of two types.

a) Physical formatting or low level formatting.

b) Logical Formatting

Physical Formatting

Disk must be formatted before storing data.

Disk must be divided into sectors that the disk controllers can read/write.

Low level formatting files the disk with a special data structure for each sector.

Data structure consists of three fields: header, data area and trailer.

Header and trailer contain information used by the disk controller.

Sector number and Error Correcting Codes (ECC) contained in the header and

trailer. For writing data to the sector – ECC is updated.

For reading data from the sector – ECC is recalculated.

Low level formatting is done at factory.

Logical Formatting

After disk is partitioned, logical formatting used.

Operating system stores the initial file system data structures onto the disk.

Boot Block

When a computer system is powered up or rebooted, a program in read only memory executes. Diagnostic check is done first.

Stage 0 boot program is executed.

Boot program reads the first sector from the boot device and contains a stage-1 boot program.

May be boot sector will not contain a boot program.

PC booting from hard disk, the boot sector also contains a partition table.

The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code. Full boot strap program is more sophisticated than the bootstrap loader in the boot ROM.

Swap Space Management

Swap space management is low level task of the operating system. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system.

Swap-Space Use

The operating system needs to release sufficient main memory to bring in a process that is ready to execute. Operating system uses this swap space in various way. Paging systems may simply store pages that have been pushed out of main memory. Unix operating system allows the use of multiple swap space are usually put on separate disks, so the load placed on the I/O system by paging and swapping can be spread over the systems I/O devices.

Swap Space Location

Swap space can reside in two places:

1. Separate disk partition
2. Normal file System

If the swap space is simply a large file within the file system, normal file system routines can be used to create it, name it and allocate its space. This is easy to implement but also inefficient. External fragmentation can greatly increase swapping times. Catching is used to improve the system performance. Block of information is cached in the physical memory, and by using special tools to allocate physically continuous blocks for the swap file.

Swap space can be created in a separate disk partition. No file system or directory structure is placed on this space. A separate swap space storage manager is used to allocate and deallocate the blocks. This manager uses algorithms optimized for speed. Internal fragmentation may increase. Some operating systems are flexible and can swap both in raw partitions and in file system space.

Stable Storage Implementation

The write ahead log, which required the availability of stable storage.

By definition, information residing in stable storage is never lost.

To implement such storage, we need to replicate the required information on multiple storage devices (usually disks) with independent failure modes.

We also need to coordinate the writing of updates in a way that guarantees that a failure during an update will not leave all the copies in a damaged state and that, when we are recovering from failure, we can force all copies to a consistent and correct value, even if another failure occurs during the recovery.

Disk Reliability

Good performance means high speed, another important aspect of performance is reliability.

A fixed disk drive is likely to be more reliable than a removable disk or tape drive.

An optical cartridge is likely to be more reliable than a magnetic disk or tape.

A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

Chapter 3

Loaders and Linkers

This Chapter gives you…

Basic Loader Functions
Machine-Dependent Loader Features
Machine-Independent Loader Features
Loader Design Options
Implementation Examples

2. Introduction

The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This

conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

Loading - which allocates memory location and brings the object program into memory for execution - (Loader)

Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)

Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

3.1 Basic Loader Functions

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 3.1. In figure 3.1 translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution. In figure 3.2 the translator is specifically an assembler, which generates the object loaded, which becomes input to the loader. The figure 3.3 shows the role of both loader and linker.

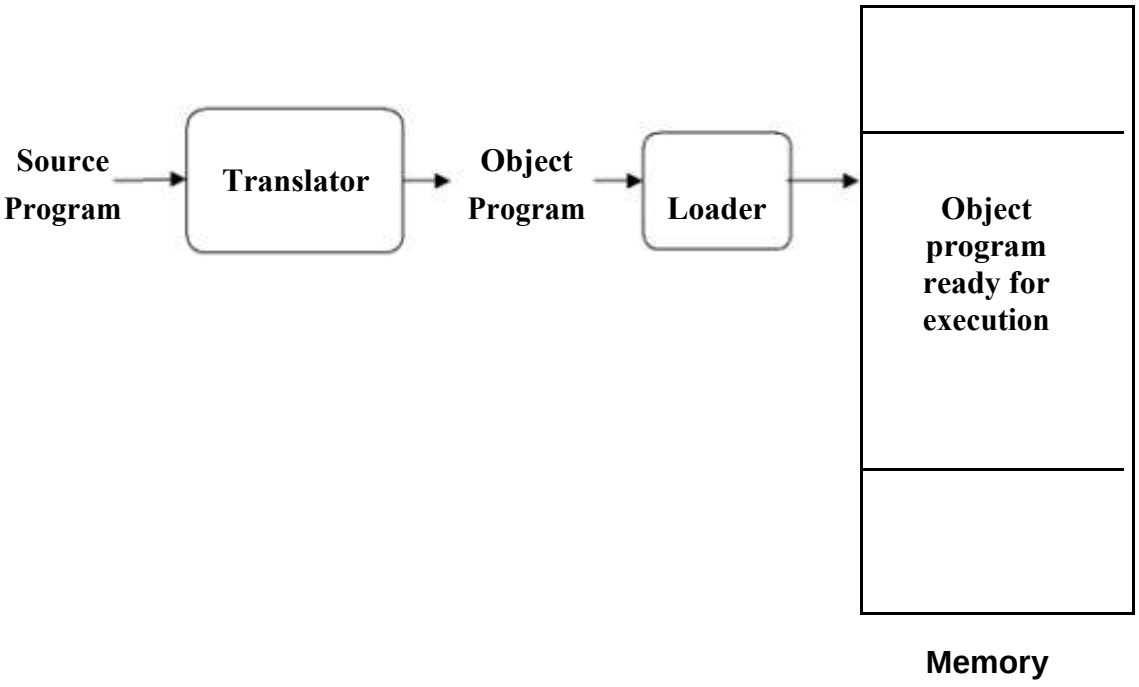


Figure 3.1 : The Role of Loader

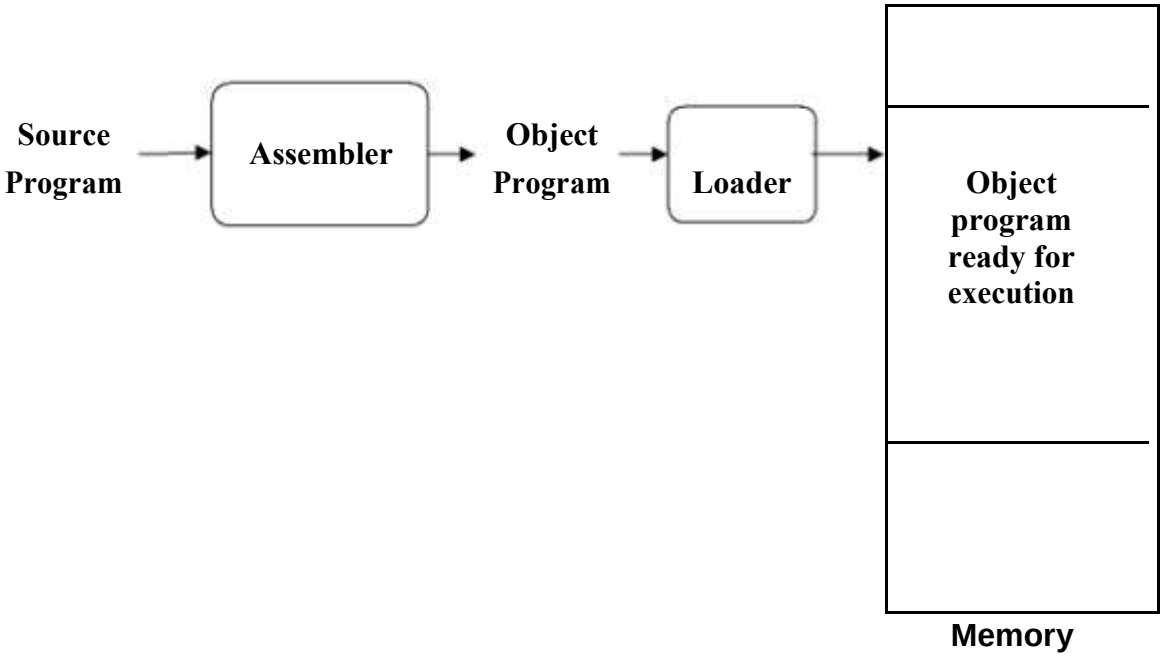


Figure 3.2: The Role of Loader with Assembler

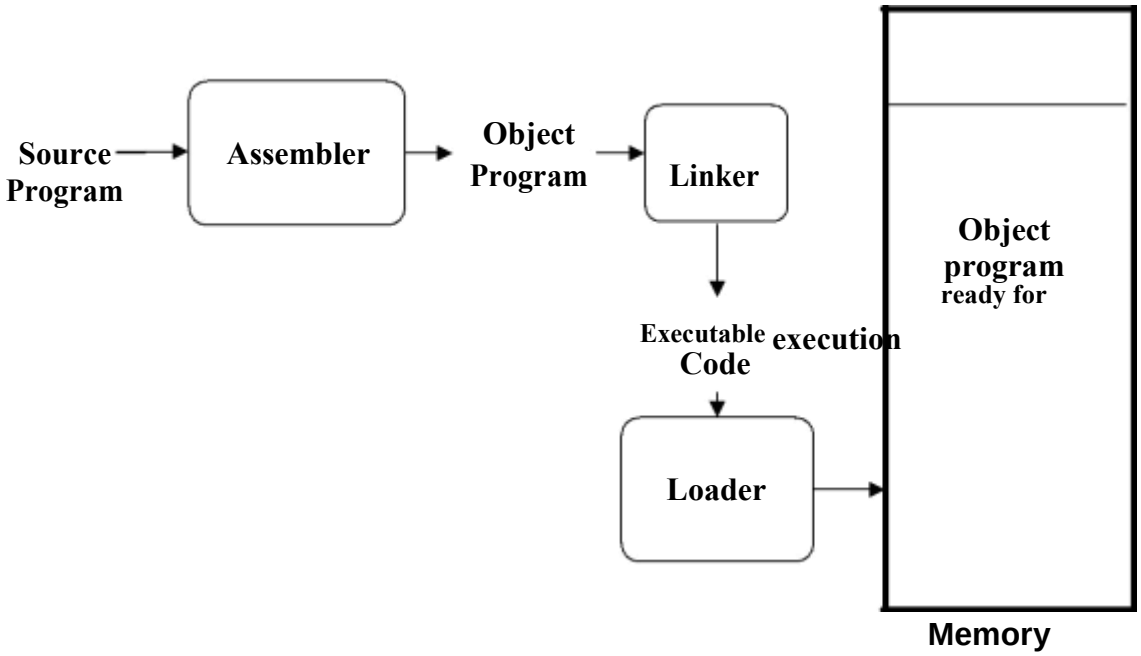


Figure 3.3 : The Role of both Loader and Linker

3.3 Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader. The following sections discuss the functions and design of all these types of loaders.

3.3.1 Absolute Loader

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader is as shown in the figure 3.3.1. The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

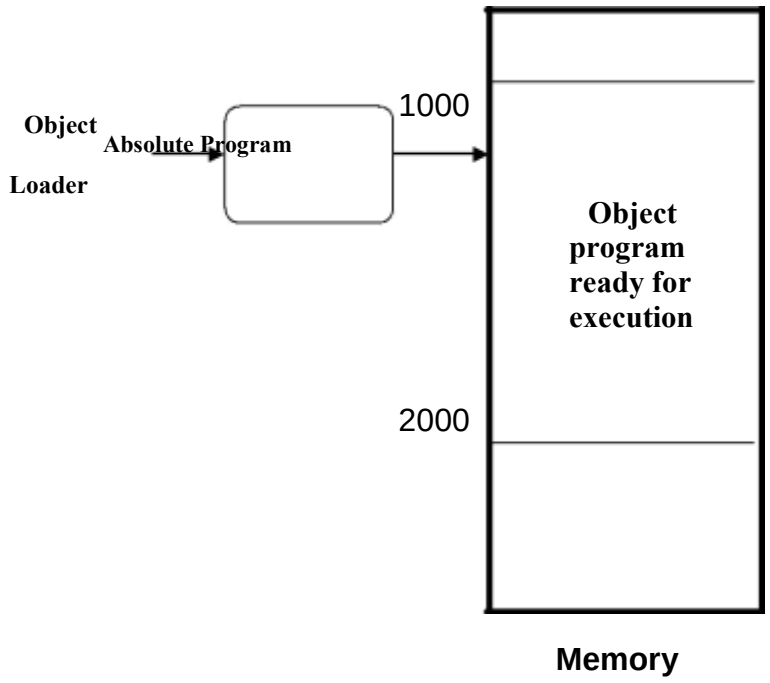


Figure 3.3.1: The Role of Absolute Loader

The algorithm for this type of loader is given here. The object program and, the object program loaded into memory by the absolute loader are also shown. Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte. Most machine store object programs in a binary form, and we must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters.

Begin
read Header record
verify program name and length
read first Text record
while record type is <> 'E'
    do begin
    {if object code is in character form, convert into internal representation}
    move object code to specified location in memory
    read next object program record
    end
jump to address specified in End
record **end**

```
HCOPY  001000001007A
T001000 1E 14103348 20390010 36281030 30101548 20613C10 0300102A 0C103900 102D0C10
T00101E 15 0C103648 2061081033 4C0000454F 46 000003000000
T0020391E 041030001030 E0205030203F D8205D 2810303020575490392C205E38203F
T00205710 1010364C0000 F001000041030 E02079302064509039DC2079 2C1036
T002073073820644C000005
E001000
```

(a) Object program

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
1000	14103348	20390010	36281030	30101548	
1010	20613C10	0300102A	0C103900	102D0C10	
1020	36482061	08103340	00004541	46000003	
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	← COPY
⋮	⋮	⋮	⋮	⋮	
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0	
2040	205D3020	3FD8205D	28103030	20575490	
2050	392C205E	38203F10	10364C00	00F10010	
2060	00041030	E0207930	20645090	39DC2079	
2070	2C103638	20644C00	0005xxxx	xxxxxxxx	
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

(b) Program loaded in memory

3.3.2 A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

Begin
X=0x80 (the address of the next memory location to be loaded
Loop
A←GETC (and convert it from the ASCII
character code to the value of the hexadecimal digit)
save the value in the high-order 4 bits of
S A←GETC
combine the value to form one byte A← (A+S)
store the value (in A) to the address in register X
X←X+1
End

It uses a subroutine GETC, which is

GETC A←read one character
if A=0x04 then jump to 0x80
if A<48 then GETC
A ← A-48 (0x30)
if A<10 then return
A ← A-7
return

3.4 Machine-Dependent Loader Features

Absolute loader is simple and efficient, but the scheme has potential disadvantages One of the most disadvantage is the programmer has to specify the actual starting address, from where the program to be loaded. This does not create difficulty, if one program to run, but not for several programs. Further it is difficult to use subroutine libraries efficiently.

This needs the design and implementation of a more complex loader. The loader must provide program relocation and linking, as well as simple loading functions.

3.4.1 Relocation

The concept of program relocation is, the execution of the object program using any part of the available and sufficient memory. The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time. Relocation provides the efficient sharing of the machine with larger memory and when several independent programs are to be run together. It also supports the use of subroutine libraries efficiently. Loaders that allow for program relocation are called relocating loaders or relative loaders.

3.4.2 Methods for specifying relocation

Use of modification record and, use of relocation bit, are the methods available for specifying relocation. In the case of modification record, a modification record M is used in the object program to specify any relocation. In the case of use of relocation bit, each instruction is associated with one relocation bit and, these relocation bits in a Text record is gathered into bit masks.

Modification records are used in complex machines and is also called Relocation and Linkage Directory (RLD) specification. The format of the modification record (M) is as follows. The object program with relocation by Modification records is also shown here.

Modification
record col 1: M
col 2-7: relocation address
col 8-9: length (halfbyte) col
10: flag (+/-)
col 11-17: segment name

H∧COPY ∧000000 001077
T∧000000 ∧1D∧17202D∧69202D∧48101036∧…∧4B105D∧3F2FEC∧032010
T∧00001D∧13∧0F2016∧010003∧0F200D∧4B10105D∧3E2003∧454F46
T∧001035 ∧1D∧B410∧B400∧B440∧75101000∧…∧332008∧57C003∧B850
T∧001053∧1D∧3B2FEA∧134000∧4F0000∧F1∧..∧53C003∧DF2008∧B850
T∧00070∧07∧3B2FEF∧4F0000∧05
M∧000007∧05+COPY
M∧000014∧05+COPY
M∧000027∧05+COPY
E∧000000

The relocation bit method is used for simple machines. Relocation bit is 0: no modification is necessary, and is 1: modification is needed. This is specified in the columns 10-12 of text record (T), the format of text record, along with relocation bits is as follows.

Text record
col 1: T
col 2-7: starting address
col 8-9: length (byte) col
10-12: relocation bits
col 13-72: object code

Twelve-bit mask is used in each Text record (col:10-12 – relocation bits), since each text record contains less than 12 words, unused words are set to 0, and, any value that is to be modified during relocation must coincide with one of these 3-byte segments. For absolute loader, there are no relocation bits column 10-69 contains object code. The object program with relocation by bit mask is as shown below. Observe FFC - means all ten words are to be modified and, E00 - means first three records are to be modified.

H∧COPY ∧000000 00107A
T∧000000∧1E∧FFC∧140033∧481039∧000036∧280030∧300015∧…∧3C0003 ∧ …
T∧00001E∧15∧E00∧0C0036∧481061∧080033∧4C0000∧…∧000003∧000000
T∧001039∧1E∧FFC∧040030∧000030∧…∧30103F∧D8105D∧280030∧...
T∧001057∧0A∧ 800∧100036∧4C0000∧F1∧001000
T∧001061∧19∧FE0∧040030∧E01079∧…∧508039∧DC1079∧2C0036∧...
E∧000000

3.5 Program Linking

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

EXTDEF (external definition) - The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.

ex: EXTDEF BUFFER, BUFFEND, LENGTH
EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.

ex: EXTREF RDREC, WRREC
EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF

The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

Define record

The format of the Define record (D) along with examples is as shown here**.**

Col. 1	D
Col. 2-7	Name of external symbol defined in this control section
Col. 8-13	Relative address within this control section (hexadecimal)
Col.14-73	Repeat information in Col. 2-13 for other external symbols

Example records

D LISTA 000040 ENDA 000054
D LISTB 000060 ENDB 000070

Refer record

The format of the Refer record (R) along with examples is as shown here**.**

Col. 1	R
Col. 2-7	Name of external symbol referred to in this control section
Col. 8-73	Name of other external reference symbols

Example records

R LISTB ENDB LISTC ENDC
R LISTA ENDA LISTC ENDC
R LISTA ENDA LISTB ENDB

Here are the three programs named as PROGA, PROGB and PROGC, which are separately assembled and each of which consists of a single control section. LISTA, ENDA in PROGA, LISTB, ENDB in PROGB and LISTC, ENDC in PROGC are external definitions in each of the control sections. Similarly LISTB, ENDB, LISTC, ENDC in PROGA, LISTA, ENDA, LISTC, ENDC in PROGB, and LISTA, ENDA, LISTB, ENDB in PROGC, are external references. These sample programs given here are used to illustrate linking and relocation. The following figures give the sample programs and their corresponding object programs. Observe the object programs, which contain D and R records along with other records.

```
0000  PROGA     START   0
                EXTDEF  LISTA, ENDA
                EXTREF  LISTB, ENDB, LISTC, ENDC
                ……..
                 …….
0020  REF1      LDA     LISTA                    03201D
0023  REF2      +LDT    LISTB+4                  77100004
0027  REF3      LDX     #ENDA-LISTA              050014
                    .
                    .
0040  LISTA     EQU     *

0054  ENDA      EQU     *
0054  REF4      WORD        ENDA-LISTA+LISTC     000014
0057  REF5      WORD        ENDC-LISTC-10        FFFFF6
005A  REF6      WORD        ENDC-LISTC+LISTA-1   00003F
005D  REF7      WORD        ENDA-LISTA-(ENDB-LISTB) 000014
0060  REF8      WORD        LISTB-LISTA          FFFFC0
                 END     REF1
```

```
0000  PROGB     START   0
                EXTDEF  LISTB, ENDB
                EXTREF  LISTA, ENDA, LISTC, ENDC
                ……..
                 …….
0036  REF1      +LDA    LISTA                    03100000
003A  REF2      LDT     LISTB+4                    772027
003D  REF3      +LDX    #ENDA-LISTA              05100000
                    .
                    .
0060  LISTB     EQU     *

0070  ENDB      EQU     *
0070  REF4      WORD        ENDA-LISTA+LISTC     000000
0073  REF5      WORD        ENDC-LISTC-10        FFFFF6
0076  REF6      WORD        ENDC-LISTC+LISTA-1   FFFFFF
0079  REF7      WORD        ENDA-LISTA-(ENDB-LISTB)FFFFF0
007C  REF8      WORD        LISTB-LISTA          000060
                END
```

```
0000   PROGC      START          0
                  EXTDEF     LISTC, ENDC
                  EXTREF     LISTA, ENDA, LISTB, ENDB
                  ……..
                  ……..
0018   REF1       +LDA       LISTA                          03100000
001C   REF2       +LDT       LISTB+4                        77100004
0020   REF3       +LDX       #ENDA-LISTA                    05100000
                        .
                        .
0030   LISTC      EQU        *

0042   ENDC       EQU        *
0042   REF4       WORD            ENDA-LISTA+LISTC          000030
0045   REF5       WORD            ENDC-LISTC-10             000008
0045   REF6       WORD            ENDC-LISTC+LISTA-1        000011
004B   REF7       WORD            ENDA-LISTA-(ENDB-LISTB)000000
004E   REF8       WORD            LISTB-LISTA               000000
                   END
```

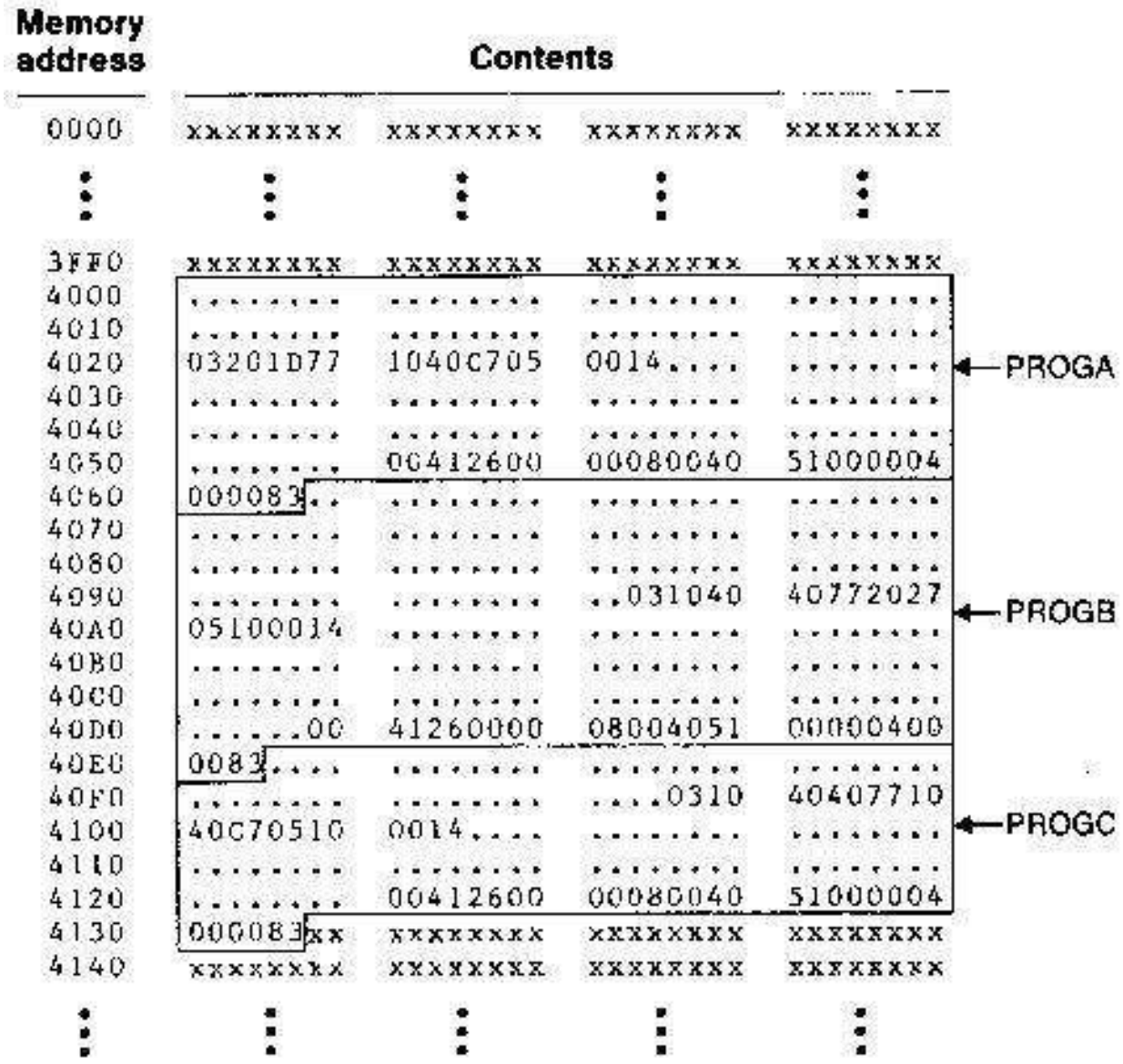
```
H PROGA 000000 000063
D LISTA   000040 ENDA   000054
R LISTB   ENDB  LISTC   ENDC
.
.
T 000020 0A 03201D 77100004 050014
.
.
T 000054 0F 000014 FFFF6 00003F 000014
FFFFC0 M000024 05+LISTB
M000054 06+LISTC
M000057 06+ENDC
M000057 06 -LISTC
M00005A06+ENDC
M00005A06 -LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

H **PROGB** 000000 00007F
D LISTB 000060 ENDB 000070
R LISTA ENDA LISTC ENDC
.
T 000036 0B 03100000 772027 05100000
.
T 000007 0F 000000 FFFFF6 FFFFFF FFFFF0 000060
M000037 05+LISTA
M00003E 06+ENDA
M00003E 06 -LISTA
M000070 06 +ENDA
M000070 06 -LISTA
M000070 06 +LISTC
M000073 06 +ENDC
M000073 06 -LISTC
M000073 06 +ENDC
M000076 06 -LISTC
M000076 06+LISTA
M000079 06+ENDA
M000079 06 -LISTA
M00007C 06+PROGB
M00007C 06-LISTA
E

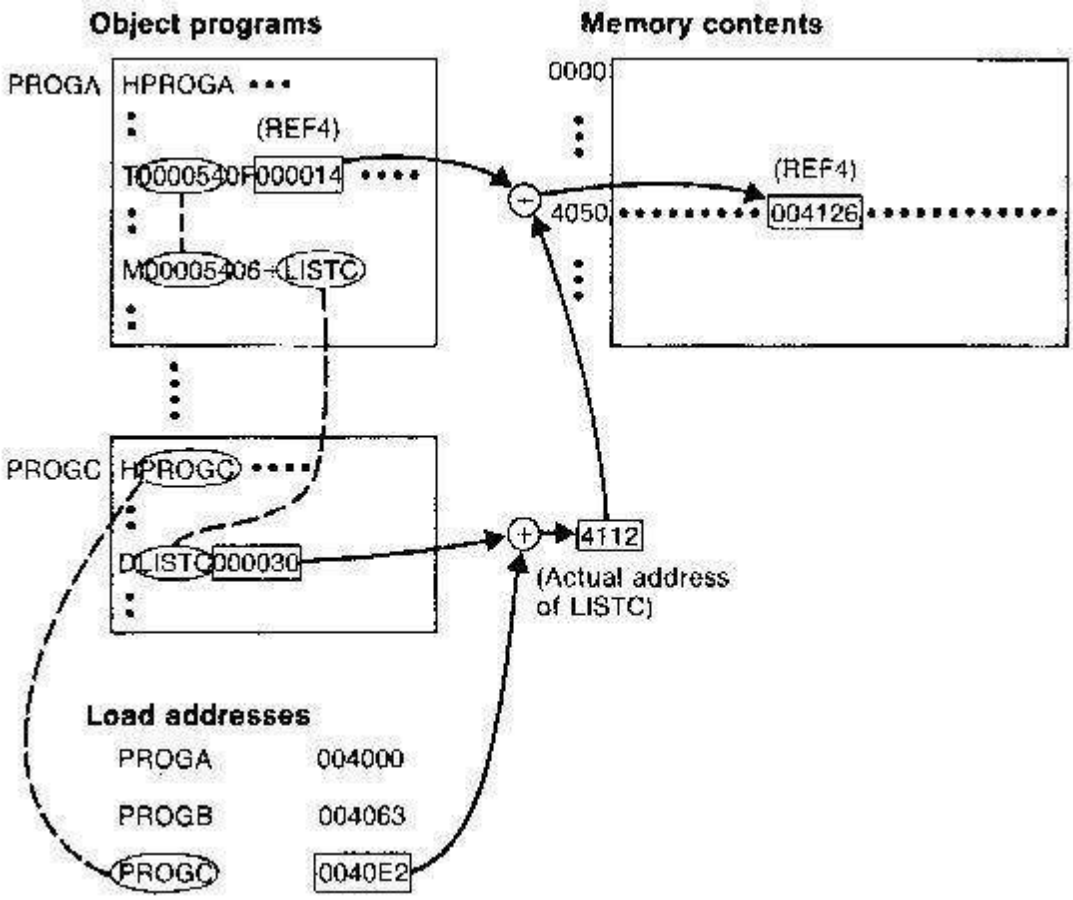
H **PROGC** 000000 000051
D LISTC 000030 ENDC 000042
R LISTA ENDA LISTB ENDB
.
T 000018 0C 03100000 77100004 05100000
.
T 000042 0F 000030 000008 000011 000000 000000
M000019 05+LISTA
M00001D 06+LISTB
M000021 06+ENDA
M000021 06 -LISTA
M000042 06+ENDA
M000042 06 -LISTA
M000042 06+PROGC
M000048 06+LISTA
M00004B 06+ENDA
M00004B 006-LISTA
M00004B 06-ENDB
M00004B 06+LISTB
M00004E 06+LISTB
M00004E 06-LISTA

E

The following figure shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately following.



For example, the value for REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054, the relative address of REF4 within PROGA). The following figure shows the details of how this value is computed.



The initial value from the Text record
T0000540F000014FFFFF600003F000014FFFFC0 is 000014. To this is added the address assigned to LISTC, which is 4112 (the beginning address of PROGC plus 30). The result is 004126.

That is REF4 in PROGA is ENDA-LISTA+LISTC=4054-4040+4112=4126.

Similarly the load address for symbols LISTA: PROGA+0040=4040, LISTB: PROGB+0060=40C3 and LISTC: PROGC+0030=4112

Keeping these details work through the details of other references and values of these references are the same in each of the three programs.

3.6 Algorithm and Data structures for a Linking Loader

The algorithm for a linking loader is considerably more complicated than the absolute loader program, which is already given. The concept given in the program linking section is used for developing the algorithm for linking loader. The modification records are used for relocation so that the linking and relocation functions are performed using the same mechanism.

Linking Loader uses two-passes logic. ESTAB (external symbol table) is the main data structure for a linking loader.

Pass 1: Assign addresses to all external symbols
Pass 2: Perform the actual loading, relocation, and linking

ESTAB - ESTAB for the example (refer three programs PROGA PROGB and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of the control section, the symbol appearing in the control section, its address and length of the control section.

Control section	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	51
	LISTC	4112	
	ENDC	4124	

3.6.1 Program Logic for Pass 1

Pass 1 assign addresses to all external symbols. The variables & Data structures used during pass 1 are, PROGADDR (program load address) from OS, CSADDR

(control section address), CSLTH (control section length) and ESTAB. The pass 1 processes the Define Record. The algorithm for Pass 1 of Linking Loader is given below.

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
    begin
        read next input record {Header record for control section}
        set CSLTH to control section length
        search ESTAB for control section name
        if found then
            set error flag {duplicate external symbol}
        else
            enter control section name into ESTAB with value CSADDR
        while record type <> 'E' do
            begin
                read next input record
                if record type = 'D' then
                    for each symbol in the record do
                        begin
                            search ESTAB for symbol name
                            if found then
                                set error flag {duplicate external symbol}
                            else
                                enter symbol into ESTAB with value
                                    (CSADDR + indicated address)
                        end {for}
            end {while <> 'E'}
        add CSLTH to CSADDR {starting address for next control section}
    end {while not EOF}
end {Pass 1}
```

3.6.2 Program Logic for Pass 2

Pass 2 of linking loader perform the actual loading, relocation, and linking. It uses modification record and lookup the symbol in ESTAB to obtain its addres. Finally it uses end record of a main program to obtain transfer address, which is a starting address needed for the execution of the program. The pass 2 process Text record and Modification record of the object programs. The algorithm for Pass 2 of Linking Loader is given below.

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin
        read next input record   {Header record}
        set CSLTH to control section length
        while record type <> 'E' do
            begin
                read next input record
                if record type = 'T' then
                    begin
                        {if object code is in character form, convert
                             into internal representation}
                        move object code from record to location
                             (CSADDR + specified address)
                    end {if 'T'}
                else if record type = 'M' then
                    begin
                        search ESTAB for modifying symbol name
                        if found then
                            add or subtract symbol value at location
                                 (CSADDR + specified address)
                        else
                            set error flag (undefined external symbol)
                    end  {if 'M'}
            end  {while <> 'E'}
        if an address is specified (in End record) then
            set EXECADDR to (CSADDR + specified address)
        add CSLTH to CSADDR
    end  {while not EOF}
jump to location given by EXECADDR (to start execution of loaded program)
end  {Pass 2}
```

3.6.3 Improve Efficiency, How?

The question here is can we improve the efficiency of the linking loader. Also observe that, even though we have defined Refer record (R), we haven't made use of it. The efficiency can be improved by the use of local searching instead of multiple searches of ESTAB for the same symbol. For implementing this we assign a reference number to each external symbol in the Refer record. Then this reference number is used in Modification records instead of external symbols. 01 is assigned to control section name, and other numbers for external reference symbols.

The object programs for PROGA, PROGB and PROGC are shown below, with above modification to Refer record (Observe R records).

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
:
T0000200A03201D771000040500014
:
T000054 0F000C014FFFFF600003F000014FFFFC0
M00002405+02
M00005406+04
M00005706+05
M00005706-04


HPROGB 00000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
:
T0000360B03100000077202705100000
:
T000070 0E000000FFFFF6FFFFFFFFFF0000060
M00003705+02
M00003E05+03
M00003E05-02
M00007006+03
M00007006-02
M00007006+04
M00007306+05
M00007306-04
M00007606+05
M00007606-04
M00007606+02
M00007906+03
M00007906-02
M00007C06+01
M00007C06-02
E
```

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
.
.
T00001800C03100000771000040510000
.
.
T0000420F000030000008000011000000000000
M00001905+02
M00001D05+04
M00002105+03
M00002105-02
M00004206+03
M00004206-02
M00004206+01
M00004806+02
M00004B06+03
M00004B06-02
M00004B06-05
M00004B06+04
M00004E06+04
M00004E06-02
E
```

Symbol and Addresses in PROGA, PROGB and PROGC are as shown below. These are the entries of ESTAB. The main advantage of reference number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section

Ref No.	Symbol	Address
1	PROGA	4000
2	LISTB	40C3
3	ENDB	40D3
4	LISTC	4112
5	ENDC	4124

Ref No.	Symbol	Address
1	PROGB	4063
2	LISTA	4040
3	ENDA	4054
4	LISTC	4112
5	ENDC	4124

Ref No.	Symbol	Address
1	PROGC	4063
2	LISTA	4040
3	ENDA	4054
4	LISTB	40C3
5	ENDB	40D3

3.7 Machine-independent Loader Features

Here we discuss some loader features that are not directly related to machine architecture and design. Automatic Library Search and Loader Options are such Machine-independent Loader Features.

3.7.1 Automatic Library Search

This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from a library as they are needed during linking. This allows programmer to use subroutines from one or more libraries. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded. The loader searches the library or libraries specified for routines that contain the definitions of these symbols in the main program.

3.7.2 Loader Options

Loader options allow the user to specify options that modify the standard processing. The options may be specified in three different ways. They are, specified using a command language, specified as a part of job control language that is processed by the operating system, and an be specified using loader control statements in the source program.

Here are the some examples of how option can be specified.

INCLUDE program-name (library-name) - read the designated object program from a library

DELETE csect-name – delete the named control section from the set pf programs being loaded

CHANGE name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs

LIBRARY MYLIB – search MYLIB library before standard libraries

NOCALL STDDEV, PLOT, CORREL – no loading and linking of unneeded routines

Here is one more example giving, how commands can be specified as a part of object file, and the respective changes are carried out by the loader.

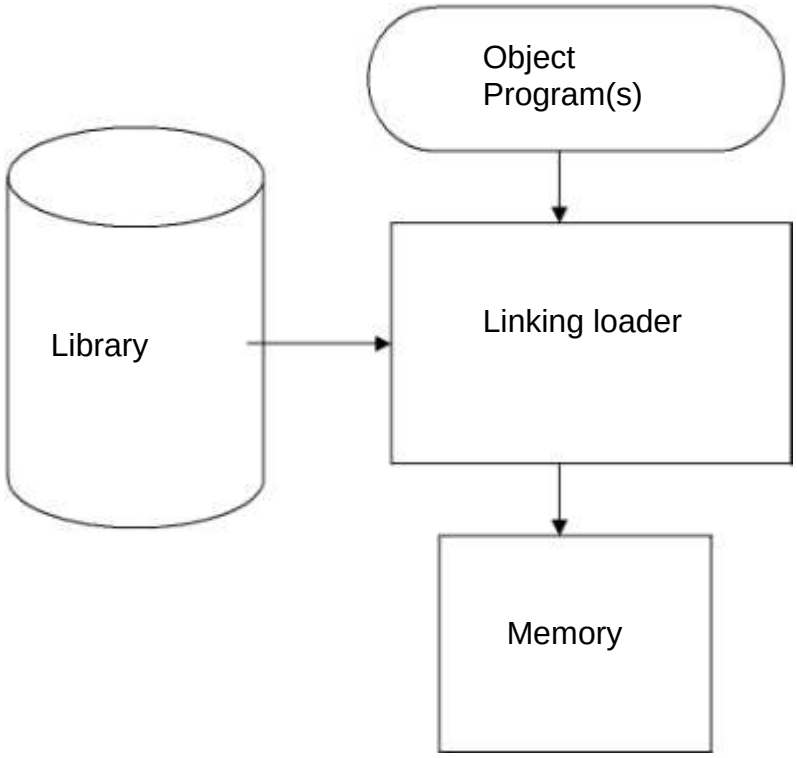
```
LIBRARY UTLIB
INCLUDE READ (UTLIB)
INCLUDE WRITE (UTLIB)
DELETE RDREC, WRREC
CHANGE RDREC, READ
CHANGE WRREC, WRITE
NOCALL  SQRT, PLOT
```

The commands are, use UTLIB (say utility library), include READ and WRITE control sections from the library, delete the control sections RDREC and WRREC from the load, the change command causes all external references to the symbol RDREC to be changed to the symbol READ, similarly references to WRREC is changed to WRITE, finally, no call to the functions SQRT, PLOT, if they are used in the program.

3.8 Loader Design Options

There are some common alternatives for organizing the loading functions, including relocation and linking. Linking Loaders – Perform all linking and relocation at load time. The Other Alternatives are Linkage editors, which perform linking prior to load time and, Dynamic linking, in which linking function is performed at execution time

3.8.1 Linking Loaders

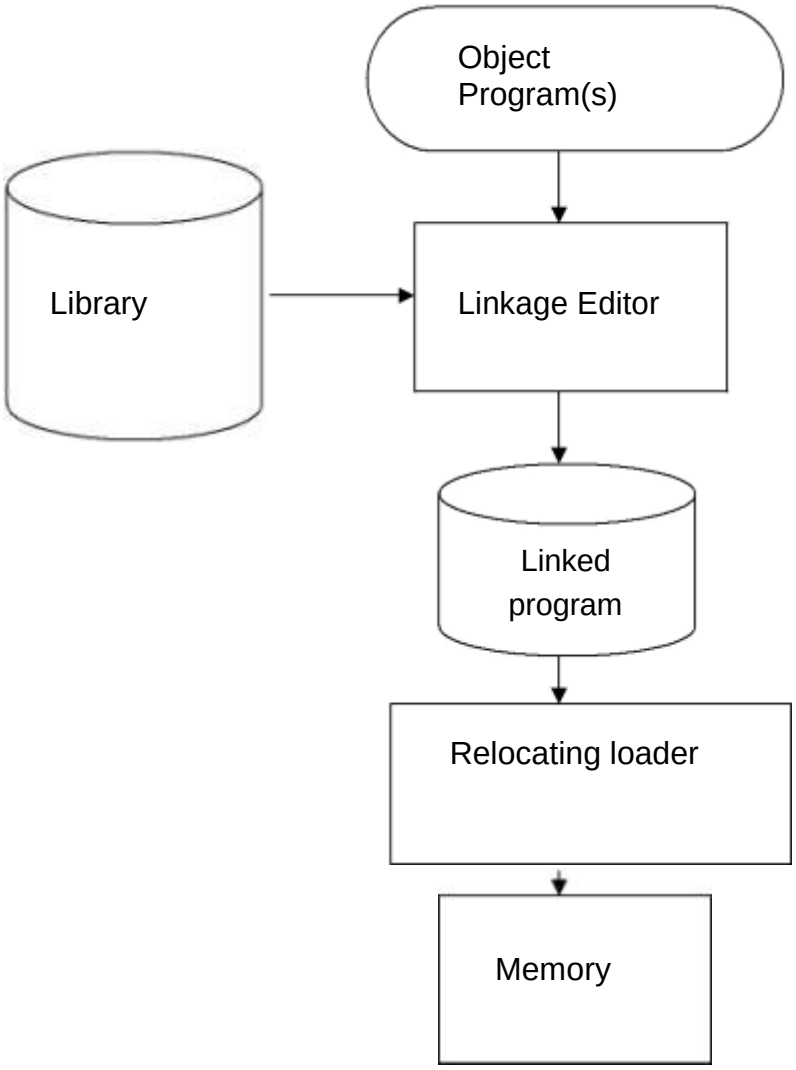


The above diagram shows the processing of an object program using Linking Loader. The source program is first assembled or compiled, producing an object program. A linking loader performs all linking and loading operations, and loads the program into memory for execution.

3.8.2 Linkage Editors

The figure below shows the processing of an object program using Linkage editor. A linkage editor produces a linked version of the program – often called a load module or an executable image – which is written to a file or library for later execution. The linked program produced is generally in a form that is suitable for processing by a relocating loader.

Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known. New versions of the library can be included without changing the source program. Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search – linking will be done later by linking loader – linkage editor + linking loader – savings in space



3.8.3 Dynamic Linking

The scheme that postpones the linking functions until execution. A subroutine is loaded and linked to the rest of the program when it is first called – usually called dynamic linking, dynamic loading or load on call. The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library. In an object oriented system, dynamic linking makes it possible for one object to be shared by several programs. Dynamic linking provides the ability to load the routines only when (and if) they are needed. The actual loading and linking can be accomplished using operating system service request.

3.8.4 Bootstrap Loaders

If the question, how is the loader itself loaded into the memory ? is asked, then the answer is, when computer is started – with no program in memory, a program present in ROM (absolute address) can be made executed – may be OS itself or A Bootstrap loader, which in turn loads OS and prepares it for execution. The first record (or records) is generally referred to as a bootstrap loader – makes the OS to be loaded. Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system.

3.9 Implementation Examples

This section contains brief description of loaders and linkers for actual computers. They are, MS-DOS Linker - Pentium architecture, SunOS Linkers - SPARC architecture, and, Cray MPP Linkers – T3E architecture.

3.9.1 MS-DOS Linker

This explains some of the features of Microsoft MS-DOS linker, which is a linker for Pentium and other x86 systems. Most MS-DOS compilers and assemblers (MASM) produce object modules, and they are stored in .OBJ files. MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program - .EXE file; this file is later executed for results.

The following table illustrates the typical MS-DOS object module

Record Types	**Description**
THEADR	Translator Header
TYPDEF,PUBDEF, EXTDEF	External symbols and references
LNAMES, SEGDEF, GRPDEF	Segment definition and grouping
LEDATA, LIDATA	Translated instructions and data
FIXUPP	Relocation and linking information
MODEND	End of object module

THEADR specifies the name of the object module. MODEND specifies the end of the module. PUBDEF contains list of the external symbols (called public names). EXTDEF contains list of external symbols referred in this module, but defined elsewhere. TYPDEF the data types are defined here. SEGDEF describes segments in the object module (includes name, length, and alignment). GRPDEF includes how segments are combined into groups. LNAMES contains all segment and class names. LEDATA contains translated instructions and data. LIDATA has above in repeating pattern. Finally, FIXUPP is used to resolve external references.

3.9.2 SunOS Linkers

SunOS Linkers are developed for SPARC systems. SunOS provides two different linkers – link-editor and run-time linker.

Link-editor is invoked in the process of assembling or compiling a program – produces a single output module – one of the following types

A relocatable object module – suitable for further link-editing

A static executable – with all symbolic references bound and ready to run

A dynamic executable – in which some symbolic references may need to be bound at run time

A shared object – which provides services that can be, bound at run time to one ore more dynamic executables

An object module contains one or more sections – representing instructions and data area from the source program, relocation and linking information, external symbol table.

Run-time linker uses dynamic linking approach. Run-time linker binds dynamic executables and shared objects at execution time. Performs relocation and linking operations to prepare the program for execution.

3.9.3 Cray MPP Linker

Cray MPP (massively parallel processing) Linker is developed for Cray T3E systems. A T3E system contains large number of parallel processing elements (PEs) – Each PE has local memory and has access to remote memory (memory of other PEs). The processing is divided among PEs - contains shared data and private data. The loaded program gets copy of the executable code, its private data and its portion of the shared data. The MPP linker organizes blocks containing executable code, private data and shared data. The linker then writes an executable file that contains the relocated and linked blocks. The executable file also specifies the number of PEs required and other control information. The linker can create an executable file that is targeted for a fixed number of PEs, or one that allows the partition size to be chosen at run time. Latter type is called plastic executable.

The Structure of a Compiler

A compiler performs two major tasks:

3. Analysis of the source program being compiled
4. Synthesis of a target program

Almost all modern compilers are *syntax-directed:* The compilation process is driven by the syntactic structure of the source program.

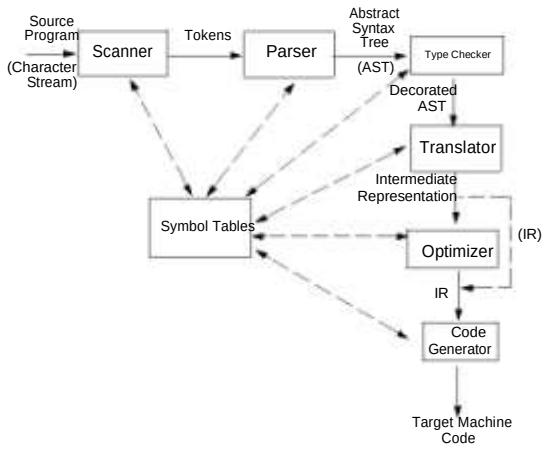
A parser builds semantic structure out of tokens, the elementary symbols of programming language syntax. Recognition of syntactic structure is a major part of the analysis task.

If an IR is generated, it then serves as input to a *code generator* component that produces the desired machine-language program. The IR may optionally be transformed by an *optimizer* so that a more efficient program may be generated.

Semantic analysis examines the meaning (semantics) of the program. Semantic analysis plays a dual role.

It finishes the analysis task by performing a variety of correctness checks (for example, enforcing type and scope rules). Semantic analysis also begins the synthesis phase.

The synthesis phase may translate source programs into some intermediate representation (IR) or it may directly generate target code.



The Structure of a Syntax-Directed Compiler

Scanner

The scanner reads the source program, character by character. It groups individual characters into tokens (identifiers, integers, reserved words, delimiters, and so on). When necessary, the actual character string comprising the token is also passed along for use by the semantic phases.

The scanner:

3. Puts the program into a compact and uniform format (a stream of tokens).
4. Eliminates unneeded information (such as comments).
5. Sometimes enters preliminary information into symbol tables (for example, to register the presence of a particular label or identifier).
   - Optionally formats and lists the source program

Building tokens is driven by token descriptions defined using *regular expression* notation.

Regular expressions are a formal notation able to describe the tokens used in modern programming languages. Moreover, they can drive the *automatic generation* of working scanners given only a specification of the tokens. Scanner generators (like Lex, Flex and Jlex) are valuable compiler-building tools.

Parser

Given a syntax specification (as a context-free grammar, CFG), the parser reads tokens and groups them into language structures.

Parsers are typically created from a CFG using a parser generator (like Yacc, Bison or Java CUP).

The parser verifies correct syntax and may issue a syntax error message.

As syntactic structure is recognized, the parser usually builds an abstract syntax tree (AST), a concise representation of program structure, which guides semantic processing.

Type Checker (Semantic Analysis)

The type checker checks the *static semantics* of each AST node. It verifies that the construct is legal and meaningful (that all identifiers involved are declared, that types are correct, and so on).

If the construct is semantically correct, the type checker "decorates" the AST node, adding type or symbol table information to it. If a semantic error is discovered, a suitable error message is issued.

Type checking is purely dependent on the semantic rules of the source language. It is independent of the compiler's target machine.

Translator (Program Synthesis)

If an AST node is semantically correct, it can be translated. Translation involves capturing the run-time "meaning" of a construct.

For example, an AST for a while loop contains two subtrees, one for the loop's control expression, and the other for the loop's body. *Nothing* in the AST shows that a while loop loops! This "meaning" is captured when a while loop's AST is translated. In the IR, the notion of testing the value of the loop control expression,

and conditionally executing the loop body becomes explicit.

The translator is dictated by the semantics of the source language. Little of the nature of the target machine need be made evident. Detailed information on the nature of the target machine (operations available, addressing, register characteristics, etc.) is reserved for the code generation phase.

In simple non-optimizing compilers (like our class project), the translator generates target code directly, without using an IR.

More elaborate compilers may first generate a high-level IR

(that is source language oriented) and then subsequently translate it into a low-level IR (that is target machine oriented). This approach allows a cleaner separation of source and target dependencies.

Optimizer

The IR code generated by the translator is analyzed and transformed into functionally equivalent but improved IR code by the optimizer.

The term optimization is misleading: we don't always produce the best possible translation of a program, even after optimization by the best of compilers.

Why?

Some optimizations are *impossible* to do in all circumstances because they involve an undecidable problem. Eliminating unreachable ("dead") code is, in general, impossible.

Other optimizations are too expensive to do in all cases. These involve NP-complete problems, believed to be inherently exponential. Assigning registers to variables is an example of an NP-complete problem.

Optimization can be complex; it may involve numerous subphases, which may need to be applied more than once.

Optimizations may be turned off to speed translation. Nonetheless, a well designed optimizer can significantly speed program execution by simplifying, moving or eliminating unneeded computations.

Code Generator

IR code produced by the translator is mapped into target machine code by the code generator. This phase uses detailed information about the target machine and includes machine-specific optimizations like *register allocation* and *code scheduling*.

Code generators can be quite complex since good target code requires consideration of many special cases.

Automatic generation of code generators is possible. The basic approach is to match a low-level IR to target instruction templates, choosing

instructions which best match each IR instruction.

A well-known compiler using automatic code generation techniques is the GNU C compiler. GCC is a heavily optimizing compiler with machine description files for over ten popular computer architectures, and at least two language front ends (C and C++).

Symbol Tables

A symbol table allows information to be associated with identifiers and shared among compiler phases. Each time an identifier is used, a symbol table provides access to the information collected about the identifier when its declaration was processed.

Example

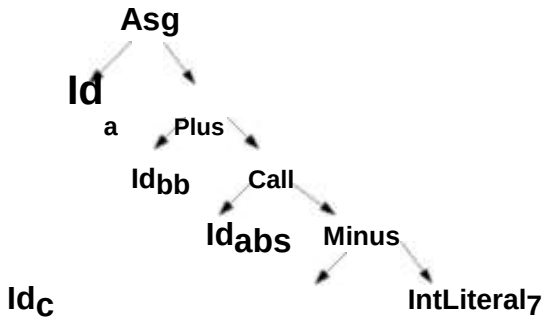
Our source language will be *CSX*, a blend of C, C++ and Java.

Our target language will be the Java JVM, using the Jasmin assembler.

- Our source line is
  ```
  a = bb+abs(c-7);
  ```
  this is a sequence of ASCII characters in a text file.
- The scanner groups characters into tokens, the basic units of a program.
  ```
  a = bb+abs(c-7);
  ```
  After scanning, we have the following token sequence:

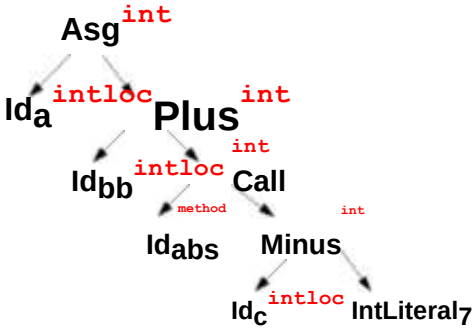
  Id_a Asg Id_{bb} Plus Id_{abs} Lparen Id_c Minus $IntLiteral_7$ Rparen Semi

- The parser groups these tokens into language constructs (expressions, statements, declarations, etc.) represented in tree form:



(What happened to the parentheses and the semicolon?)

- The type checker resolves types and binds declarations within scopes:




```
int   ; compute bb+abs(c-7)
  1 ; store result into
  local 1(a)
```

- Finally, JVM code is generated for each node in the tree (leaves first, then roots):

```
iload   3 ; push local 3 (bb)
iload   2 ; push local 2 (c)
ldc     7 ; Push literal 7
isub      ; compute c-7
invokestatic java/lang/Math/ abs(I)I
iadd
```

Interpreters

There are two different kinds of interpreters that support execution of programs, *machine interpreters* and *language interpreters*.

Machine Interpreters

Machine interpreters simulate the execution of a program compiled for a particular machine architecture. Java uses a *bytecode interpreter* to simulate the effects of programs compiled for the JVM. Programs like SPIM simulate the execution of a MIPS program on a non-MIPS computer.

Language Interpreters

Language interpreters simulate the effect of executing a program without compiling it to any particular instruction set (real or virtual). Instead some IR form (perhaps an AST) is used to drive execution.

Interpreters provide a number of capabilities not found in compilers:

- Programs may be modified as execution proceeds. This provides a straightforward interactive debugging capability. Depending on program structure, program modifications may require reparsing or repeated semantic analysis. In Python, for example, any string variable may be interpreted as a Python expression or statement and executed.

- Interpreters readily support languages in which the type of a variable denotes may change dynamically (e.g., Python or Scheme). The user program is continuously reexamined as execution proceeds, so symbols need not have a fixed type. Fluid bindings are much more troublesome for compilers, since dynamic changes in the type of a symbol make direct translation into machine code difficult or impossible.
- Interpreters provide better diagnostics. Source text analysis is intermixed with program execution, so especially good diagnostics are available, along with interactive debugging.
- Interpreters support machine independence. All operations are performed within the interpreter. To move to a new machine, we just recompile the interpreter.

However, interpretation can involve large overheads:

- As execution proceeds, program text is continuously reexamined, with bindings, types, and operations sometimes recomputed at each use. For very dynamic languages this can represent a 100:1 (or worse) factor in execution speed over compiled code. For more static languages (such as C or Java), the speed degradation is closer to 10:1.
- Startup time for small programs is slowed, since the interpreter must be load and the program partially recompiled before execution begins.

- Substantial space overhead may e involved. The interpreter and all support routines must usually be kept available. Source text is often not as compact as if it were compiled. This size penalty may lead to restrictions in the size of programs. Programs beyond these built-in limits cannot be handled by the interpreter.

Of course, many languages (including, C, C++ and Java) have both interpreters (for debugging and program development) and compilers (for production work).

UNIT 5

5. Introduction:

–

C
o
m
p
u
t
i
n
g

I
n
v

olve two main activities: 0 Program Devel

o
p
m
e
n
t

0 Use of application software

– Programs that help us for developing and using other programs, are called software tools, which perform various house keeping task involved in program development & application usage.

0 Definition: Software Tool is a system program which

– 1. Interface program with the entity generating its input data, or

– 2. Interfaces the results of a program with the entity(user) consuming them.

0 The entity generating the data or consuming the results may be
– A program or
– A user.

Example: File rewriting utility

0 It organizes the data in file to make it suitable for processing by a program. Eg. Converting pipe delimited file to excel file.

- For this, it may perform:
  – Blocking/De-Blocking,
  – Padding / Truncation,
  – Sorting, etc.

DEVELOPMENT

b) Program design, coding and documentation

c) Preparation of program to machine readable form.
d) Program translation, linking & loading.
e) Program testing & debugging.
f) Performance Tuning.

g) Reformatting data and/or results of a program to suit other programs.

Software Tools:

4. In this topic we will cover following points in detail:

Program Design & Coding
Program Entry & Editing
Program Testing & Debugging
Enhancement of Program Performance.
Program Documentation
Design of Software Tools

- Program Pre-Processing
- Program Instrumentation
- Program Interpretation
- Program Generation.

(a) Program Design & Coding

s or more sophisticated programs with text editors as front ends.

5. Two modes of Editor Function:
   – Command Mode
   – Data Mode

These tools are text editor

0 Command Mode: Accepts user commands specifying function to be performed. 0 Data Mode: In this user "keys in" text to be added to file.

0 What is the problem here?
   – Failure to recognize current mode.
   – Results to mixing up of command and data mode.

0 Solution / Remedy: Two Approaches
   – Approach 1: Quick Exit
      Quick exit from data mode through ESC key.
      0 Eg: Vi Editor.
   – Approach 2: Screen Mode
      0 Also called "what u see is what u get" mode.
      0 Cursor works for command mode.
      0 And key stroke is the data mode.
      0 Special key combination signify commands.

      0 Benefit: Need not to indicate that data input ends. 0 Eg: Turbo C

PROGRAM EDITING AND TESTING

4. Steps for program testing and debugging:
   – 1. Selection of test data for programs.
   – 2. Analysis of test results to detect errors.
   – 3. Debugging.

- Software testing and debugging to programmers may of following forms:
  - 1. Test Data Generators
  - 2. Automated Test Drivers
  - 3. Debug Monitors
  - 4. Source Code Control System

Test Data Generators:

- Help user selecting test data for programs.
- Ensure thoroughly testing of programs.

Debug Monitors:

- Helps us in obtaining information for localization of errors.

Source Code Control System:

- Helps keep track of modifications in the source code.

Software Testing Tools assisting for t

tomated Test Drivers:

- Help in regression testing.
- After every modification, program correctness is verified by subjecting it to standard set of tests.
- Regression Testing:
  - Many sets are prepared as test data and then,
  - Given as input to programs.
  - The driver then selects one set of test data at a time and organizes execution of program on that data.
- Problems with Automated Test Drivers:

0 1. Wastage of testing efforts on in-feasible paths. i.e Path which is never followed during execution . ◦ 2. Testing complex loop structures. Remedy/Solution: Manual Testing

Conclusion:

Automated testing can be used for first batch of errors at

Testing:--

0 Execution Path:

0 Used by test data selection

0 It is a sequence of program statements visited during an execution.

0 For testing, program is viewed as set of execution path.

0 Test data generator determines conditions which must be satisfied by program input for flow of control along the specific execution path.

0 Eg:

```
x:= sqrt(y) + 2.5; if x>z
        then a:=a+1.0; else

        --
```

0 Execution path is traversed only if x>z.

0 To test this, test data generator must select a value for y and z such that x>z.

0 For eg: $y = z^2$, then only z value is to be selected.

0

0 1

0 Trace off: Trace is disabled.

0 Display <list>: Values of variable in list are written in debug file.

Debugging:-

- To improve effectiveness of debugging, get option of "Dynamically specify trace & dump".
- Who would provide this solution?
  - Ans: Compiler and Interpreters.
- How?
  - By setting breakpoints (stop)
  - Change variables during debugging.
  - Set & remove breakpoints dynamically during execution.
- Eg: Break Points & Dump Facilities
  - (a) stop on <list of labels> (b) dump at <label> <list of variables>
- Eg: Interactive Debugging Facilities:
  - This commands can be issued when program reaches breakpoints.
  - Display <list of variables>
    - Displays values of variables.
  - Set <variable> = <exp>
    - Assign value.
  - Resume:
    - Resumes execution.
  - Run:
    - Restarts execution.
- Debug Monitors:
  - Debugging is provided by debug monitor software.
  - Executes program that is being debugging under its own control.
  - Provides execution efficiency.
  - Performs dynamically specified debugging actions.
  - Can be made language independent.
  - Eg: DDT (Dynamic Debugging Techniques) of DEC-10 language.

EDITORS

- 1. Line
- 2. Stream

- 3. Screen
- 4. Word Processor
- 5. Structure Editor

- Line & stream editors typically maintain multiple representation of text.
- Display Form shows text of sequence of lines.
- Internal Form performs edit operation.

LINE EDITORS

- Scope of edit operation in line editor is limited to line of text. Line is designated either
  - Positional (sr. no) or
  - Contextually (context symbol)
- Advantage: Primary Simplicity.

STREAM EDITORS

- Views entire text as a stream of characters.
- Permit edit operation to cross line boundaries.
- Support character, line and context oriented commands. Indicated by
  - Position of text pointer or
  - Search commands.
- Eg: Dos File

SCREEN EDITORS

- Doesn‟t display text in the manner in which it would appear for printing. Uses “what u see is what u get” principle.
- Displays screen full of text at a time.
- Cursor positioning & editing is supported.
- Effect of edit operation is visible on the screen same moment.
- Useful during formatting of documents to be printed.
- Eg: NotePad, WordPad.

STRUCTURE EDITORS

- Basically document editors
- Produce well formatted hard copy output.
- Features:
  - Copy / Paste / Cut
  - Find / Replace
  - Spell Check
- Wide spread among
  - Authors
  - Office Personnel
  - Computer Professionals
- Eg: Word Start, MS word

- Incorporates an awareness of structure of document.
- Useful in browsing through document.
- Creation & Modification is very easy.
- Editing requirements are specified using structures.
- Structure editors are also called syntax directed editors.
- Eg: VB, NetBeans, EditPlus

DEBUG MONITORS

Provide Following facilities:-

Setting breakpoints in program

Initializing debug conversation when control reaches breakpoint

Displaying values of variables.

Testing user defined assertions & predicts involving program variables.

- Debug monitor functions can be easily implemented in an interpreter.
- But interpreters incur considerable execution time penalties. Hence, debug monitors rely on instrumentation of program. User must compile program under debug option.
- To disable debug option for particular statements: no_op<statement number>
- Compiler generates table (var nm, address).
- Set break points with <si_Instruction><code>
- Program executes on CPU until si_instruction is reached.

- Code produces interrupt code.
- Instead of <si_instruction> we can use
  - BC ANY
  - DEBUG_MON

- Now debug monitor gains control & opens a debug conversation.

PROGRAM ENVIRONMENT

- Debug Assertion: is a relation between values of program variables.
- Assertion can be associated by program statement.
- Debug monitors verifies assertion when execution reaches the statements.
- If values matches, program execution continues otherwise, debug conversation opens.
- Debug assertion eliminates need to produce voluminous information.

- Is a system software.
- Provides integral facilities for
  - Program creation
  - Editing
  - Execution
  - Testing
  - Debugging
- Components
  1. Syntax directed editor
  2. Language Processor
     - Compiler
     - Interpreter

- Both

3. Debug monitor
4. Dialog monitor

- Components are accessed through dialog monitor.
- Syntax directed editor incorporates front end.
- Editor performs syntax analysis and construct IR giving abstract syntax tree.
- Compiler and debug monitor share IR.
- Thus, program execution or interpretation starts then.
- In between, any time during execution, programmer can interrupt execution, resume program and restart execution. Debug monitors provide easy accessibility to all functions.
- Also allow,
  - Generation of program
  - Testing functions
  - Partial compilation and program execution.
- Errors are indicated if encountered.
- Also permits
  - reversible execution &
  - step back execution

USER INTERFACE

- Role: Simplifying interaction of a user with an application. Two aspects:
  - 1. Issuing Commands.
  - 2. Exchange of Data.
- Requirement of UI?
  - Before requirement was small & hence small applications.
  - But now, applications have become so large that one programmer‟s team do not know any thing about other programmer‟s team working on same application.
  - UI will keep track of information of functionalities & educating users for those applications.

- Two components:
  - 1.Dialog Manager
  - 2.Presentation Manager
- 1. Dialog Manager:
  - Manages conversation between user & application.
  - Prompts user for command.
  - Transmits command to application.
- 2. Presentation Manager:
  - Displays data produced by the application in appropriate manner on screen.